XNET: A Real-Time Unified Secure Inference Framework Using Homomorphic Encryption

Hao Yang[1], Shiyu Shen[2], Siyang Jiang[3], Lu Zhou[1], Wangchen Dai[4], and Yunlei Zhao[2]

[1] Nanjing University of Aeronautics and Astronautics, Nanjing, China
crypto@d4rk.dev
[2] Fudan University, Shanghai, China, shenshiyu21@m.fudan.edu.cn
[3] Chinese University of Hong Kong, Hong Kong, China
[4] Zhejiang Lab, Hangzhou, China

Abstract. Homomorphic Encryption (HE) presents a promising solution to securing neural networks for Machine Learning as a Service (MLaaS). Despite its potential, the real-time applicability of current HE-based solutions remains a challenge, and the diversity in network structures often results in inefficient implementations and maintenance. To address these issues, we introduce a unified and compact network structure for real-time inference in convolutional neural networks based on HE. We further propose several optimization strategies, including an innovative compression and encoding technique and rearrangement in the pixel encoding sequence, enabling a highly efficient batched computation and reducing the demand for time-consuming HE operations. To further expedite computation, we propose a GPU acceleration engine to leverage the massive thread-level parallelism to speed up computations. We test our framework with the MNIST, Fashion-MNIST, and CIFAR-10 datasets, demonstrating accuracies of 99.14%, 90.8%, and 61.09%, respectively. Furthermore, our framework maintains a steady processing speed of 0.46 seconds on a single-thread CPU, and a brisk 31.862 milliseconds on an A100 GPU for all datasets. This represents an enhancement in speed more than 3000 times compared to pervious work, paving the way for future explorations in the realm of secure and real-time machine learning applications.

Keywords: Homomorphic Encryption · Convolutional Neural Network · Secure inference · GPU acceleration.

1 Introduction

The Convolutional Neural Network (CNN), designed to extract discriminating features that unveil inherent correlations within a data array, is extensively utilized in the domain of image analysis. In contexts necessitating engagement with substantial quantities of input data, Machine Learning as a Service (MLaaS) is frequently favored. This involves users assigning inference tasks to servers, subsequently capitalizing on the potent cloud infrastructure for efficient data analytics. In such an arrangement, the models are often deployed on cloud-based or

edge device environments. Here, they generate predictions or inferences hinging upon fresh inputs, conducting computations on enormous data sets that encompass personal, medical, financial, and other categories of sensitive information. While this has paved the way for manifold opportunities for businesses and society, it has also simultaneously triggers considerable apprehensions concerning data privacy.

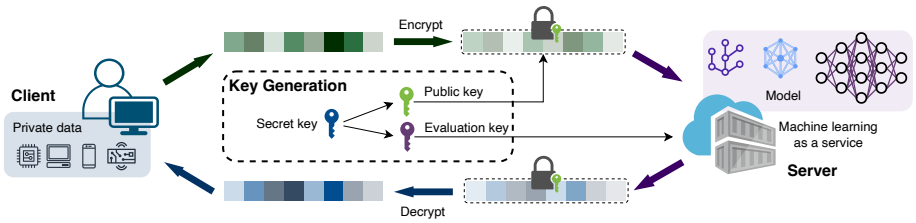


Fig. 1: The computational model of privacy-preserving Machine Learning as a Service leveraging Homomorphic Encryption.

In order to alleviate these apprehensions, a variety of privacy-preservation techniques are being devised and employed, consequently giving rise to the concept of privacy-preserving inference. The privacy-preserving neural network (PPNN) emerges as a prospective solution for executing inference on data of a sensitive nature, leveraging cryptographic tools to prevent potential information leaks and subsequently alleviate privacy-related concerns. A multitude of recent studies have proposed several frameworks for PPNN computation, based on Homomorphic Encryption (HE), Multi-Party Computation (MPC), or an amalgamated approach of HE and MPC. The MPC-oriented methods [3, 14, 18] necessitate client participation in the computation and require data transfers of several gigabytes during the inference process. The HE-enabled Convolutional Neural Network (HCNN) [1, 8, 11, 17] innately facilitates direct evaluation over ciphertexts, conferring numerous benefits such as one-shot communication, low bandwidth usage, and non-interactive computation. Specifically, the client only needs to engage with the server once to supply the data and can remain offline throughout the evaluation process.

In spite of continuous advancements in this domain, the existing solutions exhibit significant constraints. A multitude of studies are dedicated to the optimization and acceleration of HCNN by refining message encoding techniques, network structures, or implementing strategies such as hardware acceleration to capitalize on parallelism for enhanced efficiency. Nevertheless, these efforts are yet to satisfy the pressing demands of real-time applications. Firstly, the employed models are often intricate and diverse, necessitating model-specific implementations. The proposed models also have dependency on the specific datasets being processed, thereby undermining the feasibility of a unified and universally applicable network structure. Secondly, the diversity of computa-

tions across various types of model layers poses significant challenges, particularly when adapting homomorphic evaluations over ciphertexts. Finally, the inherent nature of homomorphic encryption exacerbates these challenges by imposing high computational and memory overheads. This translates into extensive resource requirements even for basic operations, thereby limiting the practical application of such methodologies in contexts where computational resources or time frames are restricted. Even the optimized low-latency framework [8] requires 730 seconds along with 12 GB of RAM for a single prediction in the CIFAR-10 dataset [15], wherein the input is a 32×32 pixel figure. Meanwhile, efforts focusing on GPU-acceleration [1] indicate a time frame of 304.43 seconds for the same CIFAR-10 dataset. Thus, while HCNN represents a promising avenue in the context of PPNN, these constraints highlight the pressing need for further refinement and innovation within the field.

Contribution. In this work, we present an optimized HE-based inference framework specifically designed for real-time application in convolutional neural networks. Our contributions are summarized as follows:

- We develop a unified and compact structure that is well-suited for HE applications. This structure efficiently combines multiple operations, making it practical and easy to use.
- We propose several optimization strategies to improve efficiency. We devise an optimized compression and encoding method that reduces the overhead of processing 3-channel color images. Simultaneously, we meticulously arrange the encoding sequence of pixels to allow highly batched computations across different layers, substantially reducing the number of time-demanding HE operations like homomorphic multiplication and rotation.
- Based on this, we harness the power of massive thread-level parallelism inherent in GPUs to further improve the efficiency. We introduce a GPU acceleration engine that significantly speeds up computations, enabling a substantial speed increase.
- We test our framework on three datasets, including MNIST [16], Fashion-MNIST [19], and CIFAR-10 [15], achieving an accuracy of 99.14%, 90.8%, and 61.09%, respectively. Meanwhile, we maintain a steady processing speed of 0.46 seconds on a single-thread CPU and 31.862 milliseconds on an A100 GPU. This represents a speedup of over 3000 times compared to previous work, demonstrating the high efficiency and effectiveness of our framework.

Related Work. The domain of private predictions has received substantial scholarly interest in recent years, with several noteworthy contributions to the field. Initial research in this area adopted a solution based on batch encrypted processing. CryptoNets, introduced by Gilad-Bachrach et al. [11], marked the first instance of a privacy-preserving encrypted prediction as a service solution. Despite leveraging YASHE [6], an HE scheme now considered insecure, CryptoNets demonstrated the capacity to deliver predictions with an accuracy of

98.95% on the MNIST data set. Building upon this, Boemer et al. [5] proposed nGraph-HE, an HE-based extension to the Intel nGraph deep learning compiler. The nGraph-HE approach echoed the techniques used in CryptoNets, although different homomorphic encryption schemes, namely BFV [7, 12] and CKKS [9], were employed. An optimized version of this work, termed nGraph-HE2 [4], subsequently addressed the relatively shallow network limitations of nGraph-HE and extended its scope to facilitate privacy-preserving inference on standard pretrained models, utilizing their inherent activation functions and number fields. Badawi et al. [1] contributed to the field by presenting a GPU implementation designed to accelerate execution.

These aforementioned works collectively emphasized a high-throughput design, wherein each node in the network is encrypted as a separate ciphertext. They advocated batch processing of all images to be inferred, an approach that can cause high memory overhead when applied to larger networks. Despite boasting a small amortized time, such methods are not ideal when the number of images for inference is limited. Contrasting with this line of work, Brutzkus et al. [8] proposed a low latency privacy-preserving inference method, termed LoLa. LoLa introduces an innovative data batching method that encrypts entire layers and alters the data representation during computation, enabling efficient low latency inference for a single image. Building upon the LoLa concept, Lou et al. [17] observed the excessive homomorphic rotations in LoLa and designed a homomorphic discrete Fourier transform algorithm to shift the ciphertexts to the spectral domain, effectively reducing the overall number of rotations.

2 Preliminaries

2.1 Notation

In this work, we employ the following notations and mathematical concepts. We denote by q an integer, and by N a power of two. The set of integers is represented by $\mathbb{Z}$, and $\mathbb{Z}_q$ stands for integers modulo q. We introduce the polynomial ring $R = \mathbb{Z}[X]/(X^N + 1)$ and the residue ring modulo q as $R_q = \mathcal{R}/q\mathcal{R}$. Polynomials, which are elements of the ring $\mathcal{R}$, are denoted by bold, italic lowercase letters. For instance, $\boldsymbol{f} := \sum_{i=0}^{N-1} f_i X^i$. The symbol $\boldsymbol{f} \leftarrow \mathcal{S}$ signifies that the polynomial $\boldsymbol{f}$ is chosen according to a specific distribution $\mathcal{S}$. We represent the encryption function as $[\![\cdot]\!]$ to denote $\mathsf{Enc}_{\mathbf{pk}}(\cdot)$ under the public key $\mathbf{pk}$, and the decryption function as $\mathsf{Dec}_{\mathbf{sk}}(\cdot)$ under the secret key $\mathbf{sk}$. We use the notation $[a]_q$ to signify the integer equivalent of a modulo q. We denote the width and height of an image as W and H, respectively, and the filter size as $F \times F$. Regarding the homomorphic operations, we assign particular notations for clarity: CAdd signifies the addition between ciphertext and plaintext, HAdd denotes the addition of two ciphertexts, CMult represents the multiplication between ciphertext and plaintext, while HMult corresponds to the multiplication of two ciphertexts. Additionally, the rotation operation with a step size of i is denoted as HRot_i.

2.2 Residue Number System

The Residue Number System (RNS) is a numerical representation that can enhance parallelism and modularity in computations. In RNS, we can represent a large integer belonging to $\mathbb{Z}_Q$ as a set of smaller residues in $\mathbb{Z}_{q_i}$, where $Q := \Pi_{i=0}^{L-1} q_i$. Consequently, a ring element $\boldsymbol{x} \in R_Q$ is represented as:

$$[\boldsymbol{x}]_{Q_L} = (\boldsymbol{x}^{(0)}, \boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(L-1)}) \in R_{q_0} \times R_{q_0} \times \cdots \times R_{q_{L-1}}$$

The base of the RNS can be switched from Q to $R := \Pi_{i=0}^{K-1} r_i$ using the following method [2], assuming the small moduli are pairwise co-prime:

$$\texttt{Conv}_{Q \to R}(\boldsymbol{x}) = \left(\left[\sum_{i=0}^{L-1} [\boldsymbol{x}_i \cdot \tilde{q}_i]_{q_i} \cdot q_i^* \right]_{r_j} \right)_{j=1}^{K-1}$$

Where $\boldsymbol{x}^{(i)} := [\boldsymbol{x}]_{q_i}$, $i \in [0, L)$.

In the light of this formulation, we can define methods for modulus extension and reduction:

- $\texttt{ModUp}_{Q \to QR}([\boldsymbol{x}]_Q) := ([\boldsymbol{x}]_Q, \texttt{Conv}_{\mathcal{Q} \to \mathcal{B}}([\boldsymbol{x}]_Q))$
- $\texttt{ModDown}_{QR \to Q}([\boldsymbol{x}]_Q, [\boldsymbol{x}']_R) := [R^{-1}]_Q \cdot ([\boldsymbol{x}]_Q - \texttt{Conv}_{R \to Q}([\boldsymbol{x}']_R))$

This methodology significantly bolsters computational efficacy, particularly by facilitating parallel processing, a vital attribute for managing extensive computations in large-scale contexts.

2.3 CKKS Scheme

In the realm of Fully Homomorphic Encryption (FHE) schemes, the Cheon-Kim-Kim-Song (CKKS) scheme [9] emerges as one of the most noteworthy, primarily owing to its support for floating-point and complex numbers. This compatibility aligns favorably with the needs of real-world applications. The CKKS is classified as a leveled homomorphic scheme, signifying its capability to evaluate computations with a pre-determined depth of multiplication, once the requisite parameters have been established. The scheme employs a moduli chain, denoted as $Q_L := \Pi_{i=0}^{L} q_i$, as the ciphertext modulus, while the switching to a smaller modulus, represented as Q', is utilized for managing noise. The plaintext space in the CKKS scheme is represented by R, with the ciphertext space being defined as $R_{Q_l} = \mathbb{Z}_{Q_l}[X] / \left(X^N + 1\right)$ at level l. In this representation, $Q_l := \Pi_{i=0}^{l} q_i$ and N symbolizes a power of 2. This structure facilitates noise management and complexity reduction, crucial aspects for maintaining computational efficiency in complex number manipulations.

Encoding. In the CKKS scheme, up to $N/2$ distinct messages can be encoded into individual slots of a single plaintext to facilitate batch processing, allowing for enhanced flexibility with input data types such as floating-point and complex numbers. To achieve message encoding and decoding, the scheme leverages two mapping functions. The initial one is the canonical embedding $\sigma : \boldsymbol{p} \mapsto (\boldsymbol{p}(\xi^j))_{j\in\mathbb{Z}_{2N}^*}$ which maps $\boldsymbol{p} \in \mathbb{R}[X]/(X^N+1)$ to $\mathbb{H} = \{(z_j)_{j\in\mathbb{Z}_{2N}^*} : z_{2N-j} = \bar{z_j}\} \subseteq \mathbb{C}^N$. The subsequent one is the natural projection $\pi : (z_j)_{j\in\mathbb{Z}_{2N}^*} \mapsto (z_j)_{j\in T}$, with T as a subgroup of $\mathbb{Z}_{2N}^*$. Consequently, decoding is accomplished through the sequential transformation $\mathbb{R}[X]/(X^N+1) \xrightarrow{\sigma} \mathbb{H} \xrightarrow{\pi} \mathbb{C}^{N/2}$, with encoding represented as the inverse of this process. To accommodate floating-point numbers, a scaling factor Δ is introduced to convert them to integers. With this architecture in place, the encoding and decoding functions in the CKKS scheme are realized as $\mathsf{Ecd}(\mathbf{z} \in \mathbb{C}^{N/2}; \Delta) : \mathbb{C}^{N/2} \to R, \mathbf{z} \mapsto \boldsymbol{m} = \lfloor \Delta \cdot \sigma^{-1}(\pi^{-1}(\mathbf{z}))\rceil$ and $\mathsf{Dcd}(\boldsymbol{m} \in R; \Delta) : R \to \mathbb{C}^{N/2}, \boldsymbol{m} \mapsto \mathbf{z} = \pi(\sigma(\Delta^{-1} \cdot \boldsymbol{m}))$, respectively. This structure endows the CKKS scheme with remarkable efficiency and versatility in handling varied data types.

Encryption and Decryption. The encryption and decryption process in CKKS is defined as follows. The secret key is composed of a random ring element $\boldsymbol{s} \in R$ sampled following the distribution $\mathcal{X}_k$, and is represented as $\mathbf{sk} := (1, \boldsymbol{s})$. The public key, denoted as $\mathbf{pk} := (\boldsymbol{b}, \boldsymbol{a})$, is formulated as $([-\boldsymbol{a}\cdot\boldsymbol{s}+\boldsymbol{e}]_{Q_L}, \boldsymbol{a}) \in R_Q^2$, where $\boldsymbol{a} \xleftarrow{\$} R_{Q_L}$ and $\boldsymbol{e} \leftarrow \mathcal{X}_e$. For the encryption of a plaintext $\boldsymbol{m}$, the resulting ciphertext $\mathsf{ct} := (\boldsymbol{c}_0, \boldsymbol{c}_1)$ is computed as $\mathsf{Enc}_{\mathbf{pk}}(\boldsymbol{m}) := [\boldsymbol{r} \cdot (\boldsymbol{b}, \boldsymbol{a}) + (\boldsymbol{e}_0 + \boldsymbol{m}, \boldsymbol{e}_1)]_{Q_L} \in R_{Q_L}$, where $\boldsymbol{r} \leftarrow \mathcal{X}_k$ and $\boldsymbol{e}_0, \boldsymbol{e}_1 \leftarrow \mathcal{X}_e$. Conversely, to decrypt the ciphertext $\mathsf{ct} = (\boldsymbol{c}_0, \boldsymbol{c}_1) \in R_{Q_l}^2$ at level l, the decryption result is computed as $\boldsymbol{m}' := [\boldsymbol{c}_0 + \boldsymbol{c}_1 \cdot \boldsymbol{s}]_{Q_l}$.

Homomorphic Evaluation. In the context of the CKKS, homomorphic evaluation is achieved through various operations. Taking ring elements $\boldsymbol{x}$, $\boldsymbol{y}$ as two plaintexts, with $\boldsymbol{x}$ encoding a vector $\mathbf{x}$, the following properties are provided:

- Addition ($\oplus$): $\mathsf{Dec}(\llbracket \boldsymbol{x} \rrbracket \oplus \llbracket \boldsymbol{y} \rrbracket) = \boldsymbol{x} + \boldsymbol{y}, \mathsf{Dec}(\llbracket \boldsymbol{x} \rrbracket \oplus \boldsymbol{y}) = \boldsymbol{x} + \boldsymbol{y}$.
- Multiplication ($\otimes$): $\mathsf{Dec}(\llbracket \boldsymbol{x} \rrbracket \otimes \llbracket \boldsymbol{y} \rrbracket) = \boldsymbol{x} \cdot \boldsymbol{y}, \mathsf{Dec}(\llbracket \boldsymbol{x} \rrbracket \otimes \boldsymbol{y}) = \boldsymbol{x} \cdot \boldsymbol{y}$.
- Rotation ($\mathsf{HRot}_i(\cdot)$): $\mathsf{Dec}(\mathsf{HRot}_i(\llbracket \boldsymbol{x} \rrbracket)) = \mathsf{Ecd}(\mathbf{x} \ll i)$.

2.4 HE-enabled Neural Network

In the architecture of an HCNN, neurons are meticulously organized into several layers. These layers typically contain two categories based on their function: linear and nonlinear layers.

- **Linear Layers.** The principal components of linear layers are convolutional layers and fully connected layers. These layers primarily perform linear weighted-sum operations denoted as $\llbracket \boldsymbol{y} \rrbracket = \boldsymbol{w} \otimes \llbracket \boldsymbol{x} \rrbracket \oplus \boldsymbol{b}$, where $\boldsymbol{w}$ represents

the weight matrix and $\boldsymbol{b}$ the bias vector post-encoding. The data input is encoded as $\boldsymbol{x}$ and the output is represented by $\boldsymbol{y}$. The convolutional layer only selects a subset of the output from the previous layer as input to each neuron through the use of a filter. Contrarily, the fully connected layer receives inputs from all neurons in the previous layer.

– **Non-Linear Layers.** The activation layers, which generally perform nonlinear functions, fall under this category. These layers are instrumental in preventing overfitting in the model. However, these nonlinear functions often necessitate polynomial approximations for homomorphic evaluations. The Rectified Linear Unit (ReLU) is a commonly used activation function in this context, performing an element-wise operation defined as $y = \max(0, x)$.

The image pixel values are typically normalized to a floating-point number within the interval $[0, 1]$, which is compatible with the input domain of CKKS. Furthermore, the entirety of the vector can be encoded into a single plaintext, thereby allowing batch processing, a technique that enhances computational efficiency significantly. This integration of the data input with the encryption scheme proves advantageous for the utilization of HE in neural networks.

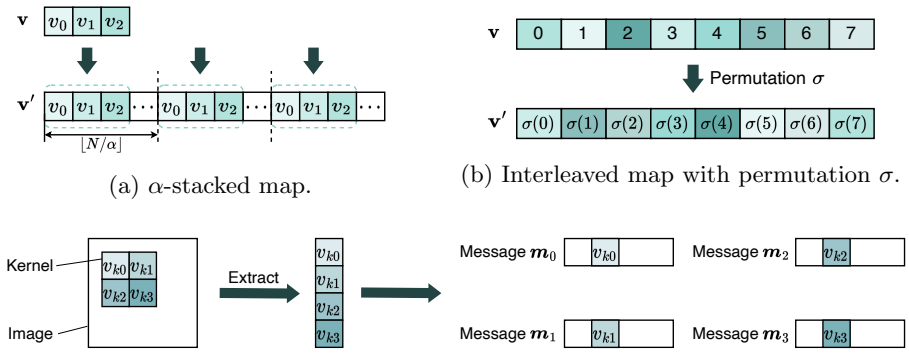
3 XNET: Improved HCNN

In this section, we provide a comprehensive overview of our proposed network, the XNET, highlighting its distinctive features and improvements over the conventional HCNN. We start by introducing the unified network structure of XNET, which streamlines data processing and promotes operational efficiency. Subsequently, we delve into the specifics of spatial operations adaptation, detail the activation layer approximation, and elucidate the implementation of highly batched dense layer computations, which play a crucial role in enhancing computational performance.

3.1 Vertorized Message Maps in XNET

Below we introduce our approach that focuses on the efficient handling of image data in privacy-preserving inference. Our proposition entails several mappings for vectorized data representations, consequently obtaining messages primed for encoding and encryption or changing the context of ciphertexts. These mappings are meticulously designed to enhance computational efficiency during subsequent model evaluation.

Convolution Map. The convolution map plays an integral role in bridging the gap between an original image and a specialized representation, facilitating computations conducive to HE operations. The central objective of this map lies in the reconstruction of the image in such a way that the pixels in the same position of each receptive field can be computed in an SIMD approach. In the convolution map process, the image is first decomposed in accordance with a given convolution filter, allowing for the extraction of each receptive field. The pixels that



(a) α-stacked map.

(b) Interleaved map with permutation σ.

(c) Convolution map with a kernel of 4 weights to 4 messages.

Fig. 2: Descriptions of the defined vectorized message maps in XNET.

occupy the same position within these receptive fields are then transferred to identical messages. This means that the j-th pixel of the i-th receptive field is positioned in the i-th slot of the j-th message. For an image of size $W \times H$ and a filter of size $F \times F$ with a stride of S, the total number of messages represented equates to the size of the filter. Each message length corresponds to the receptive fields coverage $((W-F)/S+1) \times ((H-F)/S+1)$. Through this mapping technique, efficient and streamlined computations can be performed on convolutional layers in a manner that is amenable to HE operations.

Stacked Map. The stacked map mechanism involves the creation of a message encompassing several duplicates of a short vector $\mathbf{v}$, which are uniformly distributed within the message. The fundamental purpose of this mapping technique is to harness the substantial volume of slots to enhance the parallelism intrinsic to batch processing. This strategy contributes to reducing the count of homomorphic operations needed. In a standard scenario, an α-stacked map applied to a message of length n results in a message containing α copies of $\mathbf{v}$. This procedure is depicted by the formula $v_i \mapsto (v'_i, v'_{i+\lfloor n/\alpha \rfloor}, v'_{i+2\lfloor n/\alpha \rfloor}, \ldots, v'_{i+(\alpha-1)\lfloor n/\alpha \rfloor})$. Through this mapping, each vector is systematically replicated and placed throughout the message, maximizing slot usage and promoting a high degree of parallelism, consequently reducing the computational overhead of HE operations.

Interleaved Map. The interleaved map plays a pivotal role in reordering the sequence of a given vector based on a provided permutation σ. This results in the generation of a vector that adheres to this newly established sequence. For a vector $\mathbf{v}$, where v_i denotes the i-th element, the interleaved map method facilitates the creation of a new vector $\mathbf{v}'$, where i-th element v'_i is essentially $v_{\sigma(i)}$, i.e., the $\sigma(i)$-th element of the initial vector. A special case worth noting arises when the permutation σ is an identity permutation. In such a scenario, the map exhibits a direct one-to-one correspondence between the elements of

Table 1: Illustrative breakdown of our proposed HCNN architecture: procedural stages, input/output dimensions, and corresponding operations.

Procedure		Input size	Output size	Descryption
Pre-process	Grayscaling	$32 \times 32 \times 3$	$32 \times 32 \times 1$	Transform the 3-color channel input to a grayscale representation
	Batching	$32 \times 32 \times 1$	1024×9	Reorganize the grayscale image through the convolutional mapping
Inference	Convolution	1024×9	1024×8	Pad to 34×34 and apply 8 filters of 3×3 with stride $(1, 1)$ for feature extraction
	Activation	1024×8	1024×8	Polynomial approximation to ReLu for activation mapping
	Pooling	1024×8	64×8	Conduct dimensionality reduction using an extent of 4 and stride 4
	Dense	512×1	128×1	Apply a linear operation with 128 filters to integrate global features
	Activation	128×1	128×1	Polynomial approximation to ReLu for activation mapping
	Dense	128×1	10×1	Execute a final linear operation with 10 filters for optimized feature extraction
	Output	10×1	10×1	Final classification vector representing the output labels

the vector and the result, effectively denoted as $v_i \mapsto v'_i$. The Interleaved Map, therefore, serves as a powerful tool in data manipulation, enabling flexibility in structuring the sequence based on desired permutations.

3.2 Unified Network Structure

Practical considerations underline the merit of maintaining a single unified model, as it presents a more resource-efficient solution compared to training and managing a plethora of specialized models. With this insight, we propose a compact and unified HCNN architecture, as presented in Table 1. We elucidate the methodology integral to the proposed design, which streamlines data analysis by transforming and simplifying inputs, adeptly managing features, and classifying operations to accommodate the unique needs of homomorphic evaluations. This consolidated approach brings significant practical benefits, especially in terms of resource efficiency.

Pre-Processing. In the methodology put forth, a pre-processing phase is incorporated, designed to consolidate the three color channels of a 3-dimensional image into a singular grayscale channel. This step effectively transmutes the image dimensions from $32 \times 32 \times 3$ to $32 \times 32 \times 1$. This transformation is achieved by computing the average of the pixel values spanning across the Red, Green, and Blue color channels for each pixel location, thereby converting the color image into a grayscale equivalent. The fundamental motivation behind this preprocessing measure is to streamline the ensuing processing stages, consequently

reducing computational complexity without significantly undermining the critical image features requisite for detailed analysis. It should be noted that this step capitalizes on the fact that human visual perception predominantly depends on luminance, a feature aptly encapsulated in grayscale images. Through this pre-processing phase, the original image can be represented in a more compact form, thereby mitigating both data size and computational complexity.

Private Inference. The inference phase of the proposed architecture is characterized by multiple layer types, each playing a distinctive role in the extraction and refinement of features from the given input data. The initiation of this phase involves the execution of a convolutional operation that applies multiple filters to the batched inputs, thereby resulting in a set of ciphertexts signifying the primary extracted features. Specifically, an assortment of eight filters, each of size 3×3, is employed to systematically scrutinize the batched inputs. The filter weights and bias are encoded into disparate plaintexts through the stacked mapping, and subsequently, these are multiplied or added respectively to the encrypted pixel vectors, leading to a transformation from an input dimensionality of 1024×9 to an output of 1024×8. It should be noted that the images undergo a process of zero-padding, resulting in a dimensionality expansion to 34×34. The comprehensive coverage paired with the filters' relatively small size facilitates an extensive feature extraction from the input, enabling detection of small-scale patterns.

Following the convolutional layer, an activation function is applied, thus enabling the model to discern more complex relationships within the data. The output from the activation function is then subjected to an average pooling operation. This operation is performed with a size of 4 and stride of 4, thus reducing the spatial dimensions of the feature maps to 64×8. The pooling operation holds twofold benefits, it mitigates overfitting by offering an abstracted representation, and decreases computational cost by reducing the dimensionality of the representation.

Finally, the condensed feature maps are then passed through a series of dense layers. Each of these dense layers performs a linear operation on its input, amalgamating the features within a global context. The dense layers, equipped with 128 and 10 filters respectively, continuously refine the representations, thereby reducing their dimensionality whilst preserving the complexity of the features. As a result, the model achieves an efficient and robust feature analysis during private inference.

Classification of Operations. Within the context of our proposed network architecture, we classify the operations conducted across the previously discussed layers into two primary categories based on the structure of the operation:

– Spatial Operations. This category encompasses both convolutional and pooling operations. These operations intrinsically consider spatial relationships among pixels in an image by employing two-dimensional kernels that slide

across the width and height of the image. These layers retain the spatial structure of the image and enable the network to learn and comprehend spatial hierarchies.
– Non-Spatial Operations. This category refers to the dense and activation layers where the input data is regarded as a flat array rather than a two-dimension image. In a dense layer, for instance, every neuron is interconnected with every neuron in the previous layer, regardless of their spatial relationships in the original image. Similarly, activation functions apply a non-linear transformation to each input independently.

The core objective of this classification is to address the unique processing requirements associated with homomorphic evaluation. Homomorphic encryption schemes present distinct operational complexities that vary depending on operation structure. By distinguishing these types, we can tailor the application of homomorphic operations to better fit the computation of each layer, thereby optimizing performance.

3.3 Adaptations of Spatial Operations

Spatial operations, in the context of our proposed HCNN architecture, primarily involve the convolutional layer and the average pooling layer. These layers function in a sequential flow of computation that begins with a convolutional operation extracting receptive fields, followed by the application of an activation function, and finally, the average pooling of the resultant output.

In this process, it is worth noting that the computations involving multiplications with weights and additions with bias, which can be computed in parallel for each pixel. We achieved this by initially packing the pixels to allow batch processing, thus optimizing computational efficiency. This technique, designed to finely adapt the HE operations, is a key component in our architecture.

In detail, we initially apply the convolutional map to obtain F^2 messages. These messages are then encoded and encrypted, resulting in ciphertexts that effectively hide the original data. Concurrently, we apply the stacked map to encode each weight in the convolutional filters into different plaintexts. For each channel, the ciphertexts are then multiplied with the corresponding plaintexts. The results of these multiplications across all channels are then aggregated, yielding the convolutional results for each individual channel. Given that our architecture accommodates 8 channels, this step produces 8 distinct ciphertexts. Subsequent to these operations, we apply the activation function to the processed data. The final step in this sequence is the computation of average pooling. This operation serves to abstract and reduce the spatial dimensions of the processed data, consequently resulting in a more manageable and efficient representation for subsequent layers.

Interleaved Batching Technique. In order to optimize the processing of the pooling component, we introduce an interleaved convolutional map in our proposed HCNN architecture. Our design deviates from the conventional approach

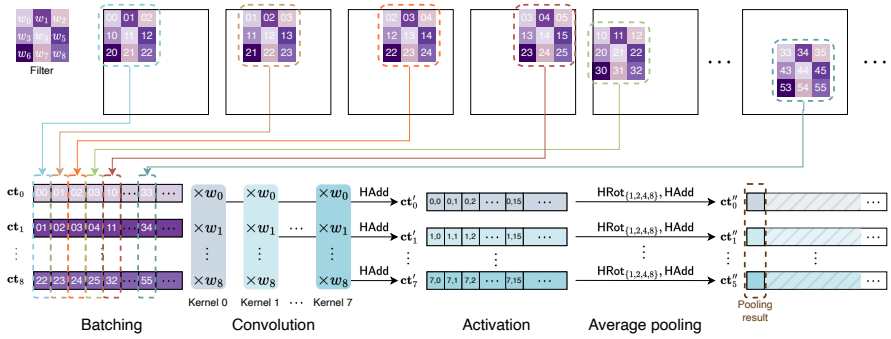


Fig. 3: A schematic representation of the interleaved convolutional mapping using 8 filters of size 3×3 in our proposed HCNN architecture, and the computational flow from convolution to average pooling.

of sequentially storing each receptive field; instead, it stores the set of receptive fields processed by the same pooling operation in a sequential manner. This innovative approach enables the direct execution of pooling operations following batched convolutional operations.

The interleaved convolutional map is essentially a fusion of convolutional map and interleaved map techniques. To illustrate this, consider an image of size $W \times H$ being processed by a filter of size $F \times F$ with a stride of S. Initially, all receptive fields of size F^2 are extracted and mapped to F^2 messages. This process involves a permutation that affects the position of each receptive field but leaves the convolutional operation unaffected, given that computations are batched and executed in an SIMD manner. Following this, we perform four homomorphic rotations and additions to compute the sum value, in alignment with the pooling layer's function of averaging 16 values. Fig. 3 presents a visual representation of our approach, exemplifying the use of a filter of size 3×3. This innovative technique of interleaved batching effectively adapts the processing of the pooling part, enhancing computational efficiency and data flow in our proposed architecture.

3.4 Effective Approximation of Activation Layer

In our proposed HCNN architecture, we apply the ReLu as the activation function, defined as $\mathsf{ReLu}(x) = \max(0, x)$. As a popular option in contemporary network designs, the ReLU function is distinguished by its property of zeroing out negative values while maintaining linearity for positive inputs. However, due to its non-linearity, its application within the context of the CKKS homomorphic encryption scheme requires an approximation technique, most commonly the Taylor series or Chebyshev polynomials. These higher-degree polynomial approximations, while effective within a certain range of x values, may deviate significantly from the ReLU function for larger or smaller x values. Bearing this

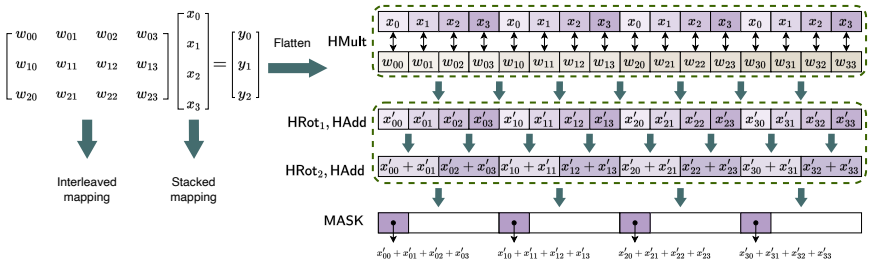


Fig. 4: Example of the parallelized dense layer computation in HCNN with dimension $k_1 = 4$ and $k_2 = 3$.

in mind, our approach leverages a simpler quadratic approximation, representing the function as $\mathsf{ReLU}(x) \approx x^2$. The simplicity and universal continuity and differentiability of this function assures computational efficiency.

Admittedly, the quadratic approximation lacks the sparsity property of ReLU, which yields efficiency by driving many of the neurons to output zero. However, given that the operands are ciphertexts in homomorphic operations, the plaintext value will not influence the computation pattern. Therefore, the lack of sparsity does not significantly impact the computational performance. Consequently, for these reasons, we employ the quadratic approximation for the ReLU function, enabling the activation operation to be conducted with a single homomorphic multiplication.

3.5 Efficient Batched Computation in Dense Layer

In the dense layer, every neuron exhibits a connection to each neuron from the preceding layer, and each linkage carries an associated weight. The output of a neuron is calculated by summing the products of these weights with their corresponding inputs, adding a bias term, and passing the resultant sum through an activation function. Mathematically, let us denote $\mathbf{x}$ as an input vector of dimension k_1, and $\mathbf{W}$ as a weight matrix of dimension $k_1 \times k_2$. Here, k_1 represents the number of neurons in the previous layer, and k_2 denotes the number of neurons in the current dense layer. Furthermore, let $\mathbf{b}$ be the bias vector of dimension k_2. Consequently, the output vector $\mathbf{y}$ of the dense layer can be computed as: $\mathbf{y} := \mathbf{W} \cdot \mathbf{x} + \mathbf{b}$. As the computation for each neuron can be performed independently, a form of parallel computation is naturally facilitated. In fig. 4, we provide an illustrative example of this process, with $k_1 = 4$ and $k_2 = 3$.

For the dense layer of our proposed network, we achieve highly batched computation and introduce two optimizations. These are specifically designed to significantly reduce computational and memory overhead, thereby enhancing overall performance. These methods are detailed below.

Low-Complexity Transition. In our proposed network, before average pooling, we possess eight ciphertexts, with each storing 64 pooling results across

1024 slots at intervals of 16. The traditional method for transitioning from the pooling layer to the dense layer involves homomorphically rotating the i-th ciphertext by $1024i$ steps and summing all ciphertexts. This results in a ciphertext that accommodates 512 results within 8192 slots, paving the way for the batch processing computation of the dense layer.

However, we deviate from this standard approach and propose a method with lower complexity. In the initial stage of the pre-processing phase, we employ a stacked map. Specifically, in contrast to the traditional batching approach, we store eight copies in each ciphertext, and encode the weights in the same position across the eight channels into a single plaintext. This strategy reduces the original 64 plaintexts to just eight.

We then perform eight homomorphic multiplications on the ciphertext and sum all the results. Consequently, we are left with a single ciphertext containing eight stacked results. This optimization is beneficial as it not only reduces the homomorphic multiplication with weight plaintexts and the homomorphic addition with bias plaintexts, but also obviates the need for homomorphic rotation to achieve the stacked map, thereby simplifying the computational process.

Tight Packing. After the average pooling stage, the results are stored at intervals of 16, which leads to an inefficient utilization of available slots and underutilizes the potential for parallel computation. To address this inefficiency, we introduce a tight packing technique. This technique involves iterative application of homomorphic rotation and addition on the ciphertext with steps defined in $1, 2, 4, 8$. Following this procedure, each of the previously empty 15 slots now stores a copy of the pooling result. Subsequently, the positions of the weights of the dense layer are adjusted to correspond to this sequence, and the then encoded to produce the plaintexts. By applying this method, we achieve a significant increase in parallelism, improving it by a factor of 16. In the architecture of our proposed network, the first dense layer stores 512×16 weights in each weight plaintext, resulting in a total of 8 weight plaintexts.

This highly parallel structure provides substantial benefits, markedly enhancing processing speed and overall performance. This enhancement is particularly significant when handling complex network operations in the dense layer, thereby improving the efficiency of our proposed network architecture.

4 Implementation and Acceleration

4.1 Instantiation of Data Sets

We deploy the proposed network on three datasets: MNIST [16], Fashion-MNIST [19], and CIFAR-10 [15]. MNIST and Fashion-MNIST both include 28×28 grayscale images distributed over 10 categories. The CIFAR-10 dataset features 32×32 color images, categorized into 10 classes. To accommodate these datasets into our network, we pad all images to 34×34 dimensions. Given our use of a

Table 2: Breakdown of resource consumption across different layers in the proposed network.

		Convolution	Activation	Pooling	Dense	Activation	Dense
Input	Ciphertext	9	1	1	1	1	1
	Plaintext	10	0	1	10	0	2
Output	Ciphertext	1	1	1	1	1	1
HE operation	HAdd	8	0	4	83	0	16
	CAdd	1	0	0	1	0	1
	HMult	0	1	0	0	1	0
	CMult	9	0	1	16	0	1
	HRot	0	0	4	83	0	16
Consumed depth		1	1	1	2	1	1

unified network, the processing steps remain consistent across all datasets. Table 2 presents a detailed account of resource consumption at each stage of our network, showcasing how our approach effectively caters to the distinct characteristics of each dataset. This marks a significant stride towards efficient resource utilization within the scope of homomorphically encrypted operations.

4.2 Scheme Configuration

The total multiplication depth is 7 in our network, as a consequence, the modulus chain is instantiated with $|Q_L| = 340$ and $L = 7$. For our requirements, this setting provides a substantial security level of 128-bit. To streamline computations, we keep ciphertexts in a double-CRT representation, where each residue under the RNS representation is in the Number Theoretic Transformation (NTT) domain. We particularly set the dimension $N = 2^{14}$ to meet our computational needs. The key-switching, which is an integral part of our implementation, is performed with a special modulus $|P| = 60$. This setting allows for a larger ciphertext decompose number, while simultaneously enabling a reduction in other HE parameters, such as the ring dimension N. Notably, this provides a balance between complexity and parameter size. It upholds the same security level, thereby making it a more efficient choice than the full RNS variant [13], particularly for circuits where the evaluation depth is not excessively deep.

4.3 Hierarchical Decomposition and Acceleration of Layers

To efficiently compute the HCNN layers, we closely examine the underlying HE evaluation operations constituting each layer, as well as their hierarchical structure. Leveraging our custom-developed GPU acceleration engine, we optimize these operations for top-tier performance. Importantly, RNS level operations, which form the fundamental components of these HE operations, are particularly amenable to batch processing on a GPU, given that the residues can be processed in parallel. Fig. 5 presents our approach.

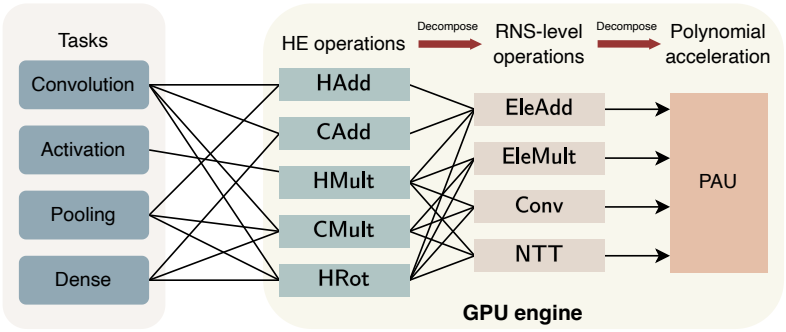


Fig. 5: Illustration of the hierarchical construction of HCNN layers, corresponding HE evaluation operations, and our custom-built GPU acceleration engine for efficient execution.

The foundation of our GPU acceleration engine is the Polynomial Acceleration Unit (PAU). The primary function of the PAU is to expedite polynomial arithmetic operations under the RNS representation by batching residues. In the context of the RNS-level operations facilitated by the PAU:

- EleAdd: performs modular addition, indicating an element-wise addition operation on the polynomials. This operation underscores the simple yet crucial aspect of polynomial arithmeticaddition at the element level.
- EleMult: represents the Hadamard (element-wise) product of two polynomials, manifesting an integral part of the PAU's capabilities in executing complex polynomial operations.
- Conv: signifies the base conversion operation, underscoring the PAU's ability to facilitate flexible arithmetic operations in different numerical bases.
- NTT: corresponds to the Number Theoretic Transform, a critical component in the polynomial arithmetic domain for transforming polynomials between different domain to accelerate polynomial multiplications.

Based on this unit, we develop the homomorphic evaluation operations, and the pseudocode is present in Algorithm 1. Each of these HE operations can be reconstructed by the more elemental RNS level operations. Specifically, the operations HAdd and CAdd consist of EleAdd. On the other hand, the operations ciphertext-ciphertex multiplication HMult, plaintext-ciphertex multiplication CMult, and rotation HRot are more complex, containing operations EleMult, EleAdd, Conv, and NTT, respectively.

By utilizing this hierarchical decomposition approach, we can streamline the computation process in HCNNs, making the execution of complex operations more manageable and efficient. Furthermore, our GPU acceleration engine ensures that these operations are performed at an expedited rate, thus enhancing the overall performance of our HCNN implementation.

Algorithm 1 Implemented Homomorphic Evaluation Operations

function $\mathsf{HAdd}(\mathbf{ct}, \mathbf{ct}')$
  $\mathbf{ct} := (\boldsymbol{c}_0, \boldsymbol{c}_1)$, $\mathbf{ct}' := (\boldsymbol{c}'_0, \boldsymbol{c}'_1)$
  $\boldsymbol{d}_0 := \boldsymbol{c}_0 + \boldsymbol{c}'_0$, $\boldsymbol{d}_1 := \boldsymbol{c}_1 + \boldsymbol{c}'_1$
  return $\mathbf{ct}_A := (\boldsymbol{d}_0, \boldsymbol{d}_1)$
function $\mathsf{HMult}(\mathbf{ct}_0, \mathbf{ct}_1, \mathbf{swk})$
  $\mathbf{ct} := (\boldsymbol{c}_0, \boldsymbol{c}_1)$, $\mathbf{ct}' := (\boldsymbol{c}'_0, \boldsymbol{c}'_1)$
  $\boldsymbol{d}_0 := \boldsymbol{c}_0 \cdot \boldsymbol{c}'_0$, $\boldsymbol{d}_2 := \boldsymbol{c}_1 \cdot \boldsymbol{c}'_1$, $\boldsymbol{d}_1 := \boldsymbol{c}_1 \cdot \boldsymbol{c}'_0 + \boldsymbol{c}_0 \cdot \boldsymbol{c}'_1$
  $(\boldsymbol{d}'_0, \boldsymbol{d}'_1) := \mathsf{KeySwitch}(\boldsymbol{d}_0, \mathbf{swk})$
  return $\mathbf{ct}_M := (\boldsymbol{d}_0 + \boldsymbol{d}'_0, \boldsymbol{d}_1 + \boldsymbol{d}'_1)$
function $\mathsf{HRot}(\mathbf{ct}, sn, \mathbf{swk})$
  $\mathbf{ct} := (\boldsymbol{c}_0, \boldsymbol{c}_1)$
  $\boldsymbol{d}_0 := \mathsf{FrobeniusMap}(\boldsymbol{c}_0, sn)$, $\boldsymbol{d}_1 := \mathsf{FrobeniusMap}(\boldsymbol{c}_1, sn)$
  $(\boldsymbol{d}'_0, \boldsymbol{d}'_1) := \mathsf{KeySwitch}(\boldsymbol{d}_0, \mathbf{swk})$
  return $\mathbf{ct}_R := (\boldsymbol{d}'_0, \boldsymbol{d}_1 + \boldsymbol{d}'_1)$
function $\mathsf{Rescale}(\mathbf{ct})$
  $\mathbf{ct} := (\boldsymbol{c}_0, \boldsymbol{c}_1) \in R_{Q_l}$, $j \in [0, l)$
  $\boldsymbol{d}_0 := [q_l^{-1} \cdot (\boldsymbol{c}_0^{(j)} - \boldsymbol{c}_0^{(l)})]_{q_l}$, $\boldsymbol{d}_1 := [q_l^{-1} \cdot (\boldsymbol{c}_1^{(j)} - \boldsymbol{c}_1^{(l)})]_{q_l}$
  return $\mathbf{ct}' := (\boldsymbol{d}_0, \boldsymbol{d}_1)$
function $\mathsf{KeySwitch}(\mathbf{ct}, \mathbf{swk})$
  $\mathbf{ct} := (\boldsymbol{c}_0, \boldsymbol{c}_1) \in R_{Q_l}, \mathbf{swk} := \{(\boldsymbol{b}_i, \boldsymbol{a}_i)\}_{i \in [0,l]}$
  $(\boldsymbol{c}'_0, \boldsymbol{c}'_1) := (0, 0)$
  for $i \in [0, l]$ **do**
    $\boldsymbol{c}^* := \mathsf{ModUp}_{q_i \to Q_l P}(\boldsymbol{c}_1^{(i)})$
    $(\boldsymbol{c}_0^*, \boldsymbol{c}_1^*) := (\langle \boldsymbol{c}^*, \boldsymbol{b}_i \rangle, \langle \boldsymbol{c}^*, \boldsymbol{a}_i \rangle)$
    $(\boldsymbol{c}'_0, \boldsymbol{c}'_1) := (\boldsymbol{c}'_0, \boldsymbol{c}'_1) + (\boldsymbol{c}_0^*, \boldsymbol{c}_1^*)$
  $(\boldsymbol{c}''_0, \boldsymbol{c}''_1) := (\mathsf{ModDown}_{Q_l P \to Q_l}(\boldsymbol{c}'_0), \mathsf{ModDown}_{Q_l P \to Q_l}(\boldsymbol{c}'_1))$
  return $\mathbf{ct}'' = (\boldsymbol{c}''_0, \boldsymbol{c}''_1)$

5 Results and Comparison

5.1 Hardware Specifications

Our experiments are performed on an Arch Linux system with kernel 5.15. The C/C++ code is compiled using g++ 12.2.0, and GPU tasks are handled using CUDA 11.8. For the CPU baseline, we employ an Intel(R) Core(TM) i9-12900KS CPU with 16 cores. We deploy and test our GPU implementation on a NVIDIA Tesla A100 80G PCIe. This diverse configuration allows us to assess the robustness across a variety of hardware environments, yielding a well-rounded understanding of the performance of our implementation.

5.2 Performance and Comparisons

Table 3 delivers an in-depth performance analysis of our network on both CPU and GPU hardware platforms. The CPU processing times demonstrate that the first dense layer requires the most computational resources, taking up to 369.769

Table 3: Detailed breakdown of the execution time for each layer in our network, measured in milliseconds.

Device	Convolution	Activation	Pooling	Dense	Activation	Dense
CPU	26.145	12.284	33.8	369.769	3.943	22.745
GPU	1.491	0.044	0.148	29.751	0.036	0.428
Speedup	17.5×	279.2×	228.4×	12.4×	109.5×	53.1×

milliseconds. This substantial duration can be attributed to the significant number of homomorphic multiplication and rotation operations required in this stage. Meanwhile, the pooling and convolution layers follow in terms of time consumption. On the A100 GPU, the dense layer still necessitates the majority of the processing time, albeit at a much reduced duration of 29.751 milliseconds. The convolution, pooling, and activation layers, on the other hand, are handled quite efficiently, each requiring less than 2 milliseconds. Our GPU implementation facilitates speedups of up to 17.5× for convolution, 228.4× for pooling, 12.4× for the dense layer, and 109.5× for activation.

Table 4: Comparison of execution time and accuracy between our network and prior works on MNIST [16], Fashion-MNIST [19], and CIFAR-10 datasets [15]. Times are measured in seconds.

Method	Dataset	Accuracy (%)	Latency (s)	Platform
CryptoNets [11]	MNIST	98.95	205	CPU
FCryptoNets [10]	MNIST	98.71	39.1	CPU
LoLa [8]	MNIST	98.95	2.2	CPU
Falcon [17]	MNIST	98.95	1.2	CPU
LoLa [8]	CIFAR-10	76.5	730	CPU
Falcon [17]	CIFAR-10	76.5	107	CPU
XNET	MNIST	99.14	0.46	CPU
	MNIST	99.14	0.032	GPU
	Fashion-MNIST	90.8	0.46	CPU
	Fashion-MNIST	90.8	0.032	GPU
	CIFAR-10	61.09	0.46s	CPU
	CIFAR-10	61.09	0.032	GPU

Table 4 provides a performance comparison of our proposed method with other established techniques, evaluated on NIST [16], Fashion-MNIST [19], and CIFAR-10 datasets [15]. Across all datasets, our method not only achieves competitive accuracy, but also significantly reduces the latency compared to other works. In the context of MNIST, our approach attains the highest accuracy of 99.14%, outperforming CryptoNets, FCryptoNets, LoLa, and Falcon. More impressively, it reduces the execution time from 205 seconds in CryptoNets to 0.46 seconds on CPU and a mere 0.032 seconds on GPU. When it comes to the more complex CIFAR-10 and Fashion-MNIST datasets, our solution still

maintains superior performance in terms of latency, staying consistent at 0.46 seconds on CPU and 0.032 seconds on GPU, despite a slight drop in accuracy. This demonstrates that our method successfully provides a real-time solution for secure neural network inference across various datasets and platforms, breaking through the limitations of existing works.

6 Conclusion

In this study, we present an optimized HE-based convolutional neural network inference framework for real-time applications. This is achieved through a unified and compact network structure, and several innovative optimization strategies, including a unique compression and encoding technique, and efficient rearrangement of the encoding sequence of pixels. We further exploit GPU's thread-level parallelism for accelerated computation. Experimental results on three datasets show remarkable inference accuracies and a notable processing speed improvement, with a 3000× speedup compared to prior works. Our findings contributes to future research in secure and real-time machine learning applications, inspiring further advancements in network structure design, optimization techniques, and hardware utilization.

References

1. Al Badawi, A., Jin, C., Lin, J., Mun, C.F., Jie, S.J., Tan, B.H.M., Nan, X., Aung, K.M.M., Chandrasekhar, V.R.: Towards the alexnet moment for homomorphic encryption: Hcnn, the first homomorphic cnn on encrypted data with gpus. IEEE Transactions on Emerging Topics in Computing **9**(3), 1330–1343 (2021). https://doi.org/10.1109/tetc.2020.3014636
2. Bajard, J., Eynard, J., Hasan, M.A., Zucca, V.: A full rns variant of fv like somewhat homomorphic encryption schemes. In: Selected Areas in Cryptography - SAC 2016. pp. 423–442. Lecture Notes in Computer Science (2017). https://doi.org/10.1007/978-3-319-69453-5_23
3. Bian, S., Wang, T., Hiromoto, M., Shi, Y., Sato, T.: ENSEI: efficient secure inference via frequency-domain homomorphic convolution for privacy-preserving visual recognition. In: 2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition, CVPR 2020, Seattle, WA, USA, June 13-19, 2020. pp. 9400–9409. Computer Vision Foundation / IEEE (2020). https://doi.org/10.1109/CVPR42600.2020.00942
4. Boemer, F., Costache, A., Cammarota, R., Wierzynski, C.: ngraph-he2: A high-throughput framework for neural network inference on encrypted data. In: Brenner, M., Lepoint, T., Rohloff, K. (eds.) Proceedings of the 7th ACM Workshop on Encrypted Computing & Applied Homomorphic Cryptography, WAHC@CCS 2019, London, UK, November 11-15, 2019. pp. 45–56. ACM (2019). https://doi.org/10.1145/3338469.3358944, `https://doi.org/10.1145/3338469.3358944`
5. Boemer, F., Lao, Y., Cammarota, R., Wierzynski, C.: ngraph-he: a graph compiler for deep learning on homomorphically encrypted data. In: Palumbo, F., Becchi, M., Schulz, M., Sato, K. (eds.) Proceedings of the 16th ACM International Conference

on Computing Frontiers, CF 2019, Alghero, Italy, April 30 - May 2, 2019. pp. 3–13. ACM (2019). https://doi.org/10.1145/3310273.3323047, https://doi.org/10.1145/3310273.3323047

6. Bos, J.W., Lauter, K.E., Loftus, J., Naehrig, M.: Improved security for a ring-based fully homomorphic encryption scheme. In: Stam, M. (ed.) Cryptography and Coding - 14th IMA International Conference, IMACC 2013, Oxford, UK, December 17-19, 2013. Proceedings. Lecture Notes in Computer Science, vol. 8308, pp. 45–64. Springer (2013). https://doi.org/10.1007/978-3-642-45239-0_4, https://doi.org/10.1007/978-3-642-45239-0_4
7. Brakerski, Z.: Fully homomorphic encryption without modulus switching from classical gapsvp. In: Advances in Cryptology - CRYPTO 2012. Lecture Notes in Computer Science, vol. 7417, pp. 868–886 (2012). https://doi.org/10.1007/978-3-642-32009-5_50
8. Brutzkus, A., Gilad-Bachrach, R., Elisha, O.: Low latency privacy preserving inference. pp. 812–821 (2019)
9. Cheon, J.H., Kim, A., Kim, M., Song, Y.: Homomorphic encryption for arithmetic of approximate numbers. In: Advances in Cryptology - ASIACRYPT 2017. Lecture Notes in Computer Science, vol. 10624, pp. 409–437 (2017). https://doi.org/10.1007/978-3-319-70694-8_15
10. Chou, E., Beal, J., Levy, D., Yeung, S., Haque, A., Fei-Fei, L.: Faster cryptonets: Leveraging sparsity for real-world encrypted inference. CoRR **abs/1811.09953** (2018), http://arxiv.org/abs/1811.09953
11. Dowlin, N., Gilad-Bachrach, R., Laine, K., Lauter, K., Naehrig, M., Wernsing, J.: Cryptonets: Applying neural networks to encrypted data with high throughput and accuracy. In: Proceedings of the 33nd International Conference on Machine Learning, ICML 2016. JMLR Workshop and Conference Proceedings, vol. 48, pp. 201–210 (2016). https://doi.org/10.5555/3045390.3045413
12. Fan, J., Vercauteren, F.: Somewhat practical fully homomorphic encryption. IACR Cryptol. ePrint Arch. p. 144 (2012)
13. Han, K., Ki, D.: Better bootstrapping for approximate homomorphic encryption. In: Topics in Cryptology - CT-RSA 2020. pp. 364–390. Lecture Notes in Computer Science (2020). https://doi.org/10.1007/978-3-030-40186-3_16
14. Juvekar, C., Vaikuntanathan, V., Chandrakasan, A.P.: GAZELLE: A low latency framework for secure neural network inference. In: Enck, W., Felt, A.P. (eds.) 27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018. pp. 1651–1669. USENIX Association (2018)
15. Krizhevsky, A., Hinton, G.: Learning multiple layers of features from tiny images (2009)
16. LeCun, Y., Bottou, L., Bengio, Y., Haffner, P.: Gradient-based learning applied to document recognition. Proceedings of the IEEE **86**(11), 2278–2324 (1998)
17. Lou, Q., Lu, W., Hong, C., Jiang, L.: Falcon: Fast spectral inference on encrypted data (2020)
18. Mishra, P., Lehmkuhl, R., Srinivasan, A., Zheng, W., Popa, R.A.: Delphi: A cryptographic inference service for neural networks. In: Capkun, S., Roesner, F. (eds.) 29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020. pp. 2505–2522. USENIX Association (2020)
19. Xiao, H., Rasul, K., Vollgraf, R.: Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms. CoRR **abs/1708.07747** (2017), http://arxiv.org/abs/1708.07747