Low-Complexity Deep Convolutional Neural Networks on Fully Homomorphic Encryption Using Multiplexed Convolutions

Eunsang Lee
Department of Electrical and Computer Engineering, INMC, Seoul National University
shaeunsang@snu.ac.kr

Joon-Woo Lee*
Department of Electrical and Computer Engineering, INMC, Seoul National University
joonwoo42@snu.ac.kr

Junghyun Lee
Department of Electrical and Computer Engineering, INMC, Seoul National University
ljhfree530@snu.ac.kr

Young-Sik Kim
Department of Information and Communication Engineering, Chosun University
iamyskim@chosun.ac.kr

Yongjune Kim
Department of Information and Communication Engineering, DGIST
yjk@dgist.ac.kr

Jong-Seon No
Department of Electrical and Computer Engineering, INMC, Seoul National University
jsno@snu.ac.kr

Woosuk Choi
Samsung Advanced Institute of Technology
woosuk0.choi@samsung.com

ABSTRACT

Privacy-preserving machine learning on fully homomorphic encryption (FHE) is one of the most influential applications of the FHE scheme. Recently, Lee et al. [16] implemented the standard ResNet-20 model for the CIFAR-10 dataset with residue number system variant Cheon-Kim-Kim-Song (RNS-CKKS) scheme, one of the most promising FHE schemes, for the first time. However, its implementation should be improved because it requires large number of key-switching operations, which is the heaviest operation in the RNS-CKKS scheme. In order to reduce the number of key-switching operations, it should be studied to efficiently perform neural networks on the RNS-CKKS scheme utilizing full slots of RNS-CKKS ciphertext as much as possible. In particular, since the packing density is reduced to 1/4 whenever a convolution of stride two is performed, it is required to study convolution that maintains packing density of the data. On the other hand, when bootstrapping should be performed, it is desirable to use sparse slot bootstrapping that requires fewer key-switching operations instead of full slot bootstrapping. In this paper, we propose a new packing method that makes several tensors for multiple channels to be multiplexed into one tensor. Then, we propose new convolution method that outputs a multiplexed tensor for the input multiplexed tensor, which makes it possible to maintain a high packing density during the entire ResNet with strided convolution. In addition, we propose a method that parallelly performs convolutions for multiple output channels using repeatedly packed input data, which reduces the running time of convolution. Further, we fine-tune the parameters to reach the standard 128-bit security level and to further reduce the number of the bootstrapping operations. As a result, the number of key-switching operations is reduced to 1/107 compared to Lee et al.'s implementation in the ResNet-20 model on the RNS-CKKS scheme. The proposed method takes about 37 minutes with only one thread for classification of one CIFAR-10 image compared to 3 hours with 64 threads of Lee et al.'s implementation. Furthermore, we even implement ResNet-32/44/56/110 models for the first time on RNS-CKKS scheme with the linear time of the number of layers, which is generally difficult to be expected in the leveled homomorphic encryption. Finally, we successfully classify the CIFAR-100 dataset on RNS-CKKS scheme using standard ResNet-32 model, and we obtain a running time of 3,942s and an accuracy of 69.4% close to the accuracy of backbone network 69.5%.

KEYWORDS

Artificial intelligence; Cheon-Kim-Kim-Song (CKKS); Convolution; Fully homomorphic encryption (FHE); Privacy-preserving machine learning (PPML); Residue number system variant Cheon-Kim-Kim-Song (RNS-CKKS); ResNet model

1 INTRODUCTION

Various performance improvements have been made in artificial intelligence due to the emergence of deep learning, and its practicality has been improved a lot. However, if the privacy of the users' data is not guaranteed, the users may not be able to use artificial intelligence services due to privacy leakage. In some cases, enterprises having high-end artificial intelligence technologies may not be able to provide their services due to legal issues of data privacy. For these reasons, providing artificial intelligence services preserving privacy of users' data, called privacy-preserving machine learning (PPML), has become a more important research topic [21].

A possible way to implement PPML is to use homomorphic encryption (HE), which allows algebraic operations on the encrypted data. Although fully homomorphic encryption (FHE) technologies that allow algebraic operations without restriction of number of operations have also been proposed, the high latency of FHE operations has been pointed out as the reason for the low practicality of FHE schemes. Many works overcame the problem by using HE-friendly network, that is, the neural network that has small number of layers or low-degree approximate polynomial of activation function using leveled HE schemes [8, 10, 18, 19] or combining HE with multi-party computation (MPC) technique [2, 12, 20, 22]. However, with the recent reduction of latency of FHE operations, the practicality of FHE in the PPML has been increased significantly [1, 11, 23]. Hence, the PPML only using FHE has become practically possible.

*Corresponding author

Packing in HE is the method that loads multiple data into a single ciphertext. Since operations on multiple data can be performed simultaneously by one operation on ciphertext, it would be desirable to pack data into the ciphertext as many as possible for efficient implementation of the deep neural network. However, it is not easy to perform a desired task such as convolution using only fully-packed ciphertexts because the operations between slots are very limited on the packed ciphertext. Since higher packing density leads lower ciphertext overhead and run-time overhead, some studies [7, 12] have been studied deep neural networks that utilize HE ciphertexts with high packing density. As for hybrid methods using HE and MPC technique, the packing density in HE ciphertext could be increased by re-encrypting using MPC technique even if the packing density was temporarily lowered. However, if PPML model is implemented using only FHE, maintaining the packing density during the entire PPML model is a difficult problem since re-encryption is not allowed and the modification of data structure is usually expensive in HE. In the case of using a low packing density, the computation time increases considerably because the total number of key-switching operations required to process the same data is inversely proportional to the packing density.

The main factor that decreases the packing density is strided convolution, i.e., convolution of stride larger than one. If the strided convolution is performed, the data structure in ciphertexts is changed. However, it is difficult to maintain the packing structure after the strided convolution without the help of MPC technique. Although Lee et al. [16] implemented strided convolution on the residue number system variant Cheon-Kim-Kim-Song (RNS-CKKS) scheme, the packing density is reduced to 1/4 after convolution of stride two. Thus, it is important to devise an efficient convolution method on FHE that maintains packing density in deep neural networks with strided convolutions.

1.1 Our Contribution

First, we propose a new packing method, `MultPack` that makes tensors for multiple channels to be multiplexed into one tensor. Then, we propose a convolution algorithm, `MultConv` that outputs a multiplexed tensor for the input multiplexed tensor. Using this algorithm, the entire ResNet network can be performed while maintaining high packing density of data. As a result, the total number of key-switching operations is significantly reduced compared to the previous work [16], where key-switching is the heaviest operation in the RNS-CKKS scheme.

We also propose a faster convolution algorithm, `PrlMultConv` that parallelly performs convolutions for multiple output channels. This method packs several repeated input tensors in one ciphertext, encodes filters to multiply each repeated tensor by different weight values, and performs convolution so that convolutions for multiple output channels can be performed simultaneously. Further, while this convolution algorithm utilizes full slots, repeatedly packed data can be considered as sparsely packed data during bootstrapping, and thus faster sparse slot bootstrapping can still be used instead of full slot bootstrapping.

As an additional contribution, we use lighter and tighter parameters than Lee et al.'s work to reduce level consumption while achieving almost the same classification accuracy. Instead of using two bootstrapping per layer and per ciphertext in Lee et al.'s work, we use only one bootstrapping per layer and per ciphertext. As a result, the total number of key-switching operations is reduced to 1/107 compared to Lee et al.'s implementation of the ResNet-20 on RNS-CKKS scheme [16].

	Lee et al. [16]	**proposed**	ratio
latency (s)	10,602	**2,271**	4.67
number of threads for one image	64	**1**	64
amortized running time (s)	10,602	**79**	134.2
number of key-switching operations	375,920	**3,489**	107.7
security level (bits)	111.6	**128**	0.87

Table 1: Comparison of various performance between Lee et al.'s implementation and the proposed implementation of the standard ResNet-20

The efficiency of the proposed methods is numerically confirmed with the SEAL library [26], which is one of the representative RNS-CKKS libraries. We implement ResNet-20 for the CIFAR-10 dataset on RNS-CKKS scheme using SEAL library. Table 1 shows various performance improvement compared to Lee et al.'s work [16]. The latency is reduced to 1/4.67, and it will be far more significant than Lee et al.'s implementation using many threads when we implement it with various hardware accelerators. Since we use only one thread rather than many threads for one image, we are able to infer many images simultaneously with many threads. If we perform the inference of 50 images with 50 threads, it takes 3,972s for total running time, whose amortized running time is 79s. It can be regarded as reduction to 1/134 compared to Lee et al.'s work in terms of the amortized running time. The security is increased to 128-bit security compared to the previous 111-bit security. Therefore, we significantly improve the performance in the aspect of latency, amortized running time, and security in the ResNet-20 on the RNS-CKKS scheme.

Further, we implement ResNet-32/44/56/110 models on RNS-CKKS scheme, and it is the first implementation result for these standard deep neural networks. From this result, it can be seen that the running time of these models is linear with the number of layers, while linear time with the number of layers is generally difficult to be expected on the leveled HE. Finally, we successfully classify the CIFAR-100 dataset on RNS-CKKS scheme using standard ResNet-32 model, and we obtain a running time of 3,942s and an accuracy of 69.4% close to the accuracy of backbone network 69.5%.

1.2 Related Works

Several works have been done in the standard deep neural networks using HE without any additional cryptographic techniques. Lou et al. [18] implemented the standard deep neural networks such as ResNet-18 and ResNet-20 with leveled version of fast fully homomorphic encryption over the torus (TFHE) [5]. Since the leveled HE sets parameters to support the depth of particular operations in advance so that bootstrapping is not required, impractically large

encryption parameters can be required to implement deeper neural networks when using the leveled HE [6]. Instead of the leveled HE, it is desirable to implement deeper neural networks using FHE that supports bootstrapping and arbitrarily deep operations with fixed practical parameters.

Chilloti et al. [6] implemented deeper neural networks on the fully homomorphic version of TFHE scheme with MNIST dataset. Although they showed some possibility of implementing deep neural networks on FHE scheme, the dataset to be classified is still rather simple, and the classification accuracy is not sufficiently high. Lee et al. [16] implemented ResNet-20 on RNS-CKKS scheme using bootstrapping. Although it is the first study to apply bootstrapping of the CKKS scheme to deep neural networks, it requires large number of key-switching operations. Further, because 64 threads are continuously used, it can only be performed on CPUs that support at least 64 threads. The total number of key-switching operations is far more important when using accelerator with many cores such as GPU or FPGA, and thus the significant reduction of total number of key-switching operations contributes greatly to practicality for PPML with FHE.

There are also many works implementing neural networks using MPC techniques partly [2, 12, 20, 22] or wholly [17, 24, 25]. Using MPC techniques, low latency is usually obtained, and low communication cost can also be achieved using HE-friendly network. However, it takes large communication cost for standard deep neural networks that require many invocations of activation functions. Previous works using MPC take several gigabytes of communication cost to infer one CIFAR-10 image in the standard ResNet models, which makes the implementations less practical for some environments not supporting communication resources enough.

Gazelle [12] and Cheetah [22] are hybrid PPML models in that they use both HE and MPC technique. Gazelle and Cheetah's strided convolution focuses only on how to perform the operation, and there is no consideration for subsequent homomorphic operations. The reason is that data can be decrypted and re-encrypted using MPC techniques, which allows packed data to be restructured to facilitate subsequent homomorphic operations. However, in order to perform the entire deep neural network only with FHE, new difficulties arise because subsequent convolution operations should be performed without re-encryption process and restructuring packing data.

2 PRELIMINARIES

2.1 Notations

We use $\mathbf{x}$ to denote a vector in $\mathbb{R}^n$ for some integer n. For $\mathbf{x} = (x_0, x_1, \cdots, x_{n-1})$, $\langle \mathbf{x} \rangle_r$ denote the cyclically shifted vector of $\mathbf{x}$ by r to the left, that is, $(x_r, x_{r+1}, \cdots, x_{n-1}, x_0, \cdots, x_{r-1})$. $\mathbf{x} \cdot \mathbf{y}$ denotes the component-wise multiplication $(x_0 y_0, \cdots, x_{n-1} y_{n-1})$. For an integer $a \in \mathbb{Z}$, the remainder of a divided by q is denoted by $a \bmod q$. For a real number $x \in \mathbb{R}$, $\lceil x \rceil$ denotes the least integer greater than or equal to x, and $\lfloor x \rfloor$ denotes the greatest integer less than or equal to x.

2.2 Description of Parameters

In this paper, various parameters such as $h_i, h_o, w_i, w_o, c_i, c_o, f_h, f_w, s, k_i, k_o, t_i, t_o, p_i, p_o$, and q are used, and the values of these parameters are determined differently for each component such as convolution, batch normalization (or convolution/batch normalization integration in Section 5), downsampling, and average pooling.

First, for a three-dimensional input *tensor* of each component $A = (A_{i_1 i_2 i_3})_{0 \le i_1 < h_i, 0 \le i_2 < w_i, 0 \le i_3 < c_i} \in \mathbb{R}^{h_i \times w_i \times c_i}$, h_i and w_i are height and width of input tensor, respectively, and c_i is the number of input channels. For the output tensor $A' = (A'_{i_1 i_2 i_3})_{0 \le i_1 < h_o, 0 \le i_2 < w_o, 0 \le i_3 < c_o} \in \mathbb{R}^{h_o \times w_o \times c_o}$, h_o and w_o are height and width of output tensor, respectively, and c_o is the number of output channels. In the convolution, f_h and f_w are kernel sizes of the filter. In this paper, the horizontal and vertical strides of the convolution are assumed to be the same for simplicity, and we denote the stride of the convolution by s. k_i and k_o are the gaps of input and output tensors (described in Section 3 in detail), respectively. We have $t_i = \lceil \frac{c_i}{k_i^2} \rceil$, $t_o = \lceil \frac{c_o}{k_o^2} \rceil$, $p_i = 2^{\lfloor \log_2(\frac{n_t}{k_i^2 h_i w_i t_i}) \rfloor}$, $p_o = 2^{\lfloor \log_2(\frac{n_t}{k_o^2 h_o w_o t_o}) \rfloor}$, and $q = \lceil \frac{c_o}{p_i} \rceil$, where n_t is the number of full slots of ciphertext. The detailed descriptions of $k_i, k_o, t_i, t_o, p_i, p_o$, and q are given in Section 3, and the specific values of parameters that are used in our simulation can be seen in Table 2 in Section 6.

2.3 RNS-CKKS Fully Homomorphic Encryption

RNS-CKKS FHE scheme is an encryption scheme that supports various fixed-point arithmetic vector operations with unlimited depths on encrypted data. The ciphertext in the RNS-CKKS scheme is the form of $(b, a) \in R_{Q_\ell}^2$, where $Q_\ell = \prod_{i=0}^{\ell} q_i$ is a product of prime numbers and $R_{Q_\ell} = \mathbb{Z}_{Q_\ell}[X]/\langle X^N + 1 \rangle$. Each ciphertext has a non-negative integer called level, which means the capacity for homomorphic multiplication operations, and we denote the level as ℓ.

We denote the encryption and decryption in RNS-CKKS scheme as $\mathsf{Enc}(\cdot)$ and $\mathsf{Dec}(\cdot)$, respectively. The data structure encrypted in a ciphertext of RNS-CKKS scheme is one-dimensional vector with length $N/2$, where N is the degree of the base polynomial of the ring. We denote this length of the vector as n_t, that is, $n_t = N/2$ in this paper. The supported homomorphic operations in RNS-CKKS scheme are described as follows without specific algorithms, where ct, ct_1, ct_2, ct_3, and ct' are ciphertexts, and $\mathbf{u}, \mathbf{v}, \mathbf{v}_1$, and $\mathbf{v}_2$ are vectors in $\mathbb{R}^n$.

- Homomorphic addition and substitution ($\oplus, \ominus$)
  - $\mathsf{ct} \oplus \mathbf{u}$ (resp. $\mathsf{ct} \ominus \mathbf{u}$) $\rightarrow \mathsf{ct}'$: If $\mathsf{Dec}(\mathsf{ct}) = \mathbf{v}$, then $\mathsf{Dec}(\mathsf{ct}') = \mathbf{v} + \mathbf{u}$ (resp. $\mathbf{v} - \mathbf{u}$).
  - $\mathsf{ct}_1 \oplus \mathsf{ct}_2$ (resp. $\mathsf{ct}_1 \ominus \mathsf{ct}_2$) $\rightarrow \mathsf{ct}_3$: If $\mathsf{Dec}(\mathsf{ct}_1) = \mathbf{v}_1$ and $\mathsf{Dec}(\mathsf{ct}_2) = \mathbf{v}_2$, then $\mathsf{Dec}(\mathsf{ct}_3) = \mathbf{v}_1 + \mathbf{v}_2$ (resp. $\mathbf{v}_1 - \mathbf{v}_2$).
- Homomorphic multiplication ($\odot, \otimes$)
  - $\mathsf{ct} \odot \mathbf{u} \rightarrow \mathsf{ct}'$: If $\mathsf{Dec}(\mathsf{ct}) = \mathbf{v}$, then $\mathsf{Dec}(\mathsf{ct}') = \mathbf{v} \cdot \mathbf{u}$.
  - $\mathsf{ct}_1 \otimes \mathsf{ct}_2 \rightarrow \mathsf{ct}_3$: If $\mathsf{Dec}(\mathsf{ct}_1) = \mathbf{v}_1$ and $\mathsf{Dec}(\mathsf{ct}_2) = \mathbf{v}_2$, then $\mathsf{Dec}(\mathsf{ct}_3) = \mathbf{v}_1 \cdot \mathbf{v}_2$.
- Homomorphic rotation (Rot)
  - $\mathsf{Rot}(\mathsf{ct}; r) \rightarrow \mathsf{ct}'$: If $\mathsf{Dec}(\mathsf{ct}) = \mathbf{v}$, then $\mathsf{Dec}(\mathsf{ct}') = \langle \mathbf{v} \rangle_r$.

The homomorphic multiplication between ciphertexts and the homomorphic rotation need the key-switching operation that switches the secret key for a ciphertext to a new secret key without any change for the message. These operations require far more time than any other homomorphic operation. Thus, the number of key-switching operations roughly determines the total amount of operations in homomorphic arithmetic circuits. Lee et al. [16] pointed out that the running time of the key-switching operation is proportional to the required level squared, and thus the total amount of operations is proportional to the sum of squared level of the input ciphertexts of each key-switching operation.

After the homomorphic multiplication is performed, the modulus is divided by one RNS prime number in modulus by rescaling procedure. Thus, the modulus of the ring for the ciphertext is gradually reduced by a chain of homomorphic multiplication operations, and there is a point when further homomorphic multiplication can not be performed. This ciphertext is called level-0 ciphertext. The bootstrapping operation makes this level-0 ciphertext to the higher-level ciphertext for which homomorphic multiplications can be performed. When the deep arithmetic operation is performed homomorphically, the bootstrapping operation requires to be performed periodically.

If the data to be encrypted is a vector with size less than n_t, we can reduce the running time for the bootstrapping with sparse packing technique. The sparse packing technique uses the homomorphism from $\mathbb{Z}[Y]/\langle Y^{2n}+1\rangle$ to $\mathbb{Z}[X]/\langle X^N+1\rangle$, where $Y = X^{N/2n}$ and n is power-of-two integer less than n_t. In this technique, a vector with length n is encoded to a polynomial in $\mathbb{Z}[Y]/\langle Y^{2n}+1\rangle$, and then apply the above homomorphism to this polynomial. If a message is encoded with this technique, the running time for the bootstrapping operation can be reduced further. When the vector to be sparsely packed into the ciphertext is $\mathbf{v} = (v_0, \cdots, v_{n-1}) \in \mathbb{R}^n$, the equivalent vector to this vector in the aspect of full-packing after applying the homomorphism is $(\mathbf{v}|\mathbf{v}|\cdots|\mathbf{v}) \in \mathbb{R}^{n_t}$, which is a vector with n_t/n vectors concatenated.

Due to the efficiency for the bootstrapping, a sparse secret key is generally used when we perform the bootstrapping. We set the Hamming weight of the secret key $s \in \mathbb{Z}^N$ as an additional parameter. If the Hamming weight of the secret key is h, the secret key is uniformly sampled from the set of the signed binary vectors in $\{-1, 0, 1\}^N$ whose Hamming weight is h. In case of using sparse secret key, the security level is determined by the hybrid dual attack in Cheon et al.'s work [4].

2.4 Convolution on Homomorphic Encryption

A three-dimensional *tensor* $A = (A_{i_1 i_2 i_3})_{0\le i_1<h_i, 0\le i_2<w_i, 0\le i_3<c_i} \in \mathbb{R}^{h_i\times w_i\times c_i}$ is the input of convolution, where h_i, w_i, and c_i are height, width, and number of input channels, respectively. The output of the convolution is three-dimensional tensor $A' = (A'_{i_1 i_2 i_3})_{0\le i_1<h_o, 0\le i_2<w_o, 0\le i_3<c_o} \in \mathbb{R}^{h_o\times w_o\times c_o}$. The filter (weight tensor) of the convolution is $U \in \mathbb{R}^{f_h\times f_w\times c_i\times c_o}$, where f_h and f_w are kernel sizes, and c_o is the number of output channels of convolution. In this paper, we only consider convolution using zero paddings.

It is often necessary to map three-dimensional tensor $\overline{A} \in \mathbb{R}^{\overline{h}_i\times\overline{w}_i\times\overline{c}_i}$ to one-dimensional vector in $\mathbb{R}^{n_t}$ to perform convolutions on the HE scheme, where n_t is the number of full slots of ciphertext, and $\overline{A}$ can be the original tensor or *(parallelly) multiplexed tensor* defined in Section 3. The following is the definition of Vec function that is used to map tensor $\overline{A}$ to a vector in $\mathbb{R}^{n_t}$,

$\mathsf{Vec}(\overline{A}) = \mathbf{y} = (y_0, \cdots, y_{n_t-1}) \in \mathbb{R}^{n_t}$ such that

$$y_i = \begin{cases} \overline{A}_{\lfloor (i \bmod \overline{h}_i\overline{w}_i)/\overline{w}_i\rfloor, i \bmod \overline{w}_i, \lfloor i/\overline{h}_i\overline{w}_i\rfloor}, & 0 \le i < \overline{h}_i\overline{w}_i\overline{c}_i \\ 0, & \text{otherwise.} \end{cases}$$

Figure 1 describes this Vec function.

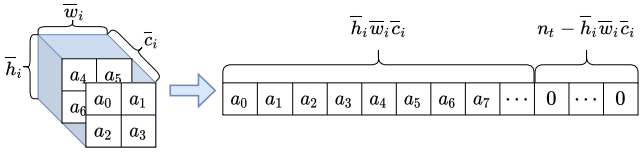


Figure 1: Vec function that maps a given tensor in $\mathbb{R}^{\overline{h}_i\times\overline{w}_i\times\overline{c}_i}$ to a vector in $\mathbb{R}^{n_t}$.

In this paper, we use $n_t = 2^{15}$, and this allows that all tensors to be encrypted can be packed into one ciphertext, that is, $\overline{h}_i\overline{w}_i\overline{c}_i \le n_t$ for each tensor $\overline{A} \in \mathbb{R}^{\overline{h}_i\times\overline{w}_i\times\overline{c}_i}$. In several figures in this paper, a three-dimensional tensor $\overline{A}$ is often identified as $\mathsf{Vec}(\overline{A})$ or the corresponding ciphertext $\mathsf{Enc}(\mathsf{Vec}(\overline{A}))$. In addition, for a three-dimensional tensor $\overline{A}$, we refer to rotation of ciphertext of $\mathsf{Vec}(\overline{A})$, that is, $\mathsf{Rot}(\mathsf{Enc}(\mathsf{Vec}(\overline{A})); r)$ for some nonnegative integer r as rotation of tensor $\overline{A}$. When a tensor is rotated, each element moves to the left, but it goes up when it reaches the leftmost point, and it moves to the front page when it reaches the top leftmost point. Furthermore, for two tensors $\overline{A}$ and $\overline{B}$, homomorphic addition, subtraction, and multiplication of $\mathsf{Enc}(\mathsf{Vec}(\overline{A}))$ and $\mathsf{Enc}(\mathsf{Vec}(\overline{B}))$ are referred to as those of $\overline{A}$ and $\overline{B}$, respectively.

In order to perform convolution on HE, the convolution should be performed using addition, multiplication, and rotation of tensors. In particular, since rotation is the heaviest operation among these operations, it is desirable to reduce the number of rotations as much as possible. Gazelle [12] proposed a method to perform convolution using these operations of tensors. In this method, each shifted input tensor is multiplied by some vector that has appropriate filter coefficients as components, and $f_h f_w$ many multiplication results are added. Then, the added results for all c_i input channels are added to obtain output tensor using diagonal grouping technique [12]. Unlike that in [12], this paper that uses FHE requires large number of full slots $n_t = 2^{15}$, and thus, all values of a tensor can be packed into one ciphertext. Considering this, each convolution of stride one can be performed using diagonal grouping technique with $f_h f_w + c_i - 2$ rotations.

A method to perform strided convolution on HE was also proposed in [12]. The strided convolution is first decomposed into a sum of simple convolutions of stride one, and each convolution of stride one can be performed on HE using the above method.

2.5 Batch Normalization on the Fully Homomorphic Encryption

Batch normalization [9] should be performed for the output tensor of convolution. As in convolution, h_i, w_i, and c_i are parameters representing the size of the input tensor, and h_o, w_o, and c_o are parameters representing the size of the output tensor in batch normalization. That is, batch normalization outputs a tensor $A' \in \mathbb{R}^{h_o \times w_o \times c_o}$ for some input tensor $A \in \mathbb{R}^{h_i \times w_i \times c_i}$. We have $h_i = h_o$, $w_i = w_o$, and $c_i = c_o$ for batch normalization.

We denote the weight, running variance, running mean, and bias of batch normalization by $T, V, M, I \in \mathbb{R}^{c_i}$. We consider a constant vector $C = (C_0, C_1, \cdots, C_{c_i-1}) \in \mathbb{R}^{c_i}$ such that $C_j = \frac{T_j}{\sqrt{V_j+\epsilon}}$ for $0 \le j < c_i$, where ϵ is an added value for numerical stability. Then, batch normalization can be seen as evaluating the equation $C_j \cdot (A_{i_1,i_2,j} - M_j) + I_j$ for $0 \le i_1 < h_i, 0 \le i_2 < w_i$, and $0 \le j < c_i$.

For the description of batch normalization on HE, it is required to define $\overline{C}$, $\overline{M}$, and $\overline{I} \in \mathbb{R}^{h_i \times w_i \times c_i}$ first. We define $\overline{C}$, $\overline{M}$, and $\overline{I}$ as $\overline{C}_{i_1,i_2,j} = C_j$, $\overline{M}_{i_1,i_2,j} = M_j$, and $\overline{I}_{i_1,i_2,j} = B_j$ for $0 \le i_1 < h_i, 0 \le i_2 < w_i$, and $0 \le j < c_i$, respectively. Then, batch normalization can be performed using the equation $\mathrm{Vec}(\overline{C}) \cdot (\mathrm{Vec}(A) - \mathrm{Vec}(\overline{M})) + \mathrm{Vec}(\overline{I}) = \mathrm{Vec}(\overline{C}) \cdot \mathrm{Vec}(A) + (\mathrm{Vec}(\overline{I}) - \mathrm{Vec}(\overline{C}) \cdot \mathrm{Vec}(\overline{M}))$. This can be implemented on HE by using one homomorphic addition and scalar multiplication. That is, for the input tensor ciphertext $\mathrm{ct_a}$, we just perform $\mathrm{Vec}(\overline{C}) \odot \mathrm{ct_a} \oplus (\mathrm{Vec}(\overline{I}) - \mathrm{Vec}(\overline{C}) \cdot \mathrm{Vec}(\overline{M}))$.

3 MULTIPLEXED CONVOLUTION ON FULLY HOMOMORPHIC ENCRYPTION

Although efficient convolution method on HE was proposed in [12], performing the entire deep neural network on the RNS-CKKS scheme without MPC technique has several differences from that in [12] that uses both HE and MPC technique. The biggest difference is strided convolution. Although it is possible to efficiently perform a neural network consisting of only simple convolutions (i.e., convolution of stride one) on FHE using the convolution method in [12], the situation becomes complicated in the standard deep neural network such as ResNet that includes one or more strided convolution. In [12], they first decompose the input tensor into four tensors and then perform simple convolution independently for each tensor to perform strided convolution. However, the decomposition process using only FHE operations such as addition, multiplication, and rotation is very challenging because it is difficult to convert data separated by two into continuous data using FHE operations. In [16], they just perform simple convolution to perform strided convolution of stride two, and only a quarter of output values are considered to be valid and used for subsequent layers. That is, the packing density is reduced to 1/4 whenever it passes through the strided convolution of stride two with this method, which causes inefficiency because the slots of FHE ciphertexts are not fully used.

In addition to the strided convolution, there are also other considerations for implementing neural network on FHE. While small number of full slots were used in [12], we require large number of full slots for FHE. Thus, two or more tensors can be packed into one ciphertext, and it is desirable to make most of ciphertext's full slots to reduce the numbers of bootstrapping and rotation. On the other hand, since the sparse slot bootstrapping takes less time than the full slot bootstrapping, it is desirable to use the sparse slot bootstrapping.

In Section 3.1, we propose a convolution algorithm `MultConv` that allows us to perform the entire deep neural network including strided convolution on FHE while maintaining high packing density. Furthermore, in Section 3.2, we propose a faster convolution algorithm `PrlMultConv`, an optimized algorithm of `MultConv`, that utilizes both operations of full slot and sparse slot bootstrappings.

3.1 `MultConv`: Convolution Using Multiplexed Packing

In [12], while performing the entire deep neural network, each tensor is packed to one-dimensional vector in a raster scan fashion using Vec function. In this paper, we propose a new *multiplexed packing* method called MultPack that makes tensors for multiple channels to be multiplexed into one tensor for some gap k_i. For $t_i = \lceil \frac{c_i}{k_i^2} \rceil$, MultPack is the function that maps a tensor $A = (A_{i_1,i_2,i_3})_{0 \le i_1 < h_i, 0 \le i_2 < w_i, 0 \le i_3 < c_i} \in \mathbb{R}^{h_i \times w_i \times c_i}$ to a vector $\mathrm{Vec}(A') \in \mathbb{R}^{n_t}$, where $A' = (A'_{i_3,i_4,i_5})_{0 \le i_3 < k_i h_i, 0 \le i_4 < k_i w_i, 0 \le i_5 < t_i} \in \mathbb{R}^{k_i h_i \times k_i w_i \times t_i}$ is a *multiplexed tensor* such that

$$A'_{i_3,i_4,i_5} = \begin{cases} A_{\lfloor i_3/k_i \rfloor, \lfloor i_4/k_i \rfloor, k_i^2 i_5 + k_i (i_3 \bmod k_i) + i_4 \bmod k_i}, \\ \qquad \text{if } k_i^2 i_5 + k_i (i_3 \bmod k_i) + i_4 \bmod k_i < c_i \\ 0, \qquad \text{otherwise,} \end{cases}$$

for $0 \le i_3 < k_i h_i, 0 \le i_4 < k_i w_i$, and $0 \le i_5 < t_i$. Figure 2 describes how to perform multiplexed packing when $h_i = w_i = 3$ and $k_i = 2$. If $k_i^2 t_i > c_i$, then we fill the remaining space with zero.

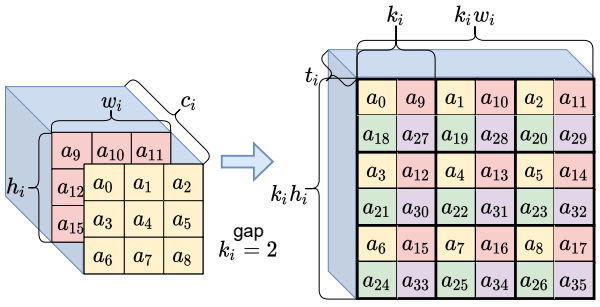


Figure 2: Multiplexed packing MultPack when $h_i = w_i = 3$ and $k_i = 2$.

This multiplexed packing method is a generalized version of raster scan packing method, and it is the same as raster scan packing method using Vec when $k_i = 1$. We require each tensor to be packed into the ciphertext slots using the multiplexed packing method throughout the entire deep neural network, where the value of gap k_i can be changed.

Now, we propose a *multiplexed convolution* algorithm `MultConv` using the addition, multiplication, and rotation of multiplexed tensors for the given input multiplexed tensor. `MultConv` takes a multiplexed tensor for gap k_i as an input and outputs a multiplexed tensor for output gap k_o, where $k_o = sk_i$.

To perform the proposed multiplexed convolution, we first multiply input and weight tensors that are shifted by input gap k_i. Next, we add the multiplied tensors for c_i input channels using addition and rotation. If the input gap k_i is greater than one, leftward and upward rotations of tensors as well as forward rotation are required. Then, we obtain c_o multiplexed tensors, where each multiplexed tensor has only $h_i w_i$ valid values in the front page, and all other values are invalid garbage values. Finally, we select only valid values in each multiplexed tensor and place all the valid values in one ciphertext. Figure 3 shows the convolution with stride $s = 2$ using `MultConv` algorithm for input multiplexed tensor for $k_i = 2$ and $h_i = w_i = 4$.

The greatest advantage of the proposed `MultConv` algorithm is that even if we perform convolution with stride s larger than one, we can perform subsequent layers without reduction of packing density. For example, for convolution with stride $s = 2$, we simply perform multiplexed convolution algorithm `MultConv` for $k_o = 2k_i$ to obtain output multiplexed tensor with gap k_o. In this case, the packing density is not reduced after the convolution with stride two, which is different from that in [16]. Now, we describe the `MultConv` algorithm in detail. For description of `MultConv` algorithm, we require some definitions and a subroutine algorithm.

First, we define $\mathsf{MultWgtPack}(U; i_1, i_2, i)$ function that maps a weight tensor $U \in \mathbb{R}^{f_h \times f_w \times c_i \times c_o}$ to an element of $\mathbb{R}^{n_t}$. Before the definition of MultWgtPack, we define three-dimensional *multiplexed shifted weight tensor* $\overline{U}'^{(i_1,i_2,i)} = (\overline{U}'^{(i_1,i_2,i)}_{i_3,i_4,i_5})_{0 \le i_3 < k_i h_i, 0 \le i_4 < k_i w_i, 0 \le i_5 < t_i} \in \mathbb{R}^{k_i h_i \times k_i w_i \times t_i}$ for given i_1, i_2, and i, where $0 \le i_1 < f_h, 0 \le i_2 < f_w$, and $0 \le i < c_o$ as follows:

$$\overline{U}'^{(i_1,i_2,i)}_{i_3,i_4,i_5} = \begin{cases} 0, & \text{if } k_i^2 i_5 + k_i(i_3 \bmod k_i) + i_4 \bmod k_i \ge c_i \\ & \quad \text{or } \lfloor i_3/k_i \rfloor - (f_h-1)/2 + i_1 \notin [0, h_i - 1] \\ & \quad \text{or } \lfloor i_4/k_i \rfloor - (f_w-1)/2 + i_2 \notin [0, w_i - 1], \\ U_{i_1,i_2,k_i^2 i_5 + k_i(i_3 \bmod k_i) + i_4 \bmod k_i, i}, & \text{otherwise,} \end{cases}$$

for $0 \le i_3 < k_i h_i$, $0 \le i_4 < k_i w_i$, and $0 \le i_5 < t_i$. Then, MultWgtPack function is defined as $\mathsf{MultWgtPack}(U; i_1, i_2, i) = \mathsf{Vec}(\overline{U}'^{(i_1,i_2,i)})$.

In addition to the weight tensor, it is also required to define *multiplexed selecting tensor* $S'^{(i)} = (S'^{(i)}_{i_3,i_4,i_5})_{0 \le i_3 < k_o h_o, 0 \le i_4 < k_o w_o, 0 \le i_5 < t_o} \in \mathbb{R}^{k_o h_o \times k_o w_o \times t_o}$, which is used to select valid values in `MultConv` algorithm, where $t_o = \lfloor \frac{c_o}{k_o^2} \rfloor$. Multiplexed selecting tensor $S'^{(i)}$ is defined as follows:

$$S'^{(i)}_{i_3,i_4,i_5} = \begin{cases} 1, & \text{if } k_o^2 i_5 + k_o(i_3 \bmod k_o) + i_4 \bmod k_o = i \\ 0, & \text{otherwise,} \end{cases}$$

for $0 \le i_3 < k_o h_o$, $0 \le i_4 < k_o w_o$, and $0 \le i_5 < t_o$.

`SumSlots` is a useful subroutine algorithm that adds m slot values spaced apart by p. Algorithm 1 shows the `SumSlots` algorithm. Then, Algorithm 2 describes the proposed multiplexed convolution algorithm, `MultConv` using MultWgtPack function, multiplexed selecting tensor $S'^{(i)}$, and `SumSlots` algorithm. Here, $\mathrm{ct}_{\mathrm{zero}}$ is a ciphertext of all-zero vector $\mathbf{0} \in \mathbb{R}^{n_t}$.

Algorithm 1: $\mathsf{SumSlots}(\mathrm{ct_a}; m, p)$

Input: Tensor ciphertext $\mathrm{ct_a}$, number of added slots m, and gap p
Output: Tensor ciphertext $\mathrm{ct_c}$

$\mathrm{ct_b}^{(0)} \leftarrow \mathrm{ct_a}$
for $j \leftarrow 1$ **to** $\lfloor \log_2 m \rfloor$ **do**
  $\mathrm{ct_b}^{(j)} \leftarrow \mathrm{ct_b}^{(j-1)} \oplus \mathsf{Rot}(\mathrm{ct_b}^{(j-1)}; 2^{j-1} \cdot p)$
end
$\mathrm{ct_c} \leftarrow \mathrm{ct_b}^{(\lfloor \log_2 m \rfloor)}$
for $j \leftarrow 0$ **to** $\lfloor \log_2 m \rfloor - 1$ **do**
  if $\lfloor m/2^j \rfloor \bmod 2 = 1$ **then**
    $\mathrm{ct_c} \leftarrow \mathrm{ct_c} \oplus \mathsf{Rot}(\mathrm{ct_b}^{(j)}; \lfloor m/2^{j+1} \rfloor \cdot 2^{j+1} p)$
  end
end
return $\mathrm{ct_c}$

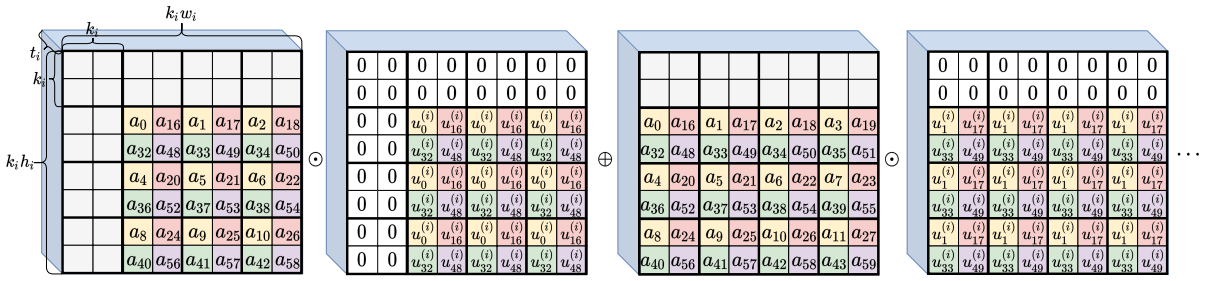
Algorithm 2: $\mathsf{MultConv}(\mathrm{ct'_a}, U)$

Input: Multiplexed tensor ciphertext $\mathrm{ct'_a}$ and weight tensor U
Output: Multiplexed tensor ciphertext $\mathrm{ct'_d}$

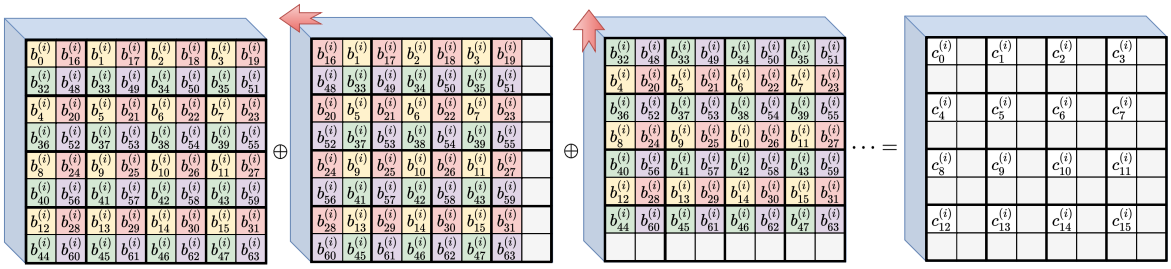
$\mathrm{ct'_d} \leftarrow \mathrm{ct_{zero}}$
for $i_1 \leftarrow 0$ **to** $f_h - 1$ **do**
  for $i_2 \leftarrow 0$ **to** $f_w - 1$ **do**
    $\mathrm{ct}'^{(i_1,i_2)} \leftarrow \mathsf{Rot}(\mathrm{ct'_a}; k_i^2 w_i(i_1 - (f_h-1)/2) + k_i(i_2 - (f_w-1)/2))$
  end
end
for $i \leftarrow 0$ **to** $c_o - 1$ **do**
  $\mathrm{ct'_b} \leftarrow \mathrm{ct_{zero}}$
  for $i_1 \leftarrow 0$ **to** $f_h - 1$ **do**
    for $i_2 \leftarrow 0$ **to** $f_w - 1$ **do**
      $\mathrm{ct'_b} \leftarrow \mathrm{ct'_b} \oplus \mathrm{ct}'^{(i_1,i_2)} \odot \mathsf{MultWgtPack}(U; i_1, i_2, i)$
    end
  end
  $\mathrm{ct'_c} \leftarrow \mathsf{SumSlots}(\mathrm{ct'_b}; k_i, 1)$
  $\mathrm{ct'_c} \leftarrow \mathsf{SumSlots}(\mathrm{ct'_c}; k_i, k w_i)$
  $\mathrm{ct'_c} \leftarrow \mathsf{SumSlots}(\mathrm{ct'_c}; t_i, k^2 h_i w_i)$
  $\mathrm{ct'_d} \leftarrow \mathrm{ct'_d} \oplus \mathsf{Rot}(\mathrm{ct'_c}; -\lfloor i/k_o^2 \rfloor k_o^2 h_o w_o - \lfloor (i \bmod k_o^2)/k_o \rfloor k_o w_o - (i \bmod k_o)) \odot \mathsf{Vec}(S'^{(i)})$
end
return $\mathrm{ct'_d}$

3.2 `PrlMultConv`: Parallel Convolution Using Multiplexed Packing

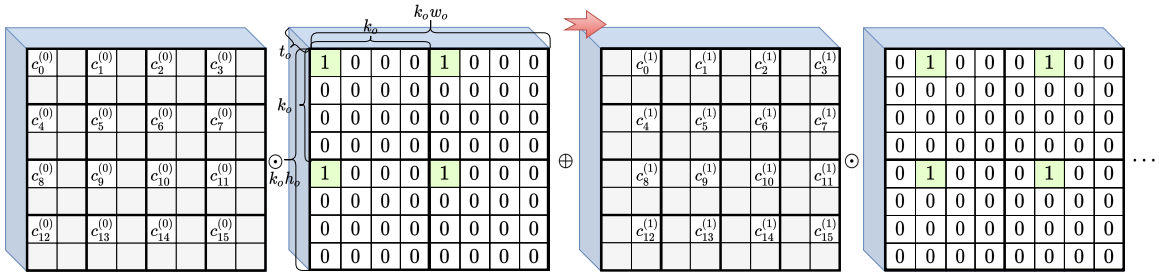
During the classification using ResNet models, there are many cases when the total size of tensor is far less than the capacity of the ciphertext, $n_t = 2^{15}$. The sparse packing technique [3] is an appropriate choice for this case in that the running time of the bootstrapping can be reduced when the sparsely packed ciphertext is used. However, if we fully use all slots of the ciphertext, we can



(a) Multiplication by filter coefficients



(b) Summation for all input channels



(c) Collecting valid values into one ciphertext

Figure 3: Strided convolution with stride $s = 2$ using `MultConv` algorithm for input multiplexed tensor for $k_i = 2$ and $h_i = w_i = 4$.

additionally reduce the number of rotation operations in the convolution process. Thus, we propose more faster convolution method that can utilize both operations of full slot and the bootstrapping with reduced running time for sparsely packed ciphertext. When the convolution is performed, we regard the sparsely packed ciphertext as the fully packed ciphertext by repetition so that several convolution operations can be performed parallelly. On the other hand, when the bootstrapping is performed, we consider the repeatedly packed input ciphertext as the sparsely packed ciphertext and use sparse slot bootstrapping.

Specifically, we propose a *parallelly multiplexed packing* method PrlMultPack that packs p_i identical multiplexed tensors into one-dimensional vector for $p_i = 2^{\lfloor \log_2(\frac{n_t}{k_i^2 h_i w_i t_i}) \rfloor}$. Figure 4 describes how to perform parallelly multiplexed packing of $3 \times 3 \times c_i$ input tensor for given gap $k_i = 2$ and number of copies p_i. For the input tensor $A \in \mathbb{R}^{h_i \times w_i \times c_i}$, this function first obtains a multiplexed tensor $A' \in \mathbb{R}^{k_i h_i \times k_i w_i \times t_i}$ such that $\mathsf{MultPack}(A) = \mathsf{Vec}(A')$ and simply places p_i copies of A' in sequence. Then, this extended tensor should be mapped to a vector in $\mathbb{R}^{n_t}$ using Vec function. If $k_i^2 h_i w_i t_i \nmid n_t$, we fill some zeros between p_i copies of A'. The definition of PrlMultPack function is given as:

$$\mathsf{PrlMultPack}(A) = \sum_{j=0}^{p_i - 1} \langle \mathsf{MultPack}(A) \rangle_{j(n_t/p_i)}.$$

We require each tensor to be packed into the ciphertext slots using the parallelly multiplexed packing method during the entire deep neural network. We propose a *parallel multiplexed convolution* algorithm, `PrlMultConv`, which is an improved algorithm of `MultConv`. `PrlMultConv` takes a *parallelly multiplexed tensor* for gap k_i as an input and outputs a parallelly multiplexed tensor for output gap k_o. Let $q = \lceil \frac{c_o}{p_i} \rceil$. Then, while the previous multiplexed convolution algorithm `MultConv` performs multiplication by weight and summing up c_o times, parallel multiplexed convolution algorithm `PrlMultConv` performs only q times, reducing the required number of rotations to about $1/p_i$.

Before description of `PrlMultConv` in detail, it is required to define $\mathsf{PrlMultWgtPack}(U; i_1, i_2, i_3)$ that maps weight tensor $U \in \mathbb{R}^{h_i \times w_i \times c_i \times c_o}$ to an element of $\mathbb{R}^{n_t}$. To define PrlMultWgtPack,

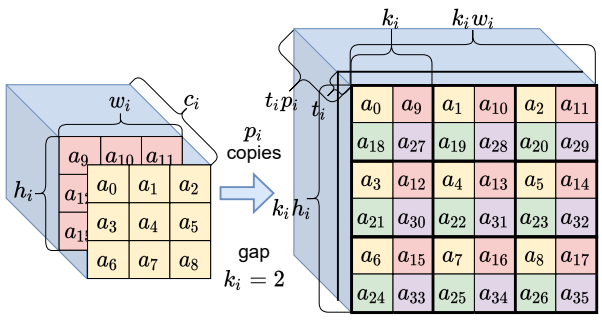


Figure 4: Parallelly multiplexed packing method PrlMultPack when $k_i^2 h_i w_i t_i \mid n_t$.

parallelly multiplexed shifted weight tensor $\overline{U}''^{(i_1,i_2,i_3)} = (\overline{U}''^{(i_1,i_2,i_3)}_{i_5,i_6,i_7})_{0\le i_5<k_ih_i,0\le i_6<k_iw_i,0\le i_7<t_ip_i} \in \mathbb{R}^{k_ih_i\times k_iw_i\times t_ip_i}$ should be defined first for $0 \le i_1 < f_h, 0 \le i_2 < f_w$, and $0 \le i_3 < q$ as follows:

$$\overline{U}''^{(i_1,i_2,i_3)}_{i_5,i_6,i_7} =$$

$$\begin{cases} 0, & \text{if } k_i^2(i_7 \bmod t_i) + k_i(i_5 \bmod k_i) + i_6 \bmod k_i \ge c_i \\ & \quad \text{or } \lfloor i_7/t_i \rfloor + p_i i_3 \ge c_o \\ & \quad \text{or } \lfloor i_5/k_i \rfloor - (f_h-1)/2 + i_1 \notin [0, h_i-1] \\ & \quad \text{or } \lfloor i_6/k_i \rfloor - (f_w-1)/2 + i_2 \notin [0, w_i-1], \\ U_{i_1,i_2,k_i^2(i_7 \bmod t_i)+k_i(i_5 \bmod k_i)+i_6 \bmod k_i, \lfloor i_7/t_i \rfloor + p_i i_3}, & \text{otherwise,} \end{cases}$$

for $0 \le i_5 < k_ih_i$, $0 \le i_6 < k_iw_i$, and $0 \le i_7 < t_ip_i$. Then, PrlMultWgtPack is defined as $\mathsf{PrlMultWgtPack}(U; i_1, i_2, i_3) = \mathsf{Vec}(\overline{U}''^{(i_1,i_2,i_3)})$. The multiplexed selecting tensor $S'^{(i)}$ defined in Section 3.1 is also used in PrlMultConv.

Then, Algorithm 3 shows the proposed parallel multiplexed convolution algorithm PrlMultConv, where $t_o = \lfloor \frac{c_o}{k_o^2} \rfloor$ and $p_o = 2^{\lfloor \log_2(\frac{n_t}{k_o^2 h_o w_o t_o}) \rfloor}$.

Algorithm 3: $\mathsf{PrlMultConv}(\mathrm{ct}''_{\mathrm{a}}, U)$

Input: Parallelly multiplexed tensor ciphertext $\mathrm{ct}''_{\mathrm{a}}$ and weight tensor U
Output: Parallelly multiplexed tensor ciphertext $\mathrm{ct}''_{\mathrm{d}}$

$\mathrm{ct}''_{\mathrm{d}} \leftarrow \mathrm{ct}_{\mathrm{zero}}$
for $i_1 \leftarrow 0$ **to** $f_h - 1$ **do**
  for $i_2 \leftarrow 0$ **to** $f_w - 1$ **do**
    $\mathrm{ct}''^{(i_1,i_2)} \leftarrow \mathrm{Rot}(\mathrm{ct}''_{\mathrm{a}}; k_i^2 w_i(i_1 - (f_h-1)/2) + k_i(i_2 - (f_w-1)/2))$
  end
end
for $i_3 \leftarrow 0$ **to** $q - 1$ **do**
  $\mathrm{ct}''_{\mathrm{b}} \leftarrow \mathrm{ct}_{\mathrm{zero}}$
  for $i_1 \leftarrow 0$ **to** $f_h - 1$ **do**
    for $i_2 \leftarrow 0$ **to** $f_w - 1$ **do**
      $\mathrm{ct}''_{\mathrm{b}} \leftarrow \mathrm{ct}''_{\mathrm{b}} \oplus \mathrm{ct}''^{(i_1,i_2)} \odot \mathsf{PrlMultWgtPack}(U; i_1, i_2, i_3)$
    end
  end
  $\mathrm{ct}''_{\mathrm{c}} \leftarrow \mathsf{SumSlots}(\mathrm{ct}''_{\mathrm{b}}; k_i, 1)$
  $\mathrm{ct}''_{\mathrm{c}} \leftarrow \mathsf{SumSlots}(\mathrm{ct}''_{\mathrm{c}}; k_i, k_i w_i)$
  $\mathrm{ct}''_{\mathrm{c}} \leftarrow \mathsf{SumSlots}(\mathrm{ct}''_{\mathrm{c}}; t_i, k_i^2 h_i w_i)$
  for $i_4 \leftarrow 0$ **to** $\min(p_i - 1, c_o - 1 - p_i i_3)$ **do**
    $i \leftarrow p_i i_3 + i_4$
    $\mathrm{ct}''_{\mathrm{d}} \leftarrow \mathrm{ct}''_{\mathrm{d}} \oplus \mathrm{Rot}(\mathrm{ct}''_{\mathrm{c}}; -\lfloor i/k_o^2 \rfloor k_o^2 h_o w_o + \lfloor n_t/p_i \rfloor (i \bmod p_i) - \lfloor (i \bmod k_o^2)/k_o \rfloor k_o w_o - i \bmod k_o) \odot \mathsf{Vec}(S'^{(i)})$
  end
end
for $j \leftarrow 0$ **to** $\log_2 p_o - 1$ **do**
  $\mathrm{ct}''_{\mathrm{d}} \leftarrow \mathrm{ct}''_{\mathrm{d}} \oplus \mathrm{Rot}(\mathrm{ct}''_{\mathrm{d}}; -2^j(n_t/p_o))$
end
return $\mathrm{ct}''_{\mathrm{d}}$

4 PARALLEL BATCH NORMALIZATION, DOWNSAMPLING, AND AVERAGE POOLING USING MULTIPLEXED PACKING

In Section 3.2, we proposed parallel multiplexed convolution algorithm, PrlMultConv that works for an input parallelly multiplexed tensor. Besides convolution, the ResNet model has also batch normalization and average pooling. For the CIFAR-10 dataset, the ResNet model also has downsampling. Batch normalization, average pooling, and downsampling should be implemented to be also compatible with the parallelly multiplexed packing method. Thus, new batch normalization, downsampling, and average pooling algorithms that work for a parallelly multiplexed tensor packed using the PrlMultPack are proposed in this section.

4.1 Parallel Multiplexed Batch Normalization

We propose an algorithm PrlMultBN that performs batch normalization for a given input parallelly multiplexed tensor. To this end, it is required to define new packing function PrlBNPack that packs batch normalization constant vectors $C, M, I \in \mathbb{R}^{c_i}$ (explained in Section 2.4) properly. For a given input constant vector $H \in \mathbb{R}^{c_i}$, PrlBNPack outputs a vector $\mathbf{h}'' = (h''_0, h''_1, \cdots, h''_{n_t-1}) \in \mathbb{R}^{n_t}$ satisfying

$$h''_j = \begin{cases} 0, & \text{if } j \bmod (n_t/p_i) \ge k_i^2 h_i w_i t_i \\ & \quad \text{or } k_i^2 i_3 + k_i(i_1 \bmod k_i) + i_2 \bmod k_i \ge c_i \\ H_{k_i^2 i_3 + k_i(i_1 \bmod k_i) + i_2 \bmod k_i}, & \text{otherwise,} \end{cases}$$

for $0 \le j < n_t$, where $i_1 = \lfloor ((j \bmod (n_t/p_i)) \bmod k_i^2 h_i w_i)/k_i w_i \rfloor$, $i_2 = (j \bmod (n_t/p_i)) \bmod k_i w_i$, and $i_3 = \lfloor (j \bmod (n_t/p_i))/k_i^2 h_i w_i \rfloor$. We propose PrlMultBN that performs batch normalization using this PrlBNPack packing method, and Algorithm 4 describes the proposed PrlMultBN.

4.2 Parallel Multiplexed Downsampling

ResNet models for the CIFAR-10 dataset require two downsampling processes. We propose DownSamp algorithm that performs

Algorithm 4: $\mathtt{PrlMultBN}(\mathrm{ct}''_{\mathrm{a}}, C, M, I)$

Input: Parallelly multiplexed tensor ciphertext $\mathrm{ct}''_{\mathrm{a}}$ and batch normalization constant vectors $C, M, I \in \mathbb{R}^{c}_{i}$
Output: Parallelly multiplexed tensor ciphertext $\mathrm{ct}''_{\mathrm{b}}$

$\mathbf{c}'' \leftarrow \mathsf{PrlBNPack}(C)$, $\mathbf{m}'' \leftarrow \mathsf{PrlBNPack}(M)$, $\mathbf{i}'' \leftarrow \mathsf{PrlBNPack}(I)$
$\mathrm{ct}''_{\mathrm{b}} \leftarrow \mathbf{c}'' \odot \mathrm{ct}''_{\mathrm{a}} \oplus (\mathbf{i}'' - \mathbf{c}'' \cdot \mathbf{m}'')$
return $\mathrm{ct}''_{\mathrm{b}}$

downsampling for a given input parallelly multiplexed tensor. This prevents the density of valid values from decreasing after downsampling. To specifically describe the proposed downsampling algorithm, it is required to define downsampling selecting tensor $S''^{(i_1,i_2)} = (S''^{(i_1,i_2)}_{i_3,i_4,i_5})_{0\le i_3<k_ih_i,0\le i_4<k_iw_i,0\le i_5<t_i} \in \mathbb{R}^{k_ih_i\times k_iw_i\times t_i}$, which is used to select $4k_i$ valid values. Downsampling selecting tensor $S''^{(i_1,i_2)} = (S''^{(i_1,i_2)}_{i_3,i_4,i_5})_{0\le i_3<k_ih_i,0\le i_4<k_iw_i,0\le i_5<t_i}$ for $0 \le i_1 < k_i$ and $0 \le i_2 < t_i$ is defined as follows:

$$S''^{(i_1,i_2)}_{i_3,i_4,i_5} = \begin{cases} 1, & \text{if } (\lfloor i_3/k_i \rfloor) \bmod 2 = 0 \\ & \text{and } (\lfloor i_4/k_i \rfloor) \bmod 2 = 0 \\ & \text{and } i_3 \bmod k_i = i_1 \\ & \text{and } i_5 = i_2 \\ 0, & \text{otherwise,} \end{cases}$$

for $0 \le i_3 < k_ih_i, 0 \le i_4 < k_iw_i$, and $0 \le i_5 < t_i$. Algorithm 5 describes the proposed downsampling algorithm DownSamp.

Algorithm 5: $\mathtt{Downsamp}(\mathrm{ct}''_{\mathrm{a}})$

Input: Parallelly multiplexed tensor ciphertext $\mathrm{ct}''_{\mathrm{a}}$
Output: Parallelly multiplexed tensor ciphertext $\mathrm{ct}''_{\mathrm{c}}$

$\mathrm{ct}''_{\mathrm{c}} \leftarrow \mathrm{ct}_{\mathrm{zero}}$
for $i_1 \leftarrow 0$ **to** $k_i - 1$ **do**
  for $i_2 \leftarrow 0$ **to** $t_i - 1$ **do**
    $i_3 \leftarrow \lfloor ((k_ii_2 + i_1) \bmod 2k_o)/2 \rfloor$
    $i_4 \leftarrow (k_ii_2 + i_1) \bmod 2$
    $i_5 \leftarrow \lfloor (k_ii_2 + i_1)/2k_o \rfloor$
    $\mathrm{ct}''_{\mathrm{b}} \leftarrow \mathrm{ct}''_{\mathrm{a}} \odot \mathsf{Vec}(S''^{(i_1,i_2)})$
    $\mathrm{ct}''_{\mathrm{c}} \leftarrow \mathrm{ct}''_{\mathrm{b}} \oplus \mathsf{Rot}(\mathrm{ct}''_{\mathrm{b}}; k_i^2h_iw_i(i_2-i_5) + k_iw_i(i_1-i_3) - k_ii_4)$
  end
end
$\mathrm{ct}''_{\mathrm{c}} \leftarrow \mathsf{Rot}(\mathrm{ct}''_{\mathrm{c}}; -k_o^2h_ow_ot_i/8)$
for $j \leftarrow 0$ **to** $\log_2 p_o - 1$ **do**
  $\mathrm{ct}''_{\mathrm{c}} \leftarrow \mathrm{ct}''_{\mathrm{c}} \oplus \mathsf{Rot}(\mathrm{ct}''_{\mathrm{c}}; -2^jk_o^2h_ow_ot_o)$
end
return $\mathrm{ct}''_{\mathrm{c}}$

4.3 Average Pooling

When we reach the average pooling after performing all convolutions, batch normalizations, and ReLU functions in the ResNet model, we have a ciphertext that contains data packed using PrlMultPack packing method. The data packed by this multiplexed packing method is arranged in a complex order in one dimension, which limits execution of fully connected layer. Thus, we propose an average pooling algorithm AvgPool that not only performs average pooling but also rearranges indices.

Average pooling is the process that obtains a vector of $\mathbb{R}^{c_i}$ by computing the average value for h_iw_i values for an input tensor of $\mathbb{R}^{h_i\times w_i\times c_i}$. To this end, we can add h_iw_i values using rotations and additions of tensors. Dividing by h_iw_i can be performed instead in the process of multiplying selecting vector. Then, in each page, only k_i^2 values are valid out of the $k_i^2h_iw_i$ values, and the rest are the invalid garbage values. We place only $k_i^2t_i$ valid values sequentially in one-dimensional vector. For this rearranging process, it is required to define selecting vector $\bar{\mathbf{s}}'^{(i_3)} = (\bar{\mathbf{s}}'^{(i_3)}_j)_{0\le j<n_t} \in \mathbb{R}^n$, which is defined as follows:

$$\bar{\mathbf{s}}'^{(i_3)}_j = \begin{cases} \frac{1}{h_iw_i}, & \text{if } j - k_ii_3 \in [0, k_i - 1] \\ 0, & \text{otherwise,} \end{cases}$$

for $0 \le j < n_t$ and $0 \le i_3 < k_it_i$. Algorithm 6 shows the proposed average pooling algorithm that uses this selecting vector. Figure 6 describes the rearranging process that selects and places $k_i^2t_i$ valid values sequentially in Algorithm 6.

Algorithm 6: $\mathtt{AvgPool}(\mathrm{ct}''_{\mathrm{a}})$

Input: Parallelly multiplexed tensor ciphertext $\mathrm{ct}''_{\mathrm{a}}$
Output: One-dimensional array ciphertext ct_{b}

$\mathrm{ct}_{\mathrm{b}} \leftarrow \mathrm{ct}_{\mathrm{zero}}$
for $j \leftarrow 0$ **to** $\log_2 w_i - 1$ **do**
  $\mathrm{ct}''_{\mathrm{a}} \leftarrow \mathsf{Rot}(\mathrm{ct}''_{\mathrm{a}}; 2^j \cdot k_i)$
end
for $j \leftarrow 0$ **to** $\log_2 h_i - 1$ **do**
  $\mathrm{ct}''_{\mathrm{a}} \leftarrow \mathsf{Rot}(\mathrm{ct}''_{\mathrm{a}}; 2^j \cdot k_i^2w_i)$
end
for $i_1 \leftarrow 0$ **to** $k_i - 1$ **do**
  for $i_2 \leftarrow 0$ **to** $t_i - 1$ **do**
    $\mathrm{ct}_{\mathrm{b}} \leftarrow \mathrm{ct}_{\mathrm{b}} \oplus \mathsf{Rot}(\mathrm{ct}''_{\mathrm{a}}; k_i^2h_iw_ii_2 + k_iw_ii_1 - k_i(k_ii_2 + i_1)) \odot \bar{\mathbf{s}}'^{(k_ii_2+i_1)}$
  end
end
return ct_{b}

5 OPTIMIZATION OF LEVEL CONSUMPTION

In this paper, we use bootstrapping and approximate ReLU function that work for input values in $[-1, 1]$. However, there are many values with an absolute value greater than one in the deep learning process. Thus, it is required to do scaling before and after the bootstrapping/approximate ReLU function. We set sufficiently large B so that all real values used in the execution of ResNet model fall within $[-B, B]$. We set $B = 40$ and $B = 65$ for the CIFAR-10 and CIFAR-100 datasets, respectively, and each value of B is obtained by adding some margin to the maximum value of all used values.

In this paper, convolution, batch normalization, bootstrapping, and approximate ReLU function are repeatedly performed in this

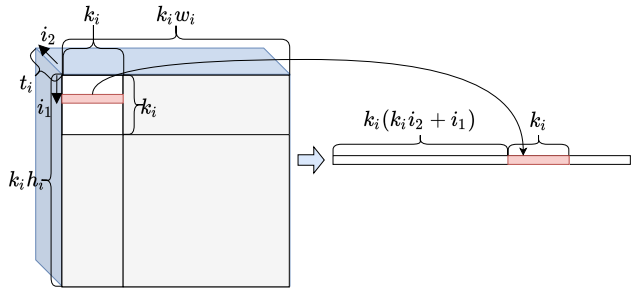


Figure 5: Rearranging process that selects and places $k_i^2 t_i$ valid values sequentially in AvgPool algorithm.

order. Since bootstrapping and approximate ReLU deal with values scaled by $1/B$, it is required to multiply them by B before convolution and by $1/B$ after batch normalization. In addition, batch normalization requires multiplication by a constant vector $\mathbf{c}''$. Thus, scaling process and batch normalization require a total of three level consumption. However, it is important to reduce level consumption as much as possible because the security level decreases as the level consumption increases. Thus, we propose an algorithm PrlMultConvBN that integrates convolution and batch normalization while removing the need for three level consumptions by scaling process and batch normalization.

For a given input ciphertext ct_x, we can perform scaling processes, convolution, and batch normalization by evaluating $\mathsf{ct}_\mathsf{x} \odot (B \cdot \mathbf{1})$, $\mathtt{PrlMultConv}(\mathsf{ct}_\mathsf{x}, U)$, $\mathbf{c}'' \odot \mathsf{ct}_\mathsf{x} \oplus (\mathbf{i}'' - \mathbf{c}'' \cdot \mathbf{m}'')$, and $\mathsf{ct}_\mathsf{x} \odot (\frac{1}{B} \cdot \mathbf{1})$ functions sequentially, where $\mathbf{1}$ is all-one vector in $\mathbb{R}^n$. Considering PrlMultConv is a linear function, these operations are equivalent to evaluating

$$(\mathbf{c}'' \odot \mathtt{PrlMultConv}(\mathsf{ct}_\mathsf{x}, BU) \oplus (\mathbf{i}'' - \mathbf{c}'' \cdot \mathbf{m}'')) \odot (\frac{1}{B} \cdot \mathbf{1})$$
$$= \mathbf{c}'' \odot \mathtt{PrlMultConv}(\mathsf{ct}_\mathsf{x}, U) \oplus \frac{1}{B}(\mathbf{i}'' - \mathbf{c}'' \cdot \mathbf{m}'').$$

Here, if we perform $\mathtt{PrlMultConv}(\mathsf{ct}_\mathsf{x}, U)$ while replacing the original selecting tensor $\mathsf{Vec}(S'^{(i)})$ by $\mathsf{PrlBNPack}(C) \cdot \mathsf{Vec}(S'^{(i)})$, we can perform $\mathbf{c}'' \odot \mathtt{PrlMultConv}(\mathsf{ct}_\mathsf{x}, U)$ without additional level consumption. In addition, computation of $\frac{1}{B}(\mathbf{i}'' - \mathbf{c}'' \cdot \mathbf{m}'')$ requires no additional level consumption since it simply requires operations for plaintext vectors. Thus, we can perform scaling processes, convolution, and batch normalization with only two level consumptions. Algorithm 7 describes the proposed convolution/batch normalization integration algorithm that uses level optimization technique.

Algorithm 7: $\mathtt{PrlMultConvBN}(\mathsf{ct}''_\mathsf{a}, U, C, M, I)$

Input: Parallelly multiplexed tensor ciphertext ct''_a, weight tensor U, and batch normalization constant vectors C, M, I

Output: Parallelly multiplexed tensor ciphertext ct''_d

$\mathsf{ct}''_\mathsf{d} \leftarrow \mathsf{ct}_\mathsf{zero}$
for $i_1 \leftarrow 0$ **to** $f_h - 1$ **do**
  for $i_2 \leftarrow 0$ **to** $f_w - 1$ **do**
    $\mathsf{ct}''^{(i_1,i_2)} \leftarrow \mathsf{Rot}(\mathsf{ct}''_\mathsf{a}; k_i^2 w_i(i_1 - (f_h-1)/2) + k_i(i_2 - (f_w-1)/2))$
  end
end
for $i_3 \leftarrow 0$ **to** $q - 1$ **do**
  $\mathsf{ct}''_\mathsf{b} \leftarrow \mathsf{ct}_\mathsf{zero}$
  for $i_1 \leftarrow 0$ **to** $f_h - 1$ **do**
    for $i_2 \leftarrow 0$ **to** $f_w - 1$ **do**
      $\mathsf{ct}''_\mathsf{b} \leftarrow \mathsf{ct}''_\mathsf{b} \oplus \mathsf{ct}''^{(i_1,i_2)} \odot \mathsf{PrlMultWgtPack}(U; i_1, i_2, i_3)$
    end
  end
  $\mathsf{ct}''_\mathsf{c} \leftarrow \mathtt{SumSlots}(\mathsf{ct}''_\mathsf{b}; k_i, 1)$
  $\mathsf{ct}''_\mathsf{c} \leftarrow \mathtt{SumSlots}(\mathsf{ct}''_\mathsf{c}; k_i, k_i w_i)$
  $\mathsf{ct}''_\mathsf{c} \leftarrow \mathtt{SumSlots}(\mathsf{ct}''_\mathsf{c}; t_i, k_i^2 h_i w_i)$
  for $i_4 \leftarrow 0$ **to** $\min(p_i - 1, c_o - 1 - p_i i_3)$ **do**
    $i \leftarrow p_i i_3 + i_4$
    $\mathsf{ct}''_\mathsf{d} \leftarrow \mathsf{ct}''_\mathsf{d} \oplus \mathsf{Rot}(\mathsf{ct}''_\mathsf{c}; -\lfloor i/k_o^2 \rfloor k_o^2 h_o w_o + \lfloor n_t/p_i \rfloor (i \bmod p_i) - \lfloor (i \bmod k_o^2)/k_o \rfloor k_o w_o - i \bmod k_o) \odot (\mathsf{PrlBNPack}(C) \cdot \mathsf{Vec}(S'^{(i)}))$
  end
end
for $j \leftarrow 0$ **to** $\log_2 p_o - 1$ **do**
  $\mathsf{ct}''_\mathsf{d} \leftarrow \mathsf{ct}''_\mathsf{d} \oplus \mathsf{Rot}(\mathsf{ct}''_\mathsf{d}; -2^j (n_t/p_o))$
end
$\mathsf{ct}''_\mathsf{d} \leftarrow \mathsf{ct}''_\mathsf{d} \ominus \frac{1}{B}(\mathbf{c}'' \cdot \mathbf{m}'' - \mathbf{i}'')$
return ct''_d

6 RESNET ON THE RNS-CKKS SCHEME

6.1 Parameter Setting

We set polynomial degree N to 2^{16}. Then, the number of full slots is $n_t = 2^{15}$. We optimize some parameters used in [16] to achieve higher security level. First, we set the Hamming weight of secret key to 192, which is larger than 64 used in [16] because larger Hamming weight of secret key increases available modulus bits. In addition, we set base modulus, special modulus, and bootstrapping modulus to 51-bit prime instead of 60-bit prime, and we set default modulus to 46-bit prime instead of 50-bit prime. Even if the length of the modulus bits is reduced, high accuracy of bootstrapping or approximate ReLU function can be achieved, which will be explained in detail in Section 6.2. Based on the hybrid dual attack for the LWE with the sparse secret key [4], the total modulus bit length that can be used with 128-bit security is 1553-bit.

6.2 Bootstrapping

We use the bootstrapping for sparsely packed ciphertext with $n = 2^{14}, 2^{13}$, and 2^{12} since data in each input ciphertext for the bootstrapping is less than $n_t = 2^{15}$. CoeffToSlot and SlotToCoeff procedures are performed with level collapsing technique with three levels. The degrees of the approximate polynomials for the cosine function and the inverse sine function are 59, 1, respectively, and the number of double-angle formula is two. The total level consumption is 14 in the bootstrapping, and the total modulus

consumption is 644. We refer to the sparse slot bootstrapping for $n = 2^{14}, 2^{13}$, and 2^{12} as `Boot14`, `Boot13`, `Boot12`, respectively.

6.3 Approximate Homomorphic ReLU Algorithm

We use the approximate homomorphic ReLU algorithm that uses approximate polynomial for the ReLU function using composition of minimax approximate polynomial as in [14, 15]. We use precision parameter $\alpha = 13$, margin $\eta = 2^{-15}$, max function factor $\zeta = 16$, set of degrees $\{15, 15, 27\}$, and set of scaling values $\{1, 2, 1.7\}$. The approximate homomorphic ReLU algorithm takes a ciphertext ct_x of $\mathbf{x} \in \mathbb{R}^n$ as input and outputs another ciphertext ct_y of $\mathbf{y} \in \mathbb{R}^n$. We refer to this algorithm for the parameters that we chose as $\mathtt{AppReLU}(\mathsf{ct}_x)$. $\mathtt{AppReLU}(\mathsf{ct}_x)$ satisfies $|\mathbf{y} - \mathrm{ReLU}(\mathbf{x})| \leq 2^{-\alpha}$ for $\mathbf{x} = (x_0, x_1, \cdots, x_{n-1}) \in \mathbb{R}^n$ such that $x_i \in [-1, 1]$, $0 \leq i < n$. All inputs of `AppReLU` should be scaled by $1/B$ so that all components of input vectors of `AppReLU` are within $[-1, 1]$.

Since `AppReLU` is quite precise, we can use pre-trained parameters for standard ResNet network. That is, there is no need to retrain for a specialized network that has approximate ReLU functions instead of exact ReLU functions.

6.4 Structure of the Proposed ResNet Models on the RNS-CKKS Scheme

We classify 32×32 CIFAR-10 or CIFAR-100 images. We implement ResNet-20/32/44/56/110 models on the RNS-CKKS scheme using `PrlMultConvBN`, `AppReLU`, `Boot`, `AvgPool`, and fully connected layer. For classification of CIFAR-10 dataset, we also uses `Downsamp`. From now on, we simply refer to the proposed convolution/batch normalization integration algorithm `PrlMultConvBN` as `ConvBN`. We implement fully connected layer using the diagonal method in [12]. Figure 6 shows the proposed ResNet structure on the RNS-CKKS scheme, and Table 2 presents the parameters that are used in each `ConvBN` or `Downsamp` process.

While two bootstrappings are required to perform approximate ReLU function, convolution, and batch normalization once in [16], only single use of bootstrapping is required in this paper because we reduce the required level consumption for convolution, batch normalization, and bootstrapping a lot compared to [16].

7 SIMULATION RESULTS

In this section, numerical results of the proposed ResNet on the RNS-CKKS scheme are presented. The numerical analyses are conducted on the representative RNS-CKKS scheme library SEAL [26] on AMD Ryzen Threadripper PRO 3995WX at 2.096 GHz (64 cores) with 512 GB RAM, running the Ubuntu 20.04 operating system. We employ the CIFAR-10 and CIFAR-100 datasets for evaluation, which are both composed of 50,000 images for training and 10,000 images for testing [13]. We use pre-trained parameters for standard ResNet-20/32/44/56/110 networks.

7.1 ResNet for the CIFAR-10 dataset

We perform ResNet-20/32/44/56/110 using the proposed parallelly multiplexed packing method on the RNS-CKKS scheme. We have

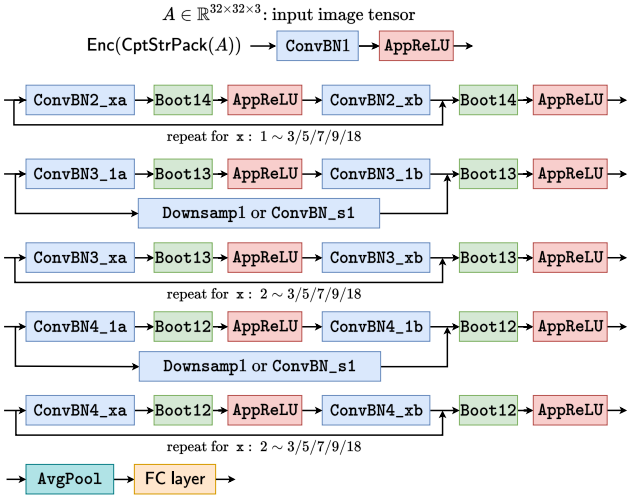


Figure 6: Structure of the proposed ResNet-20/32/44/56/110 on the RNS-CKKS scheme.

component		f_h	f_w	s	h_i	h_o	w_i	w_o	c_i	c_o	k_i	k_o	t_i	t_o	p_i	p_o	q
ConvBN1		3	3	1	32	32	32	32	3	16	1	1	3	16	8	2	2
ConvBN2_xa		3	3	1	32	32	32	32	16	16	1	1	16	16	2	2	8
ConvBN2_xb		3	3	1	32	32	32	32	16	16	1	1	16	16	2	2	8
ConvBN3_xa	x = 1	3	3	2	32	16	32	16	16	32	1	2	16	8	2	4	16
	otherwise	3	3	1	16	16	16	16	32	32	2	2	8	8	4	4	8
ConvBN3_xb		3	3	1	16	16	16	16	32	32	2	2	8	8	4	4	8
ConvBN4_xa	x = 1	3	3	2	16	8	16	8	32	64	2	4	8	4	4	8	16
	otherwise	3	3	1	8	8	8	8	64	64	4	4	4	4	8	8	8
ConvBN4_xb		3	3	1	8	8	8	8	64	64	4	4	4	4	8	8	8
ConvBN_s1		1	1	2	32	16	32	16	16	32	1	2	16	8	2	4	16
ConvBN_s2		1	1	2	16	8	16	8	32	64	2	4	8	4	4	8	16
Downsamp1		-	-	-	32	16	32	16	16	32	1	2	16	8	2	4	-
Downsamp2		-	-	-	16	8	16	8	32	64	2	4	8	4	4	8	-

Table 2: Parameters that are used in each `ConvBN` or `Downsamp` process

	Lee et al.[16]	proposed	ratio
number of key-switchings	375,920	3,489	107.7
sum of squared level	5.66×10^7	1.10×10^6	51.68

Table 3: Comparison of the number of key-switching operations and the sum of level squared for key-switching operation

significantly reduced the number of key-switching operations compared to that in [16], where the key-switching operation is by far the heaviest operation in FHE. Table 3 shows the total number of the key-switching operations of the previous work and our proposed work. We reduce the number of key-switching operations to 1/107.7 compared to Lee et al.'s algorithm. Lee et al. [16] suggested

component	Lee et al. [16] (64 threads)		proposed method (single thread)									
	ResNet-20		ResNet-20		ResNet-32		ResNet-44		ResNet-56		ResNet-110	
	runtime	percent	runtime	percent	runtime	percent	runtime	percent	runtime	percent	runtime	percent
ConvBN	-	-	346s	15.2%	547s	14.7%	751s	14.3%	960s	14%	1,855s	14%
AppReLU	-	-	257s	11.3%	406s	10.9%	583s	11.2%	762s	11.1%	1,475s	11.1%
Boot	-	-	1,651s	72.6%	2,760s	74.0%	3,874s	74.1%	5,113s	74.6%	9,936s	74.8%
Downsamp	-	-	5s	0.2%	5s	0.1%	5s	0.09%	5s	0.07%	5s	0.04%
AvgPool	-	-	2s	0.1%	2s	0.06%	2s	0.05%	2s	0.04%	2s	0.02%
FC layer	-	-	10s	0.4%	10s	0.3%	10s	0.2%	10s	0.1%	10s	0.08%
total	10,602s	100%	2,271s	100%	3,730s	100%	5,224s	100%	6,852s	100%	13,282s	100%

Table 4: Classification runtime for one CIFAR-10 image using ResNet models on the RNS-CKKS scheme

	model	runtime	amortized runtime
Lee et al. [16] (one image, 128 threads)	ResNet-20	10,602s	10,602s
proposed method (50 images, 50 threads)	ResNet-20	3,973s	79s
	ResNet-32	6,130s	122s
	ResNet-44	8,983s	179s
	ResNet-56	11,303s	226s
	ResNet-110	22,778s	455s

Table 5: Classification (amortized) runtime for multiple CIFAR-10 images using ResNet models on the RNS-CKKS scheme

model	number of test images	number of success	backbone accuracy	obtained accuracy
ResNet-20	10,000	9,132	91.52%	91.32%
ResNet-32	10,000	9,240	92.49%	92.4%
ResNet-44	2,000	1,852	92.76%	92.6%
ResNet-56	2,000	1,856	93.27%	92.8%
ResNet-110	2,000	1,858	93.5%	92.9%

Table 6: Classification accuracy for CIFAR-10 images using ResNet models on the RNS-CKKS scheme

that the sum of squared level of the input ciphertext for each key-switching operations can represent the total amount of operations more exactly. If we use this criterion, this value is reduced to 1/51.68 compared to Lee et al.'s algorithm.

Table 4 shows the classification runtime for one CIFAR-10 image using ResNet models on the RNS-CKKS scheme. Due to the large reduction in the number of key switchings, while the previous implementation in [16] takes 10,602s with maximum 64 threads to perform ResNet-20 on the RNS-CKKS scheme, the proposed implementation takes 2,271s to perform ResNet-20 even with single thread, which is about five times faster. Furthermore, we succeed in implementing ResNet-32/44/56/110 on FHE for the first time. It can be seen from Table 4 that the runtime increases almost linearly according to the number of layers.

Since servers should perform classification of multiple images of clients in many cases, not only the runtime of classification for one image, but also the amortized runtime for multiple images, i.e., runtime per image, is important. Since the proposed implementation requires only single thread, multiple threads allow us to classify multiple images simultaneously. This is impossible in [16] because they used all 64 threads to classify one image. Table 5 shows the runtime and amortized runtime of classification for multiple CIFAR-10 images using ResNet models on the RNS-CKKS scheme. The proposed implementation of ResNet-20 takes 3973s to classify 50 images using 50 threads, which corresponds to amortized runtime 79s. This is 134 times faster than the amortized runtime 10,602s in [16].

Table 6 presents the classification accuracy for CIFAR-10 images using ResNet models on the RNS-CKKS scheme. Due to our significantly reduced amortized runtime, we succeed to perform classification for all 10,000 test images for ResNet-20 while only 383 images were tested in [16]. We have 9,132 success out of 10,000 test images, that is, 91.32% accuracy, which is very close to the accuracy of backbone network 91.52%. We also obtain accuracies for ResNet-32/44/56/110, deeper neural networks than ResNet-20, on the RNS-CKKS for the first time. It can be seen from Table 6 that classification accuracies on the RNS-CKKS scheme for deeper neural networks than ResNet-20 are also close to the accuracies of backbone networks.

7.2 ResNet for the CIFAR-100 Dataset

We classify the CIFAR-100 images with ResNet-32 on the RNS-CKKS scheme using the proposed parallelly multiplexed packing method. Table 7 shows the classification runtime for ResNet-32. Table 8 presents the classification accuracy for CIFAR-100 images. We have 6,943 success out of 10,000 test images, that is, 69.43%. This is very close to the accuracy of backbone network 69.5%.

7.3 Discussion

7.3.1 Runtime. First, we reduced the runtime for classification of one image to 1/4.67 compared to that in [16] using the proposed parallelly multiplexed packing method. It should be noted that the runtime of the proposed implementation can be much more improved using GPU or hardware accelerators because we used only single thread unlike that in [16]. For example, according to [11], operations of RNS-CKKS scheme on GPU can be more than 100 times

component	one image single thread		50 images 50 threads	
	runtime	percent	runtime	amortized runtime
ConvBN	542s	13.7%	-	-
AppReLU	510s	12.9%	-	-
Boot	2,864s	72.7%	-	-
AvgPool	2s	0.05%	-	-
FC layer	24s	0.6%	-	-
total	3,942s	100%	6,351s	127s

Table 7: Classification runtime for CIFAR-100 images using ResNet models on the RNS-CKKS scheme

model	number of test images	number of success	backbone accuracy	obtained accuracy
ResNet-32	10,000	6,943	69.5%	69.43%

Table 8: Classification accuracy for CIFAR-100 images using ResNet-32 on the RNS-CKKS scheme

faster than those on CPU single thread. Thus, the implementation of ResNet using the proposed method is expected to reduce the runtime to more than 1/100 on GPU, leading to practical runtimes.

In addition, our amortized runtime for ResNet-20 is 134 times faster than that in [16], which allows the server to classify multiple images from multiple clients much faster.

7.3.2 Deeper Neural Network. For the first time, we showed results of deeper neural networks than ResNet-20 on FHE. The proposed implementation showed that the runtime increases linearly in the number of layers, which is generally difficult to be expected in leveled HE.

7.3.3 Classification Accuracy. Although the accuracy for ResNet-20 on the RNS-CKKS scheme was successfully obtained on [16], 383 test images are not sufficiently large enough to guarantee that ResNet on the RNS-CKKS scheme has also good accuracy for all 10,000 test images. Since we reduced amortized runtime much compared to that in [16], we were able to perform ResNet-20 for all 10,000 test images, showing that ResNet-20 on the RNS-CKKS scheme has very close accuracy to the accuracy of the backbone network. In addition, we also showed that the accuracy on the RNS-CKKS scheme is similar to that of backbone network for ResNet-32/44/56/110, which is deeper than ResNet-20.

It is noteworthy that our obtained accuracy is not hard limit. There is a tradeoff between the runtime and precisions of bootstrapping/approximate ReLU, and increasing the precisions of bootstrapping/approximate ReLU is expected to make the obtained accuracies on the RNS-CKKS scheme more closer to those of backbone networks. Further, since we succeeded in implementing the standard deep neural network rather than HE-friendly network, we can also expect to achieve much higher classification accuracy using the state-of-the-art deep neural networks.

7.3.4 Security. While the work in [16] satisfied only 111 bit security due to many level consumptions, the proposed implementation achieved the standard 128 bit security because we reduced required level consumption a lot.

8 CONCLUSIONS

In this paper, we studied an efficient implementation method of deep neural network on the RNS-CKKS FHE scheme. First, we proposed a multiplexed packing method that makes tensors for multiple channels to be multiplexed into one tensor. Then, we proposed a multiplexed convolution method using this packing method, which significantly reduced the number of key-switching operations in the implementation of ResNet network on the RNS-CKKS scheme. In addition, we proposed a parallelly multiplexed convolution, which is faster than multiplexed convolution by utilizing operation of full slots. Further, we fine tuned the parameters to reach the standard 128-bit security level and to further reduce the number of the bootstrapping operations. Our implementation of ResNet-20 model with the proposed techniques took about 37 minutes (2,271s) with only one thread. As for the total amount of the operations, the number of the key-switching operations, which is the heaviest operation in the RNS-CKKS scheme, was reduced to 1/107 compared to the previous implementation. Further, we even implemented ResNet-32/44/56/110 models on RNS-CKKS FHE scheme with the linear time of the number of layers, which is the first implementation result for these standard deep neural networks. Finally, we successfully classified the CIFAR-100 images on RNS-CKKS scheme using standard ResNet-32 model, and we obtained a running time of 3,942s and an accuracy of 69.4% close to the accuracy of backbone network 69.5%.

ACKNOWLEDGMENTS

This work was supported by the BK21 FOUR program of the Education and Research Program for Future ICT Pioneers, Seoul National University in 2021.

REFERENCES

[1] Jean-Philippe Bossuat, Christian Mouchet, Juan Troncoso-Pastoriza, and Jean-Pierre Hubaux. 2021. Efficient bootstrapping for approximate homomorphic encryption with non-sparse keys. In *International Conference on the Theory and Applications of Cryptographic Techniques*.

[2] Hao Chen, Miran Kim, Ilya Razenshteyn, Dragos Rotaru, Yongsoo Song, and Sameer Wagh. 2020. Maliciously secure matrix multiplication with applications to private deep learning. In *International Conference on the Theory and Application of Cryptology and Information Security*. Springer, 31–59.

[3] Jung Hee Cheon, Kyoohyung Han, Andrey Kim, Miran Kim, and Yongsoo Song. 2018. Bootstrapping for approximate homomorphic encryption. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 360–384.

[4] Jung Hee Cheon, Minki Hhan, Seungwan Hong, and Yongha Son. 2019. A hybrid of dual and meet-in-the-middle attack on sparse and ternary secret LWE. *IEEE Access* 7 (2019), 89497–89506.

[5] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. 2020. TFHE: fast fully homomorphic encryption over the torus. *Journal of Cryptology* 33, 1 (2020), 34–91.

[6] Ilaria Chillotti, Marc Joye, and Pascal Paillier. 2021. Programmable bootstrapping enables efficient homomorphic inference of deep neural networks. In *Cyber Security Cryptography and Machine Learning - 5th International Symposium, CSCML 2021, Be'er Sheva, Israel, July 8-9, 2021, Proceedings (Lecture Notes in Computer Science)*, Shlomi Dolev, Oded Margalit, Benny Pinkas, and Alexander A. Schwarzmann (Eds.), Vol. 12716. Springer, 1–19.

[7] Roshan Dathathri, Olli Saarikivi, Hao Chen, Kim Laine, Kristin Lauter, Saeed Maleki, Madanlal Musuvathi, and Todd Mytkowicz. 2019. CHET: An optimizing

compiler for fully-homomorphic neural-network inferencing. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 142–156.

[8] Ran Gilad-Bachrach, Nathan Dowlin, Kim Laine, Kristin Lauter, Michael Naehrig, and John Wernsing. 2016. Cryptonets: Applying neural networks to encrypted data with high throughput and accuracy. In *Proceedings of International Conference on Machine Learning*. PMLR, 201–210.

[9] Sergey Ioffe and Christian Szegedy. 2015. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International conference on machine learning*. PMLR, 448–456.

[10] Xiaoqian Jiang, Miran Kim, Kristin Lauter, and Yongsoo Song. 2018. Secure outsourced matrix computation and application to neural networks. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. 1209–1222.

[11] Wonkyung Jung, Sangpyo Kim, Jung Ho Ahn, Jung Hee Cheon, and Younho Lee. 2021. Over 100x faster bootstrapping in fully homomorphic encryption through memory-centric optimization with GPUs. *Cryptol. ePrint Arch., Tech. Rep. 2021/508* (2021).

[12] Chiraag Juvekar, Vinod Vaikuntanathan, and Anantha Chandrakasan. 2018. GAZELLE: A low latency framework for secure neural network inference. In *Proceedings of 27th USENIX Security Symposium*. 1651–1669.

[13] Alex Krizhevsky, Geoffrey Hinton, et al. 2009. Learning multiple layers of features from tiny images. *CiteSeerX Technical Report, University of Toronto* (2009).

[14] Eunsang Lee, Joon-Woo Lee, Jong-Seon No, and Young-Sik Kim. 2021. Minimax approximation of sign function by composite polynomial for homomorphic comparison. *IEEE Transactions on Dependable and Secure Computing, accepted for publication* (2021).

[15] Junghyun Lee, Eunsang Lee, Joon-Woo Lee, Yongjune Kim, Young-Sik Kim, and Jong-Seon No. 2021. Precise approximation of convolutional neural networks for homomorphically encrypted data. *arXiv preprint arXiv:2105.10879* (2021).

[16] Joon-Woo Lee, HyungChul Kang, Yongwoo Lee, Woosuk Choi, Jieun Eom, Maxim Deryabin, Eunsang Lee, Junghyun Lee, Donghoon Yoo, Young-Sik Kim, et al. 2021. Privacy-preserving machine learning with fully homomorphic encryption for deep neural network. *arXiv preprint arXiv:2106.07229* (2021).

[17] Jian Liu, Mika Juuti, Yao Lu, and Nadarajah Asokan. 2017. Oblivious neural network predictions via minionn transformations. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 619–631.

[18] Qian Lou and Lei Jiang. 2019. SHE: A fast and accurate deep neural network for encrypted data. *Advances in Neural Information Processing Systems* 32 (2019), 10035–10043.

[19] Qian Lou and Lei Jiang. 2021. HEMET: A homomorphic-encryption-friendly privacy-preserving mobile neural network architecture. In *Proceedings of the 38th International Conference on Machine Learning*, Vol. 139. PMLR, 7102–7110.

[20] Pratyush Mishra, Ryan Lehmkuhl, Akshayaram Srinivasan, Wenting Zheng, and Raluca Ada Popa. 2020. Delphi: A cryptographic inference service for neural networks. In *29th {USENIX} Security Symposium ({USENIX} Security 20)*. 2505–2522.

[21] Nicolas Papernot, Patrick McDaniel, Arunesh Sinha, and Michael P. Wellman. 2018. Sok: Security and privacy in machine learning. In *2018 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 399–414.

[22] Brandon Reagen, Woo-Seok Choi, Yeongil Ko, Vincent T Lee, Hsien-Hsin S Lee, Gu-Yeon Wei, and David Brooks. 2021. Cheetah: Optimizing and accelerating homomorphic encryption for private inference. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 26–39.

[23] M Sadegh Riazi, Kim Laine, Blake Pelton, and Wei Dai. 2020. Heax: An architecture for computing on encrypted data. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. 1295–1309.

[24] M Sadegh Riazi, Mohammad Samragh, Hao Chen, Kim Laine, Kristin Lauter, and Farinaz Koushanfar. 2019. {XONN}: Xnor-based oblivious deep neural network inference. In *28th {USENIX} Security Symposium ({USENIX} Security 19)*. 1501–1518.

[25] Bita Darvish Rouhani, M Sadegh Riazi, and Farinaz Koushanfar. 2018. Deepsecure: Scalable provably-secure deep learning. In *Proceedings of the 55th Annual Design Automation Conference*. 1–6.

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