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

Eunsang Lee [1] Joon-Woo Lee [1] Junghyun Lee [1] Young-Sik Kim [2] Yongjune Kim [3] Jong-Seon No [1]
Woosuk Choi [4]

Abstract

Recently, the standard ResNet-20 network was successfully implemented on residue number system variant Cheon-Kim-Kim-Song (RNS-CKKS) scheme using bootstrapping, but the implementation lacks practicality due to high latency and low security level. To improve the performance, we first minimize total bootstrapping runtime using *multiplexed parallel convolution* that collects sparse output data for multiple channels compactly. We also propose the *imaginary-removing bootstrapping* to prevent the deep neural networks from catastrophic divergence during approximate ReLU operations. In addition, we optimize level consumptions and use lighter and tighter parameters. Simulation results show that we have 4.67× lower inference latency and 134× less amortized runtime (runtime per image) for ResNet-20 compared to the state-of-the-art previous work, and we achieve standard 128-bit security. Furthermore, we successfully implement ResNet-110 with high accuracy on the RNS-CKKS scheme for the first time.

1. Introduction

The clients would be reluctant to send their sensitive private data to the server, such as medical information. To protect clients' privacy, privacy-preserving machine learnings (PPMLs) have been studied to perform inferences directly on encrypted data. Most previous PPMLs adopt the nonstandard convolutional neural networks (CNNs) that reduce the number of layers or replace activation functions with low-degree polynomials (Gilad-Bachrach et al., 2016; Dathathri et al., 2019; Boemer et al., 2019; Chou et al., 2018; Lou & Jiang, 2021; Juvekar et al., 2018; Mishra et al., 2020; Park et al., 2022). This approach requires the training stage for the newly designed CNNs. However, since training is a costly process and even access to training datasets is often restricted due to data privacy issues, the request for training is a burden on the server in many real-world applications. Furthermore, it is not easy to design nonstandard CNNs for large datasets such as ImageNet. Thus, PPML based on the standard CNNs (SCNNs) using already given pre-trained parameters is also practically important.

Although relatively simple datasets can be classified using shallow SCNNs, very deep SCNNs (VDSCNNs) are required for the more difficult datasets (Simonyan & Zisserman, 2015; Szegedy et al., 2015). Thus, it is an important and appealing goal to implement practical PPML for VDSCNNs, which is the main focus of our paper. In particular, the ResNet model (He et al., 2016) is a popular SCNN because it handles the gradient vanishing problem effectively. Thus, it will be very meaningful to implement practical PPML for very deep ResNet models (e.g., ResNet-110).

Interactive PPML is an approach that uses both homomorphic encryption (HE) and secure multi-party computation (MPC) (Juvekar et al., 2018; Mishra et al., 2020; Park et al., 2022). The interactive PPML's inference for the SCNNs requires a huge amount of data communication between the server and the client. For example, one ReLU function requires data communication of 19KB according to (Mishra et al., 2020), hence, data communication of several gigabytes is required to classify only one encrypted CIFAR-10 image using the standard ResNet-20 or ResNet-32.

Thus, we focus on *non-interactive PPML for VDSCNNs*, which performs VDSCNNs on the encrypted data using only fully homomorphic encryption (FHE) without MPC. Since available level consumption (i.e. depth of arithmetic circuit) is limited in leveled HE, previous PPMLs that adopt leveled HE only supported a small number of layers. To achieve non-interactive PPML for VDSCNNs, it is essential to use bootstrapping operations that can arbitrarily increase available level consumption. Recently, an SCNN for the encrypted CIFAR-10 images was implemented with high

[1]Dept. of Electrical and Computer Engineering, INMC, Seoul National University [2]Dept. of Information and Communication Engineering, Chosun University [3]Dept. of Electrical Engineering and Computer Science, DGIST [4]Samsung Advanced Institute of Technology. Correspondence to: Joon-Woo Lee <joonwoo42@snu.ac.kr>.

accuracy using bootstrapping for the first time in (Lee et al., 2021c). Specifically, the standard ResNet-20 was realized on residue number system variant Cheon-Kim-Kim-Song (RNS-CKKS) scheme (Cheon et al., 2018a) using bootstrapping after replacing all ReLU functions with approximate polynomials for ReLU functions (APRs). By adopting precise polynomial approximation, the pre-trained parameters over the plaintext data for the SCNNs can be used without retraining. This paper also focuses on this approach, which is useful in one of the following cases:

- Inference (classification) using pre-trained parameters (training/retraining is limited);
- Difficult classification tasks (e.g., ImageNet dataset) that cannot be handled by HE-friendly nonstandard CNNs.

Although this approach requires bootstrapping that consumes a relatively long runtime, this approach can be used practically because the runtime of bootstrapping has been significantly reduced thanks to recent advances in bootstrapping algorithms (Lee et al., 2021a;d; Bossuat et al., 2021; 2022) and acceleration of bootstrapping using GPU or hardware accelerators (Jung et al., 2021; Kim et al., 2021b).

Even though the implementation of (Lee et al., 2021c) is an important step toward VDSCNNs, deeper CNNs than ResNet-20 have not yet been implemented. In addition, the implementation has high latency of 10,602s even with 64 CPU threads. One major reason for this high latency comes from its inefficient data packing. In (Lee et al., 2021c), data for only one channel is packed into the ciphertext, which requires bootstrappings as many as the number of channels. Furthermore, the number of bootstrappings is even doubled due to non-optimized parameters and level consumption. The usefulness of this implementation is further reduced due to the security level lower than 128 bits.

Our Contributions We implement practical PPML for VDSCNNs for the first time by resolving the problems of (Lee et al., 2021c). Our contributions are summarized as follows:

- We effectively reduce the bootstrapping runtime by using a multiplexed packing method (i.e., packing data of multiple channels into one ciphertext in a compact manner);
- We propose a *multiplexed convolution* algorithm that performs convolutions for multiplexed input tensors, which also supports strided convolutions. We also propose a faster *multiplexed parallel convolution* algorithm, which reduces the number of required rotations in the multiplexed convolution algorithm by 62% by utilizing full slots of ciphertext;
- We find that a catastrophic divergence phenomenon occurs when implementing VDSCNNs using APRs. We propose *imaginary-removing bootstrapping* that prevents this phenomenon so as to maintain the accuracy of PPML for VDSCNNs;
- We optimize level consumption and use lighter and tighter parameters to achieve faster inference and the standard 128-bit security level;
- We implement ResNet-20 on the RNS-CKKS scheme using the SEAL library (SEAL) with a latency of 3,972s with only one CPU thread, which is $4.67\times$ lower than that of (Lee et al., 2021c) using 64 threads. Also, our amortized runtime (runtime per image) is $134\times$ smaller due to a significant reduction of the number of operations;
- We also implement ResNet-32/44/56/110 on the RNS-CKKS scheme with high accuracies close to those of backbone CNNs.

2. Preliminaries

2.1. RNS-CKKS Fully Homomorphic Encryption

RNS-CKKS is an FHE scheme that supports fixed-point arithmetic operations on encrypted data. The ciphertext in the RNS-CKKS scheme is in the form of $(b, a) \in R_Q^2$ for some product of prime numbers Q, where $R_Q = \mathbb{Z}_Q[X]/\langle X^N + 1\rangle$. $N/2$ real (or complex) numbers are encrypted in $N/2$ slots of a single ciphertext and we denote $N/2$ as n_t. Homomorphic operations perform the same operation on each slot simultaneously. If we simply denote the ciphertext of a vector $\mathbf{u} \in \mathbb{R}^{n_t}$ as $[\mathbf{u}]$, homomorphic addition, scalar multiplication, and rotation in the RNS-CKKS scheme can be described as follows:

- $[\mathbf{u}] \oplus [\mathbf{v}] = [\mathbf{u} + \mathbf{v}]$
- $[\mathbf{u}] \odot \mathbf{v} = \mathbf{u} \odot [\mathbf{v}] = [\mathbf{u} \cdot \mathbf{v}]$
- $\mathsf{Rot}([\mathbf{u}]; r) = [\langle \mathbf{u} \rangle_r]$,

where $\mathbf{u}\cdot\mathbf{v}$ denotes component-wise multiplication and $\langle \mathbf{u} \rangle_r$ denotes the cyclically shifted vector of $\mathbf{u}$ by r to the left.

Each ciphertext has a non-negative integer ℓ, called level, which means the capacity for homomorphic multiplication operations. After each homomorphic multiplication, the level is decreased by one through the rescaling procedure. If the level ℓ becomes zero after several multiplications, it is required to perform *bootstrapping* that makes this zero-level ciphertext to a higher-level ciphertext to enable further homomorphic multiplications.

If a message is a vector $\mathbf{v}$ with size n that is less than n_t, we can encrypt this message in a sparsely packed ciphertext (Cheon et al., 2018b). We refer to this packing method as *repetitive slot* (RS) packing. Specifically, after obtaining concatenated vector $(\mathbf{v}|\mathbf{v}|\cdots|\mathbf{v}) \in \mathbb{R}^{n_t}$ from $\mathbf{v}$, we encrypt this concatenated vector in ciphertext full slots. The bootstrapping of a ciphertext in which the message is packed

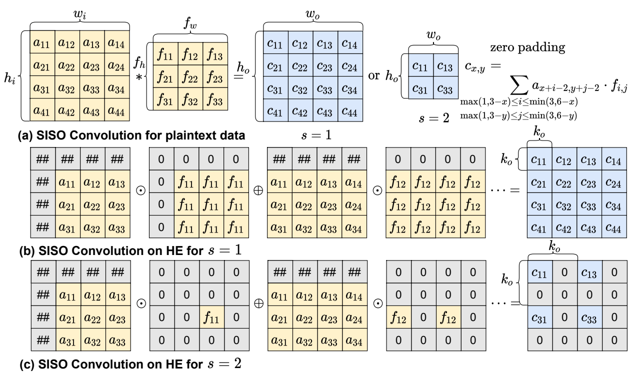

Figure 1: SISO Convolution on HE (Juvekar et al., 2018; Lee et al., 2021c).

using the RS packing, called *RS bootstrapping*, has less runtime than otherwise (Cheon et al., 2018b).

Key-switching operation (KSO) is an operation that switches the secret key for a ciphertext to a new secret key without any change of the message. Since KSO is the most time-consuming operation in the RNS-CKKS scheme, the number of KSOs roughly determines the total amount of operations in homomorphic arithmetic circuits. To reduce the number of KSOs in ResNet inference, it is desirable to reduce the number of rotations as much as possible since rotation requires a KSO and is used a lot in bootstrappings and convolutions.

2.2. Convolution on Homomorphic Encryption

The input of convolution is a three-dimensional tensor $A \in \mathbb{R}^{h_i \times w_i \times c_i}$, where h_i and w_i are the height and width of the input tensor, respectively, and c_i is the number of input channels. The output is the tensor $A' \in \mathbb{R}^{h_o \times w_o \times c_o}$, where h_o and w_o are the height and width of the output tensor, respectively, and c_o is the number of output channels. f_h and f_w are the 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. We only consider convolution using zero paddings.

Gazelle (Juvekar et al., 2018) proposed a method to perform single-input single-output (SISO) convolution ($c_i = c_o = 1$) on HE. In this method, each rotated input ciphertext is multiplied by some appropriate plaintext vector, and then the output is obtained by adding $f_h f_w$ multiplication results. We denote the result of SISO convolution for the j-th input channel and the i-th output channel by (i, j). The convolution result for the i-th output channel can be computed by using the equation $(i, 1) + (i, 2) + \cdots + (i, c_i)$, and Gazelle obtained all output data using diagonal grouping technique. Figure 1(b) shows how to perform SISO convolution on HE, where the data of $h_i \times w_i$ matrix is contained in a ciphertext or plaintext vector in a raster scan fashion. Although Gazelle also proposed a method to perform strided convolution on HE, this method that requires rearranging the data using re-encryption cannot be used in non-interactive PPML (i.e., PPML using only HE without MPC).

The ResNet-20 with strided convolutions was implemented using the method in Figure 1(c) (Lee et al., 2021c). The inference of CNNs with strided convolutions on FHE causes a *gap* between valid data in ciphertext slots. We denote the gaps of input and output ciphertext as k_i and k_o, respectively, where we have $k_o = sk_i$. In the entire ResNet, the value of k_i is one for the first layer and increases by a factor of s after each strided convolution. In Figure 1(c), the $h_o \times w_o$ plaintext output data is sparsely packed in a $k_o h_o \times k_o w_o$ matrix for the output gap k_o, and the other slots are filled with zero. We refer to this data packing method as *gap packing*. If we perform convolution for an input ciphertext whose data is packed by the gap packing, the amount to be shifted in Figure 1(b) should be increased by a factor of the input gap k_i.

In this paper, we often simply represent $k_i h_i \times k_i w_i$ matrix as a $k_i \times k_i$ matrix. The components of this simplified $k_i \times k_i$ matrix can be channel information, zero number, or ##, where ## implies arbitrary dummy data.

2.3. Threat Model

The threat model of this paper is similar to the previous PPMLs (Gilad-Bachrach et al., 2016; Lou & Jiang, 2021). The client sends the private data to the untrusted server after encryption using FHE. The server performs an inference directly on the encrypted data without decryption and sends back the ciphertext of the inference result. Only the client that holds the secret key can decrypt the inference result, guaranteeing data privacy from the server.

3. Comparison of Bootstrapping Runtime for Several Data Packing Methods

Since the most time-consuming component in the implementation of standard ResNet on the RNS-CKKS scheme is bootstrapping, it is desirable to reduce the bootstrapping runtime. The required number of KSOs and the runtime for bootstrapping according to the number of slots are presented in Table 1, where the runtime is obtained using the same parameters and simulation environments of Section 8. To reduce the total bootstrapping runtime, we should pack intermediate data into ciphertexts as compact as possible during the inference stage. In addition, the gap between valid data is increased by a factor of s after each strided convolution, leading to a reduction of packing density by a factor of s^2, but this low packing density should be resolved to effectively reduce the bootstrapping runtime.

boot $\log_2$(#slots)	10	11	12	13	14	15
#KSOs	63	70	77	84	91	94
runtime	72s	80s	86s	96s	112s	140s

Table 1: The number of KSOs and bootstrapping runtime according to various number of slots for bootstrapping

In this section, we attempt to remove this gap by packing these sparsely packed data in a compact manner. We first compare several data packing methods. We assume that the data of size less than n_t is packed in a ciphertext using RS packing so that RS bootstrapping can be used.

Gap packing Since gap packing in (Lee et al., 2021c) packs only one channel data into one ciphertext, the required number of bootstrapping operations will be the same as the number of channels. Thus, an unnecessarily large number of KSOs are required.

Gap packing with multiple channels We can improve gap packing by packing data of multiple channels into one ciphertext as much as possible. Although this packing can reduce the number of bootstrappings a lot, there are still many dummy slots as shown in Figure 1(c). For CNNs with many strided convolutions, the total bootstrapping runtime will increase exponentially with the number of strided convolutions.

Multiplexed packing Recently, HEAR (Kim et al., 2021a) used a new data packing method, referred to *multiplexed packing* herein. In this packing method, plaintext tensors of $h_i \times w_i$ size for k_i^2 channels are first mapped to one larger *multiplexed tensor* of size $k_i h_i \times k_i w_i$. Then, several multiplexed tensors are encrypted in one ciphertext. Although multiplexed packing was proposed to deal with the pooling of HE-friendly CNNs and speed up convolution in (Kim et al., 2021a), we repurpose it to reduce the bootstrapping runtime of CNNs with strided convolutions. Figure 2 describes multiplexed packing with $h_i = w_i = 4$ and $k_i = 2$. The formal description of multiplexed packing can be seen in Appendix E.

Figure 3 illustrates several packing methods for $k_i = 2$, where $c_n = \frac{n_t}{k_i^2 h_i w_i}$. Table 2 shows the required number of bootstrappings for implementation of ResNet-20 when each data packing method is used. The number of KSOs for total bootstrappings in ResNet-20 inference is also presented, and it is substantially reduced by multiplexed packing.

Thus, we require that the corresponding plaintext data be packed in the ciphertext using multiplexed packing during ResNet inference. Then, we should design a homomorphic convolution that takes an input ciphertext of multiplexed input tensor and outputs a ciphertext of multiplexed output tensor.

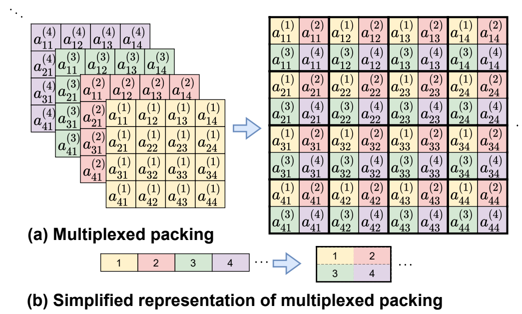

Figure 2: Multiplexed packing MultPack when $h_i = w_i = 4$ and $k_i = 2$.

$\log_2$(#slots)	before Str_conv1 (a)	(b)	(c)	before Str_conv2 (a)	(b)	(c)	after Str_conv2 (a)	(b)	(c)	total ResNet-20 (a)	(b)	(c)
10	16	0	0	32	0	0	64	0	0	672	0	0
11	0	0	0	0	0	0	0	0	0	0	0	0
12	0	0	0	0	0	0	0	0	1	0	0	6
13	0	0	0	0	0	1	0	0	0	0	0	6
14	0	1	1	0	0	0	0	0	0	0	6	6
15	0	0	0	0	1	0	0	2	0	0	18	0
total #KSOs										42,336	2,238	1,512

Table 2: The number of bootstrappings for implementation of ResNet-20 according to various data packing methods. (a), (b), and (c) imply gap packing, gap packing with multiple channels, and multiplexed packing, respectively. Str_conv1 and Str_conv2 denote the first and the second strided convolutions ($s = 2$) in ResNet-20, respectively.

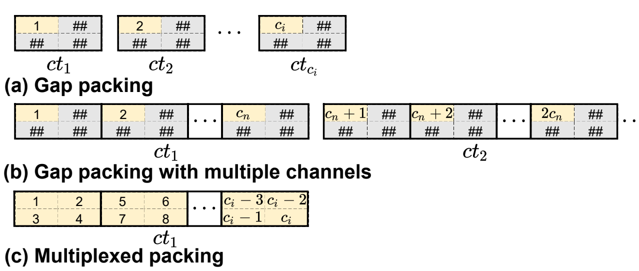

Figure 3: Comparison of several data packing methods.

4. Multiplexed Parallel Convolution on Fully Homomorphic Encryption

In this section, we propose homomorphic convolution algorithms that take a ciphertext having multiplexed input tensor and output a ciphertext having multiplexed output tensor. HEAR (Kim et al., 2021a) proposed such a convolution algorithm that supports stride one (i.e., $s = 1$). HEAR performs homomorphic convolution on multiple input channels simultaneously, adds SISO convolution results for all input channels, and collects only valid values by multiplying dummy slots by zero. We generalize this convolution algorithm to support the strided convolutions (i.e., $s \geq 2$). We propose to select and collect the valid values for the output gap $k_o = sk_i$ instead of the input gap k_i. Then, the output ciphertext has the plaintext output of strided convolution in the form of multiplexed tensor for $k_o = sk_i$. We denote this convolution algorithm as MULTCONV and Figure 4 describes the procedure of MULTCONV when $s = 2$.

Unlike previous works for HE-friendly networks not relying on bootstrapping, we require a large number of full slots, which is usually larger than the data size, to support bootstrapping and precise APRs. First, we consider packing data into ciphertext using RS packing so that RS bootstrapping can be used. Then, we note that one input channel is repeatedly used for SISO convolutions for multiple output channels. We propose a multiplexed parallel convolution algorithm, denoted as MULTPARCONV, that simultaneously performs SISO convolutions for multiple output channels, which consider the input packed by RS packing as just several independent inputs. This algorithm reduces the convolution runtime of MULTCONV while still compatible with RS bootstrapping. Figure 5 shows the procedure of MULTPARCONV using simplified representation of multiplexed packing.

The detailed algorithms of MULTCONV and MULTPARCONV are presented in Appendix F. Each execution of MULTCONV and MULTPARCONV requires $f_h f_w - 1 + c_o(2\lceil \log_2 k_i \rceil + \lceil \log_2 t_i \rceil + 1)$ and $f_h f_w - 1 + q(2\lceil \log_2 k_i \rceil + \lceil \log_2 t_i \rceil) + c_o + \log_2 p_o$ rotations, respectively, where $t_i = \lceil \frac{c_i}{k_i^2} \rceil$, $t_o = \lceil \frac{c_o}{k_o^2} \rceil$, $p_i = 2^{\lfloor \log_2 (\frac{n_t}{k_i^2 h_i w_i t_i}) \rfloor}$, $p_o = 2^{\lfloor \log_2 (\frac{n_t}{k_o^2 h_o w_o t_o}) \rfloor}$, and $q = \lceil \frac{c_o}{p_i} \rceil$. Then, the total required rotations for MULTCONV and MULTPARCONV in ResNet-20 inference are 4,360 and 1,657, respectively, which implies that MULTPARCONV requires 62% fewer rotations (i.e., number of KSOs) than MULTCONV.

5. Imaginary-Removing Bootstrapping

In this section, we propose an imaginary-removing bootstrapping, which makes it possible to implement VDSCNNs. The most sensitive component in ResNet implementation with many layers on the RNS-CKKS scheme is the APR. Since the RNS-CKKS scheme actually deals with complex numbers, precision noise during each homomorphic operation occurs not only in the real part of each data but also in the imaginary part. We find that the results of the APR in the real part can completely diverge if the accumulated noise in the imaginary part is not small enough.

We adopt the APR consisting of the composition of minimax approximate polynomials for piecewise sign functions (Lee et al., 2021a). Assume that p_1 and p_2 are sequential component minimax approximate polynomials in this order. If the range within the approximation domain of p_1 is $[-1-b, -1+b] \cup [1-b, 1+b]$, the approximation domain of p_2 is designed to be this range. Since the minimax approximate polynomial usually diverges when the input value is outside the approximation domain, the result value of p_2 will diverge greatly and lead to a failure of APR if the result value of p_1 is outside of $[-1-b, 1+b]$.

Consider the neighborhood of the local maximum point x_0 such that $p_1(x_0) = 1+b$. $p_1(x)$ can be approximated by the second Taylor polynomial $T_{p_1,2}(x) = p_1(x_0) - a(x-x_0)^2$ for positive real number a near x_0, which is also valid in the complex domain. When the value of $x - x_0$ is a pure imaginary number, the value of $T_{p_1,2}(x)$ is always greater than $p_1(x_0) = 1 + b$. Thus, there exist some values of x such that $\mathrm{Re}(p_1(x))$ is outside of $[-1-b, 1+b]$ when allowing imaginary noise, which leads to a failure in the whole ResNet inference.

Hence, to stably perform ResNet with many layers, it is important to remove the imaginary part of the input of each APR. We propose to apply the *imaginary-removing bootstrapping* operation before the APR. We homomorphically compute the formula $\mathrm{Re}(x) = x/2 + \overline{x/2}$ by halving all coefficient values in SLOTTOCOEFF operation in the bootstrapping and homomorphically computing $v + \bar{v}$. This additional operation costs only one KSO for homomorphic conjugation, and no additional level is consumed.

Figure 6 shows the mean of absolute values of imaginary parts after each layer using normal and imaginary-removing bootstrappings for one instance of ResNet-110 inference. We observe that the diverging phenomenon occurs after the 69th layer due to the accumulated noise in the imaginary part. This catastrophic divergence occurs for 12 images out of 50 tested images (i.e., 24% of tested images). The proposed imaginary-removing bootstrapping makes the noise of imaginary parts remain much smaller during deeper ResNet inference, and we confirm that imaginary-removing bootstrapping never causes this diverging phenomenon when conducting simulations for a various number of layers and test images as in Section 8. It is worth mentioning that we address this divergence problem of VDSCNNs on FHE and propose a solution for the first time.

6. Optimization of Level Consumption

In our implementation, convolution, batch normalization, bootstrapping, and APR are repeatedly performed in this order. Since the bootstrapping and APR work only for input values in $[-1, 1]$, it is required to do scaling by $1/B$ before bootstrapping and by B after the APR. We set sufficiently large B to maintain all the computed values within $[-B, B]$. We set $B = 40$ and $B = 65$ for the CIFAR-10 and CIFAR-

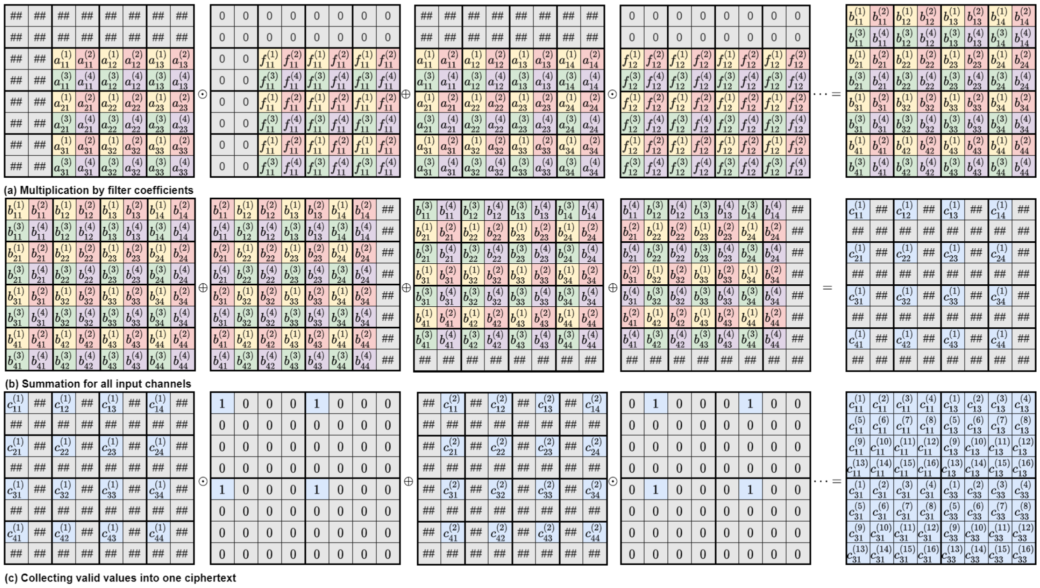

Figure 4: Multiplexed convolution algorithm for multiplexed input tensor for $s = 2$, $k_i = 2$, and $h_i = w_i = 4$.

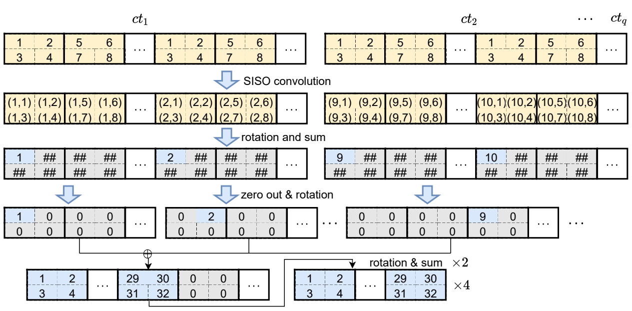

Figure 5: Multiplexed parallel convolution algorithm when $k_i = 2$ and $c_o = c_n = 32$.

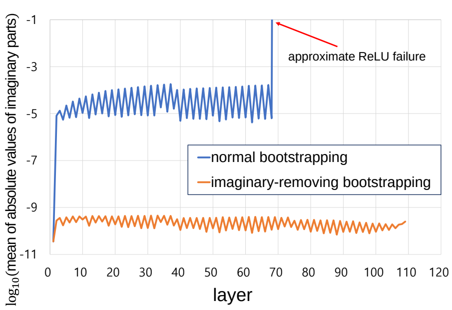

Figure 6: Mean of absolute values of imaginary parts after each layer when performing ResNet-110 inference using the normal bootstrapping and the proposed imaginary-removing bootstrapping.

100 datasets, respectively, and each value of B is obtained by adding some margin to the maximum value of all used values.

We propose a method of reducing level consumption by integrating computations, as shown in Figure 7. We multiply the constant of batch normalization (i.e., a) during the selecting procedure in convolution instead of batch normalization, and then add a modified constant vector by taking into account the value of B during batch normalization. By these judicious integrations, we can save three levels. Figure 7 describes this level optimization technique, and the proposed convolution/batch normalization integration algorithm, denoted as MULTPARCONVBN, is presented in Appendix H.

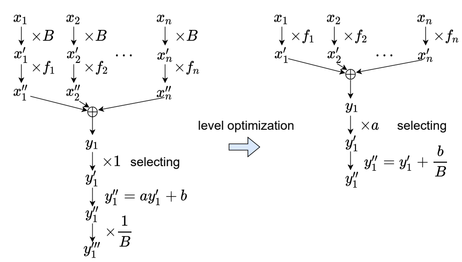

Figure 7: Level optimization by integrating computations.

7. The Proposed Architecture for ResNet on the RNS-CKKS Scheme

7.1. Parameter Setting

We set the polynomial degree $N = 2^{16}$ and the number of full slots $n_t = 2^{15}$. We optimize some parameters used in (Lee et al., 2021c) to achieve a higher security level. First, we set the Hamming weight of the secret key to 192, which is larger than 64 used in (Lee et al., 2021c) 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 APR can be achieved. Based on the hybrid dual attack for the learning with errors (LWE) problem with the sparse secret key (Cheon et al., 2019), the total modulus bit length for 128-bit security is 1,553 bits.

We use the RS bootstrapping 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 and 1, respectively, and the number of the 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 imaginary-removing bootstrappings for $n = 2^{14}, 2^{13}$, and 2^{12} as Boot14, Boot13, Boot12, respectively.

We use the approximate homomorphic ReLU algorithm that uses APRs using a composition of minimax approximate polynomial as in (Lee et al., 2021a;b). We use the precision parameter $\alpha = 13$ and set of degrees $\{15, 15, 27\}$. We refer to the homomorphic ReLU algorithm for these parameters as AppReLU(ct_x). The ℓ_1-norm approximation error of AppReLU is less than 2^{-13}, and this marginal error enables us to use the pre-trained parameters of standard ResNet models. That is, we do not need to train/retrain contrary to a nonstandard HE-friendly network.

7.2. The Proposed Structure of ResNet on the RNS-CKKS Scheme

We classify 32×32 CIFAR-10 and CIFAR-100 images for our evaluation. We devise downsampling and average pooling algorithms that support multiplexed tensors. We refer to these algorithms as Downsamp and AvgPool, presented in Appendix G. We implement fully connected layer using the diagonal method in (Halevi & Shoup, 2014). We implement ResNet-20/32/44/56/110 on the RNS-CKKS scheme using MultParConvBN, AppReLU, Boot, AvgPool, Downsamp, and fully connected layer. Figure 8 shows the proposed ResNet structure on the RNS-CKKS scheme, where MultParConvBN is simply referred to as ConvBN. The parameters used in ConvBN and Downsamp are presented in Appendix I.

While two sequential bootstrappings are required to perform APR, convolution, and batch normalization in one layer in (Lee et al., 2021c), only single use of bootstrapping is necessary for our implementation because we reduce the required level consumption for convolution, batch normalization, and bootstrapping a lot compared to (Lee et al., 2021c). In addition, the proposed architecture for ResNet uses a 1,501-bit modulus, and thus, it achieves the standard 128-bit security level.

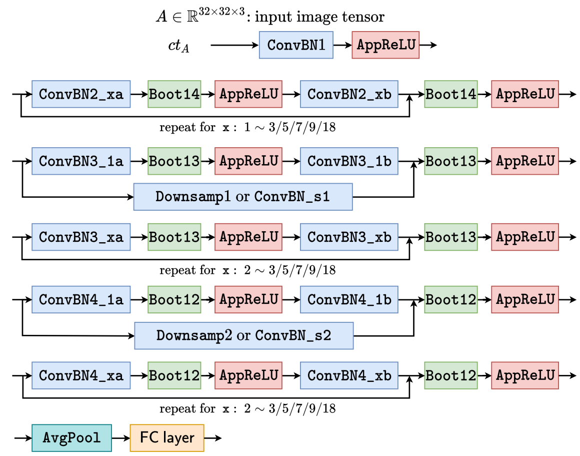

Figure 8: Structure of the proposed ResNet-20/32/44/56/110 on the RNS-CKKS scheme. The input image is packed in ct_A in a raster scan fasion and using RS packing.

8. Simulation Results

In this section, numerical results of the proposed architecture for ResNet are presented. The numerical analyses are conducted on the representative RNS-CKKS scheme library SEAL (SEAL) 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 (Krizhevsky et al., 2009). We use pre-trained parameters for standard ResNet-20/32/44/56/110.

8.1. Latency

First, we perform ResNet-20/32/44/56/110 using the proposed architecture on the RNS-CKKS scheme. We require 3,306 KSOs for ResNet-20, which is $116\times$ smaller than 384,160 in (Lee et al., 2021c). Table 3 shows the classification runtime for one CIFAR-10/CIFAR-100 image using ResNet models on the RNS-CKKS scheme. Due to the large reduction of the number of KSOs, while the previous implementation in (Lee et al., 2021c) takes 10,602s with 64 CPU threads to perform ResNet-20 on the RNS-CKKS scheme, the proposed implementation takes 2,271s to perform ResNet-20 even with one CPU thread, which is $4.67\times$ reduction in latency. Considering that our implementation only uses one CPU thread, we can expect more than $100\times$ and $1000\times$ lower latency on GPU and hardware accelera-

component	CIFAR-10													CIFAR-100	
	(Lee et al., 2021c) (64 threads)		proposed (single thread)											proposed (single thread)	
	ResNet-20		ResNet-20		ResNet-32		ResNet-44		ResNet-56		ResNet-110			ResNet-32	
	runtime	percent	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%		542s	13.7%
AppReLU	-	-	257s	11.3%	406s	10.9%	583s	11.2%	762s	11.1%	1,475s	11.1%		510s	12.9%
Boot	-	-	1,651s	72.6%	2,760s	74.0%	3,874s	74.1%	5,113s	74.6%	9,936s	74.8%		2,864s	72.7%
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%		2s	0.05%
FC layer	-	-	10s	0.4%	10s	0.3%	10s	0.2%	10s	0.1%	10s	0.08%		24s	0.6%
total	10,602s	100%	2,271s	100%	3,730s	100%	5,224s	100%	6,852s	100%	13,282s	100%		3,942s	100%

Table 3: Classification runtime for one CIFAR-10/CIFAR-100 image using ResNet on the RNS-CKKS scheme

		model	runtime	amortized runtime
CIFAR-10	(Lee et al., 2021c) (one image, 64 threads)	ResNet-20	10,602s	10,602s
	proposed (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
CIFAR-100	proposed (50 images, 50 threads)	ResNet-32	6,351s	127s

Table 4: Classification (amortized) runtime for multiple CIFAR-10/CIFAR-100 images using ResNet models on the RNS-CKKS scheme

tors, respectively (Jung et al., 2021; Kim et al., 2021b).

We also succeed in implementing the standard ResNet-32/44/56/110 on the RNS-CKKS scheme for the first time. Table 3 shows that the runtime increases *linearly* with the number of layers, which is quite difficult to be expected in leveled HEs.

8.2. Amortized Runtime

Since servers should classify multiple images of clients in many cases, not only the latency but also the amortized runtime for multiple images, i.e., runtime per image, is important. Since the proposed implementation requires only one thread unlike in (Lee et al., 2021c), multiple threads allow us to classify multiple images simultaneously. Table 4 shows the runtime and amortized runtime of classification for multiple CIFAR-10/CIFAR-100 images using ResNet models on the RNS-CKKS scheme. The proposed implementation of ResNet-20 takes 3,973s 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 (Lee et al., 2021c).

dataset	model	#test images	#success	backbone accuracy	obtained accuracy
CIFAR -10	ResNet-20	10,000	9,132	91.52%	91.31%
	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%*
CIFAR -100	ResNet-32	10,000	6,943	69.5%	69.43%

Table 5: Classification accuracies for CIFAR-10/CIFAR-100 images using ResNet models on the RNS-CKKS scheme. An asterisk (*) implies that not all 10,000 test images have been tested.

8.3. Accuracy

Table 5 presents the classification accuracies for CIFAR-10/CIFAR-100 images using ResNet models on the RNS-CKKS scheme. Thanks to resolving the catastrophic divergence phenomenon by the proposed imaginary-removing bootstrapping, all the obtained accuracies for ResNet-20/32/44/56/110 are very close to those of backbone CNNs. This implies that the proposed implementation of VDSCNNs on the RNS-CKKS scheme can benefit from high accuracies of various pre-trained VDSCNNs that have widely been developed already.

9. Conclusions

We constructed an efficient privacy-preserving VDSCNN model on the RNS-CKKS scheme. First, we minimized the bootstrapping runtime via multiplexed packing and proposed multiplexed parallel convolution algorithm that works for multiplexed input tensor, which also supports strided convolutions. Further, we addressed the catastrophic divergence problem of VDSCNNs on the RNS-CKKS scheme and resolved it by the proposed imaginary-removing bootstrapping. By carefully integrating computations, we effectively reduced the level consumption. Our simulation results reported $4.67\times$ lower latency and $134\times$ lower amortized runtime for ResNet-20 inference compared to (Lee et al., 2021c) while achieving the 128-bit security. We also successfully implemented ResNet-32/44/56/110 on the RNS-CKKS scheme for the first time.

References

Boemer, F., Lao, Y., Cammarota, R., and Wierzynski, C. nGraph-HE: A graph compiler for deep learning on homomorphically encrypted data. In *Proceedings of the 16th ACM International Conference on Computing Frontiers*, pp. 3–13, 2019.

Bossuat, J.-P., Mouchet, C., Troncoso-Pastoriza, J., and Hubaux, J.-P. Efficient bootstrapping for approximate homomorphic encryption with non-sparse keys. In *International Conference on the Theory and Applications of Cryptographic Techniques*, 2021.

Bossuat, J.-P., Troncoso-Pastoriza, J. R., and Hubaux, J.-P. Bootstrapping for approximate homomorphic encryption with negligible failure-probability by using sparse-secret encapsulation. *Cryptol. ePrint Arch., Tech. Rep. 2022/024*, 2022. https://ia.cr/2022/024.

Cheon, J. H., Han, K., Kim, A., Kim, M., and Song, Y. A full RNS variant of approximate homomorphic encryption. In *Proceedings of International Conference on Selected Areas in Cryptography*, pp. 347–368, 2018a.

Cheon, J. H., Han, K., Kim, A., Kim, M., and Song, Y. Bootstrapping for approximate homomorphic encryption. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pp. 360–384. Springer, 2018b.

Cheon, J. H., Hhan, M., Hong, S., and Son, Y. A hybrid of dual and meet-in-the-middle attack on sparse and ternary secret LWE. *IEEE Access*, 7:89497–89506, 2019.

Chou, E., Beal, J., Levy, D., Yeung, S., Haque, A., and Fei-Fei, L. Faster CryptoNets: Leveraging sparsity for real-world encrypted inference. *arXiv preprint arXiv:1811.09953*, 2018.

Dathathri, R., Saarikivi, O., Chen, H., Laine, K., Lauter, K., Maleki, S., Musuvathi, M., and Mytkowicz, T. CHET: An optimizing compiler for fully-homomorphic neural-network inferencing. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 142–156, 2019.

Gilad-Bachrach, R., Dowlin, N., Laine, K., Lauter, K., Naehrig, M., and Wernsing, J. Cryptonets: Applying neural networks to encrypted data with high throughput and accuracy. In *Proceedings of International Conference on Machine Learning*, pp. 201–210. PMLR, 2016.

Halevi, S. and Shoup, V. Algorithms in HElib. In *Annual Cryptology Conference*, pp. 554–571. Springer, 2014.

He, K., Zhang, X., Ren, S., and Sun, J. Deep residual learning for image recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 770–778, 2016.

Ioffe, S. and Szegedy, C. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International conference on machine learning*, pp. 448–456. PMLR, 2015.

Jung, W., Kim, S., Ahn, J. H., Cheon, J. H., and Lee, Y. Over 100x faster bootstrapping in fully homomorphic encryption through memory-centric optimization with GPUs. *Cryptol. ePrint Arch., Tech. Rep. 2021/508*, 2021.

Juvekar, C., Vaikuntanathan, V., and Chandrakasan, A. GAZELLE: A low latency framework for secure neural network inference. In *Proceedings of the 27th USENIX Security Symposium*, pp. 1651–1669, 2018.

Kim, M., Jiang, X., Lauter, K., Ismayilzada, E., and Shams, S. HEAR: Human action recognition via neural networks on homomorphically encrypted data. *arXiv preprint arXiv:2104.09164*, 2021a.

Kim, S., Kim, J., Kim, M. J., Jung, W., Rhu, M., Kim, J., and Ahn, J. H. BTS: An accelerator for bootstrappable fully homomorphic encryption. *arXiv preprint arXiv:2112.15479*, 2021b.

Krizhevsky, A., Hinton, G., et al. Learning multiple layers of features from tiny images. *CiteSeerX Technical Report, University of Toronto*, 2009.

Lee, E., Lee, J.-W., No, J.-S., and Kim, Y.-S. Minimax approximation of sign function by composite polynomial for homomorphic comparison. *IEEE Transactions on Dependable and Secure Computing, accepted for publication*, 2021a.

Lee, J., Lee, E., Lee, J.-W., Kim, Y., Kim, Y.-S., and No, J.-S. Precise approximation of convolutional neural networks for homomorphically encrypted data. *arXiv preprint arXiv:2105.10879*, 2021b.

Lee, J.-W., Kang, H., Lee, Y., Choi, W., Eom, J., Deryabin, M., Lee, E., Lee, J., Yoo, D., Kim, Y.-S., et al. Privacy-preserving machine learning with fully homomorphic encryption for deep neural network. *arXiv preprint arXiv:2106.07229*, 2021c.

Lee, Y., Lee, J.-W., Kim, Y.-S., Kang, H., and No, J.-S. High-precision approximate homomorphic encryption by error variance minimization. *Cryptol. ePrint Arch., Tech. Rep. 2020/834*, 2021d.

Lou, Q. and Jiang, L. HEMET: A homomorphic-encryption-friendly privacy-preserving mobile neural network architecture. In *Proceedings of the 38th International Conference on Machine Learning*, volume 139, pp. 7102–7110. PMLR, 2021.

Mishra, P., Lehmkuhl, R., Srinivasan, A., Zheng, W., and Popa, R. A. DELPHI: A cryptographic inference service for neural networks. In *Proceedings of the 29th USENIX Security Symposium*, pp. 2505–2522, 2020.

Park, J., Kim, M. J., Jung, W., and Ahn, J. H. AESPA: Accuracy preserving low-degree polynomial activation for fast private inference. *arXiv preprint arXiv:2201.06699*, 2022.

SEAL. Microsoft SEAL (release 3.6). https://github.com/Microsoft/SEAL, November 2020. Microsoft Research, Redmond, WA.

Simonyan, K. and Zisserman, A. Very deep convolutional networks for large-scale image recognition. In *International Conference on Learning Representations*, 2015.

Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D., Erhan, D., Vanhoucke, V., and Rabinovich, A. Going deeper with convolutions. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 1–9, 2015.

A. Notations and Description of Parameters

In this section, specific notations and description of parameters are provided. We use $\mathbf{x}$ to denote a vector in $\mathbb{R}^n$ for some integer n. For $\mathbf{x} = (x_0, x_1, \cdots, x_{n-1})$, $\langle \mathbf{x} \rangle_r$ denotes 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.

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 6), downsampling, and average pooling. The specific values of parameters that are used in our simulation can be seen in Table 6 in Section 7.

B. RNS-CKKS Scheme

In this section, the RNS-CKKS scheme is described in more detail. RNS-CKKS is an FHE scheme that supports fixed-point arithmetic operations on encrypted data. The ciphertext in the RNS-CKKS scheme is the form of $(b, a) \in R^2_{Q_\ell}$, where $Q_\ell = \prod_{i=0}^{\ell} q_i$ is a product of prime numbers and $R_{Q_\ell} = \mathbb{Z}_{Q_\ell}[X]/\langle X^N + 1 \rangle$. $N/2$ real (or complex) numbers are encrypted in $N/2$ slots of a single ciphertext, and we denote $N/2$ as n_t. We denote the encryption and decryption in RNS-CKKS scheme as $\mathsf{Enc}(\cdot)$ and $\mathsf{Dec}(\cdot)$, respectively. 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 $\mathbb{R}^{n_t}$.

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

C. Mapping of Three-Dimensional Tensor to One-Dimensional Vector

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, and $\overline{A}$ can be the original tensor or *(parallelly) multiplexed tensor* defined in Section F. 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 9 describes this Vec function.

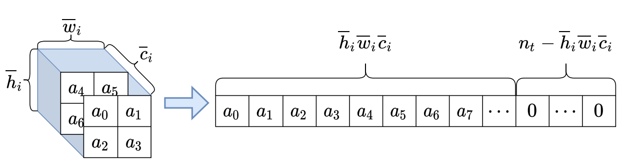

Figure 9: Vec function that maps a given tensor in $\mathbb{R}^{\overline{h}_i \times \overline{w}_i \times \overline{c}_i}$ to a vector in $\mathbb{R}^{n_t}$.

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.

D. Batch Normalization on Homomorphic Encryption

Batch normalization (Ioffe & Szegedy, 2015) 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 $\mathsf{Vec}(\overline{C}) \cdot (\mathsf{Vec}(A) - \mathsf{Vec}(\overline{M})) + \mathsf{Vec}(\overline{I}) = \mathsf{Vec}(\overline{C}) \cdot \mathsf{Vec}(A) + (\mathsf{Vec}(\overline{I}) - \mathsf{Vec}(\overline{C}) \cdot \mathsf{Vec}(\overline{M}))$. This can be implemented on HE by using one homomorphic addition and scalar multiplication. That is, for the input tensor ciphertext $\mathsf{ct_a}$, we just perform $\mathsf{Vec}(\overline{C}) \odot \mathsf{ct_a} \oplus (\mathsf{Vec}(\overline{I}) - \mathsf{Vec}(\overline{C}) \cdot \mathsf{Vec}(\overline{M}))$.

E. Multiplexed Packing

For $t_i = \lceil \frac{c_i}{k_i^2} \rceil$, $\mathsf{MultPack}$ 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 ciphertext $\mathsf{Enc}(\mathsf{Vec}(A')) \in \mathbb{R}^{n_t}$, 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

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

for $0 \le i_3 < k_i h_i, 0 \le i_4 < k_i w_i$, and $0 \le i_5 < t_i$.

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 corresponding plaintext tensor to be packed into the ciphertext slots using the multiplexed packing method throughout the entire CNN, where the value of gap k_i can be changed.

F. Convolution Algorithms for Multiplexed Tensor

F.1. Multiplexed Convolution

For description of MULTCONV algorithm, we require some definitions and a subroutine algorithm.

The filter (weight tensor) of the convolution is $U \in \mathbb{R}^{f_h \times f_w \times c_i \times c_o}$. First, we define $\mathsf{MultWgt}(U; i_1, i_2, i)$ 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 $\mathsf{MultWgt}$, 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:

Algorithm 1 SUMSLOTS$(\mathsf{ct_a}; m, p)$

Input: Tensor ciphertext $\mathsf{ct_a}$, number of added slots m, and gap p
Output: Tensor ciphertext $\mathsf{ct_c}$
$\mathsf{ct_b}^{(0)} \leftarrow \mathsf{ct_a}$
for $j \leftarrow 1$ **to** $\lfloor \log_2 m \rfloor$ **do**
    $\mathsf{ct_b}^{(j)} \leftarrow \mathsf{ct_b}^{(j-1)} \oplus \mathsf{Rot}(\mathsf{ct_b}^{(j-1)}; 2^{j-1} \cdot p)$
end for
$\mathsf{ct_c} \leftarrow \mathsf{ct_b}^{(\lfloor \log_2 m \rfloor)}$
for $j \leftarrow 0$ **to** $\lfloor \log_2 m \rfloor - 1$ **do**
    if $\lfloor m/2^j \rfloor \bmod 2 = 1$ **then**
        $\mathsf{ct_c} \leftarrow \mathsf{ct_c} \oplus \mathsf{Rot}(\mathsf{ct_b}^{(j)}; \lfloor m/2^{j+1} \rfloor \cdot 2^{j+1} p)$
    end if
end for
Return $\mathsf{ct_c}$

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

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

In addition to the weight tensor, it is also required to define *multiplexed selecting tensor* $S'^{(i)} = (S'^{(i)}_{i_3,i_4,i_5})_{0 \leq i_3 < k_o h_o, 0 \leq i_4 < k_o w_o, 0 \leq i_5 < t_o} \in \mathbb{R}^{k_o h_o \times k_o w_o \times t_o}$, 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 $0 \leq i_3 < k_o h_o, 0 \leq i_4 < k_o w_o$, and $0 \leq i_5 < t_o$.

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 $\mathsf{MultWgt}$ function, multiplexed selecting tensor $S'^{(i)}$, and SUMSLOTS algorithm. Here, $\mathsf{ct_{zero}}$ is a ciphertext of all-zero vector $\mathbf{0} \in \mathbb{R}^{n_t}$.

F.2. Multiplexed Parallel Convolution

We propose a *multiplexed parallel packing* method $\mathsf{MultParPack}$ that packs p_i identical multiplexed tensors into one ciphertext for $p_i = 2^{\lfloor \log_2 (\frac{n_t}{k_i^2 h_i w_i t_i}) \rfloor}$. Figure 10 describes how to perform multiplexed parallel 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 $\mathsf{MultPack}(A) = \mathsf{Enc}(\mathsf{Vec}(A'))$ and simply places p_i copies of A' in sequence. This extended tensor is mapped to a vector in $\mathbb{R}^{n_t}$ using Vec function and then encrypted into a ciphertext. 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 $\mathsf{MultParPack}$ function is given as:

$$\mathsf{MultParPack}(A) = \bigoplus_{j=0}^{p_i - 1} \mathsf{Rot}(\mathsf{MultPack}(A); j(n_t/p_i)).$$

We require each corresponding plaintext tensor to be packed into the ciphertext slots using the multiplexed parallel packing method during the entire CNN. We propose a *multiplexed parallel convolution* algorithm, MULTPARCONV, which is an improved algorithm of MULTCONV. MULTPARCONV takes a *parallelly multiplexed tensor* for gap k_i as an input and

Algorithm 2 $\textsc{MultConv}(\mathsf{ct}'_{\mathsf{a}}, U)$

Input: Multiplexed tensor ciphertext $\mathsf{ct}'_{\mathsf{a}}$ and weight tensor U
Output: Multiplexed tensor ciphertext $\mathsf{ct}'_{\mathsf{d}}$
$\mathsf{ct}'_{\mathsf{d}} \leftarrow \mathsf{ct}_{\mathsf{zero}}$
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 for
end for
for $i \leftarrow 0$ **to** $c_o - 1$ **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**
      $\mathsf{ct}'_{\mathsf{b}} \leftarrow \mathsf{ct}'_{\mathsf{b}} \oplus \mathsf{ct}'^{(i_1,i_2)} \odot \mathsf{MultWgt}(U; i_1, i_2, i)$
    end for
  end for
  $\mathsf{ct}'_{\mathsf{c}} \leftarrow \textsc{SumSlots}(\mathsf{ct}'_{\mathsf{b}}; k_i, 1)$
  $\mathsf{ct}'_{\mathsf{c}} \leftarrow \textsc{SumSlots}(\mathsf{ct}'_{\mathsf{c}}; k_i, kw_i)$
  $\mathsf{ct}'_{\mathsf{c}} \leftarrow \textsc{SumSlots}(\mathsf{ct}'_{\mathsf{c}}; t_i, k^2 h_i w_i)$
  $\mathsf{ct}'_{\mathsf{d}} \leftarrow \mathsf{ct}'_{\mathsf{d}} \oplus \mathsf{Rot}(\mathsf{ct}'_{\mathsf{c}}; -\lfloor i/k_o^2 \rfloor k_o^2 h_o w_o - \lfloor (i \bmod k_o^2)/k_o \rfloor k_o w_o - (i \bmod k_o)) \odot \mathsf{Vec}(S'^{(i)})$
end for
Return $\mathsf{ct}'_{\mathsf{d}}$

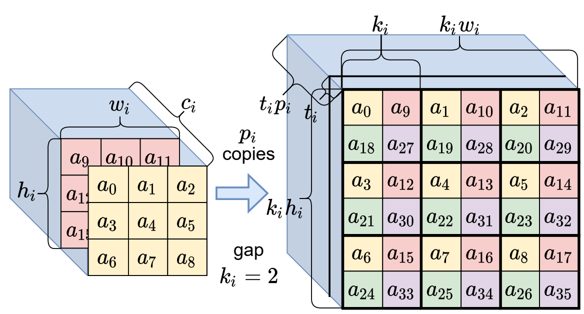

Figure 10: Multiplexed parallel packing method $\mathsf{MultParPack}$ when $k_i^2 h_i w_i t_i \mid n_t$.

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 $\textsc{MultConv}$ performs multiplication by weight and summing up c_o times, multiplexed parallel convolution algorithm $\textsc{MultParConv}$ performs only q times, reducing the required number of rotations to about $1/p_i$.

Before description of $\textsc{MultParConv}$ in detail, it is required to define $\mathsf{ParMultWgt}(U; i_1, i_2, i_3)$ 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 $\mathsf{ParMultWgt}$, parallelly multiplexed shifted weight tensor $\overline{U}''^{(i_1,i_2,i_3)} = (\overline{U}''^{(i_1,i_2,i_3)}_{i_5,i_6,i_7})_{0 \le i_5 < k_i h_i, 0 \le i_6 < k_i w_i, 0 \le i_7 < t_i p_i} \in \mathbb{R}^{k_i h_i \times k_i w_i \times t_i p_i}$ 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 $0 \le i_5 < k_i h_i$, $0 \le i_6 < k_i w_i$, and $0 \le i_7 < t_i p_i$. Then, $\mathsf{ParMultWgt}$ is defined as $\mathsf{ParMultWgt}(U; i_1, i_2, i_3) =$

Algorithm 3 $\textsc{MultParConv}(\mathsf{ct}''_\mathsf{a}, U)$

Input: Parallelly multiplexed tensor ciphertext ct'_a and weight tensor U
Output: Parallelly multiplexed tensor ciphertext
$\mathsf{ct}''_\mathsf{d} \leftarrow \mathsf{ct}_\mathsf{zero}$
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 for
end for
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**
      $\mathsf{ct}''_\mathsf{b} \leftarrow \mathsf{ct}''_\mathsf{b} \oplus \mathsf{ct}''^{(i_1,i_2)} \odot \mathsf{ParMultWgt}(U; i_1, i_2, i_3)$
    end for
  end for
  $\mathsf{ct}''_\mathsf{c} \leftarrow \textsc{SumSlots}(\mathsf{ct}''_\mathsf{b}; k_i, 1)$
  $\mathsf{ct}''_\mathsf{c} \leftarrow \textsc{SumSlots}(\mathsf{ct}''_\mathsf{c}; k_i, k_i w_i)$
  $\mathsf{ct}''_\mathsf{c} \leftarrow \textsc{SumSlots}(\mathsf{ct}''_\mathsf{c}; t_i, k_i^2 h_i w_i)$
  for $i_4 \leftarrow 0$ **to** $\min(p_i - 1, c_o - 1 - p_i i_3)$ **do**
    $i \leftarrow p_i i_3 + i_4$
    $\mathsf{ct}''_\mathsf{d} \leftarrow \mathsf{ct}''_\mathsf{d} \oplus \mathsf{Rot}(\mathsf{ct}''_\mathsf{c}; -\lfloor i/k_o^2 \rfloor k_o^2 h_o w_o + \lfloor n_t/p_i \rfloor (i \bmod p_i) - \lfloor (i \bmod k_o^2)/k_o \rfloor k_o w_o - i \bmod k_o) \odot \mathsf{Vec}(S'^{(i)})$
  end for
end for
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 for
Return ct''_d

$\mathsf{Vec}(\overline{U}''^{(i_1,i_2,i_3)})$. The multiplexed selecting tensor $S'^{(i)}$ defined in Section 3 is also used in $\textsc{MultParConv}$.

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

G. Multiplexed Parallel Batch Normalization, Downsampling, and Average Pooling

In Section 4, we proposed multiplexed parallel convolution algorithm, $\textsc{MultParConv}$ 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 multiplexed parallel packing method. Thus, new batch normalization, downsampling, and average pooling algorithms that work for an input ciphertext having plaintext tensor using $\mathsf{MultParPack}$ are described in this section.

G.1. Multiplexed Parallel Batch Normalization

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

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

for $0 \leq j < n_t$, 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 ParMultBN that performs batch normalization using this ParBNConst function, and Algorithm 4 describes the proposed ParMultBN.

Algorithm 4 ParMultBN(ct_a'', C, M, I)

1: **Input:** Parallelly multiplexed tensor ciphertext ct_a'' and batch normalization constant vectors $C, M, I \in \mathbb{R}_i^c$
2: **Output:** Parallelly multiplexed tensor ciphertext ct_b''
3: $\mathbf{c}'' \leftarrow \mathsf{ParBNConst}(C)$, $\mathbf{m}'' \leftarrow \mathsf{ParBNConst}(M)$, $\mathbf{i}'' \leftarrow \mathsf{ParBNConst}(I)$
4: $\mathsf{ct}_\mathsf{b}'' \leftarrow \mathbf{c}'' \odot \mathsf{ct}_\mathsf{a}'' \oplus (\mathbf{i}'' - \mathbf{c}'' \cdot \mathbf{m}'')$
5: **Return** ct_b''

G.2. Multiplexed Parallel Downsampling

ResNet models for the CIFAR-10 dataset require two downsampling processes. We propose DownSamp algorithm that performs 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 $S''^{(i_1,i_2)} = (S''^{(i_1,i_2)}_{i_3,i_4,i_5})_{0\leq i_3<k_ih_i, 0\leq i_4<k_iw_i, 0\leq i_5<t_i} \in \mathbb{R}^{k_ih_i\times k_iw_i\times t_i}$, which is used to select $4k_i$ valid values. Downsampling selecting tensor $S''^{(i_1,i_2)} = (S''^{(i_1,i_2)}_{i_3,i_4,i_5})_{0\leq i_3<k_ih_i, 0\leq i_4<k_iw_i, 0\leq i_5<t_i}$ for $0 \leq i_1 < k_i$ and $0 \leq i_2 < t_i$ 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 $0 \leq i_3 < k_ih_i, 0 \leq i_4 < k_iw_i$, and $0 \leq i_5 < t_i$. Algorithm 5 describes the proposed downsampling algorithm DownSamp.

Algorithm 5 DownSamp(ct_a'')

1: **Input:** Parallelly multiplexed tensor ciphertext ct_a''
2: **Output:** Parallelly multiplexed tensor ciphertext ct_c''
3: $\mathsf{ct}_\mathsf{c}'' \leftarrow \mathsf{ct}_\mathsf{zero}$
4: **for** $i_1 \leftarrow 0$ **to** $k_i - 1$ **do**
5:   **for** $i_2 \leftarrow 0$ **to** $t_i - 1$ **do**
6:     $i_3 \leftarrow \lfloor((k_ii_2 + i_1) \bmod 2k_o)/2\rfloor$
7:     $i_4 \leftarrow (k_ii_2 + i_1) \bmod 2$
8:     $i_5 \leftarrow \lfloor(k_ii_2 + i_1)/2k_o\rfloor$
9:     $\mathsf{ct}_\mathsf{b}'' \leftarrow \mathsf{ct}_\mathsf{a}'' \odot \mathsf{Vec}(S''^{(i_1,i_2)})$
10:     $\mathsf{ct}_\mathsf{c}'' \leftarrow \mathsf{ct}_\mathsf{b}'' \oplus \mathsf{Rot}(\mathsf{ct}_\mathsf{b}''; k_i^2h_iw_i(i_2 - i_5) + k_iw_i(i_1 - i_3) - k_ii_4)$
11:   **end for**
12: **end for**
13: $\mathsf{ct}_\mathsf{c}'' \leftarrow \mathsf{Rot}(\mathsf{ct}_\mathsf{c}''; -k_o^2h_ow_ot_i/8)$
14: **for** $j \leftarrow 0$ **to** $\log_2 p_o - 1$ **do**
15:   $\mathsf{ct}_\mathsf{c}'' \leftarrow \mathsf{ct}_\mathsf{c}'' \oplus \mathsf{Rot}(\mathsf{ct}_\mathsf{c}''; -2^jk_o^2h_ow_ot_o)$
16: **end for**
17: **Return** ct_c''

G.3. Average Pooling

When we reach the average pooling after performing all convolutions, batch normalizations, and APRs in the ResNet model, we have a ciphertext that contains data packed using MultParPack. The data of ciphertext packed by this multiplexed

Algorithm 6 $\textsc{AvgPool}(\mathsf{ct}''_{\mathsf{a}})$

Input: Parallelly multiplexed tensor ciphertext $\mathsf{ct}''_{\mathsf{a}}$
Output: One-dimensional array ciphertext ct_{b}
$\mathsf{ct}_{\mathsf{b}} \leftarrow \mathsf{ct}_{\mathsf{zero}}$
for $j \leftarrow 0$ **to** $\log_2 w_i - 1$ **do**
    $\mathsf{ct}''_{\mathsf{a}} \leftarrow \mathsf{Rot}(\mathsf{ct}''_{\mathsf{a}}; 2^j \cdot k_i)$
end for
for $j \leftarrow 0$ **to** $\log_2 h_i - 1$ **do**
    $\mathsf{ct}''_{\mathsf{a}} \leftarrow \mathsf{Rot}(\mathsf{ct}''_{\mathsf{a}}; 2^j \cdot k_i^2 w_i)$
end for
for $i_1 \leftarrow 0$ **to** $k_i - 1$ **do**
    for $i_2 \leftarrow 0$ **to** $t_i - 1$ **do**
        $\mathsf{ct}_{\mathsf{b}} \leftarrow \mathsf{ct}_{\mathsf{b}} \oplus \mathsf{Rot}(\mathsf{ct}''_{\mathsf{a}}; k_i^2 h_i w_i i_2 + k_i w_i i_1 - k_i(k_i i_2 + i_1)) \odot \bar{\mathbf{s}}'^{(k_i i_2 + i_1)}$
    end for
end for
Return ct_{b}

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 $\textsc{AvgPool}$ that not only performs average pooling but also rearranges indices.

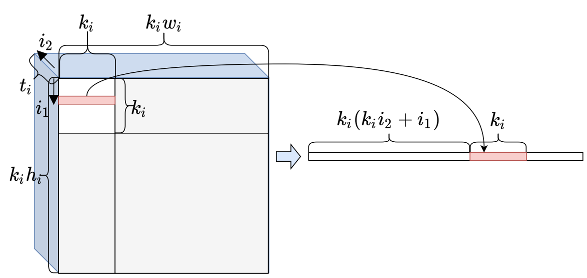

Figure 11: Rearranging process that selects and places $k_i^2 t_i$ valid values sequentially in $\textsc{AvgPool}$ algorithm.

Average pooling is the process that obtains a vector of $\mathbb{R}^{c_i}$ by computing the average value for $h_i w_i$ values for an input tensor of $\mathbb{R}^{h_i \times w_i \times c_i}$. To this end, we can add $h_i w_i$ values using rotations and additions of tensors. Dividing by $h_i w_i$ 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 $k_i^2 h_i w_i$ values, and the rest are the invalid garbage values. We place only $k_i^2 t_i$ 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:

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

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

H. Convolution/Batch Normalization Integration Algorithm

For a given input ciphertext ct_{x}, we can perform scaling processes, convolution, and batch normalization by evaluating $\mathsf{ct}_{\mathsf{x}} \odot (B \cdot \mathbf{1})$, $\textsc{MultParConv}(\mathsf{ct}_{\mathsf{x}}, U)$, $\mathbf{c}'' \odot \mathsf{ct}_{\mathsf{x}} \oplus (\mathbf{i}'' - \mathbf{c}'' \cdot \mathbf{m}'')$, and $\mathsf{ct}_{\mathsf{x}} \odot (\frac{1}{B} \cdot \mathbf{1})$ functions sequentially, where $\mathbf{1}$ is

Algorithm 7 $\text{MULTPARCONVBN}(\mathsf{ct}''_\mathsf{a}, U, C, M, I)$

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 for
end for
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**
      $\mathsf{ct}''_\mathsf{b} \leftarrow \mathsf{ct}''_\mathsf{b} \oplus \mathsf{ct}''^{(i_1,i_2)} \odot \mathsf{ParMultWgt}(U; i_1, i_2, i_3)$
    end for
  end for
  $\mathsf{ct}''_\mathsf{c} \leftarrow \text{SUMSLOTS}(\mathsf{ct}''_\mathsf{b}; k_i, 1)$
  $\mathsf{ct}''_\mathsf{c} \leftarrow \text{SUMSLOTS}(\mathsf{ct}''_\mathsf{c}; k_i, k_i w_i)$
  $\mathsf{ct}''_\mathsf{c} \leftarrow \text{SUMSLOTS}(\mathsf{ct}''_\mathsf{c}; t_i, k_i^2 h_i w_i)$
  for $i_4 \leftarrow 0$ **to** $\min(p_i - 1, c_o - 1 - p_i i_3)$ **do**
    $i \leftarrow p_i i_3 + i_4$
    $\mathsf{ct}''_\mathsf{d} \leftarrow \mathsf{ct}''_\mathsf{d} \oplus \mathsf{Rot}(\mathsf{ct}''_\mathsf{c}; -\lfloor i/k_o^2 \rfloor k_o^2 h_o w_o + \lfloor n_t/p_i \rfloor (i \bmod p_i) - \lfloor (i \bmod k_o^2)/k_o \rfloor k_o w_o - i \bmod k_o) \odot (\mathsf{ParBNConst}(C) \cdot \mathsf{Vec}(S'^{(i)}))$
  end for
end for
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 for
$\mathsf{ct}''_\mathsf{d} \leftarrow \mathsf{ct}''_\mathsf{d} \ominus \frac{1}{B}(\mathbf{c}'' \cdot \mathbf{m}'' - \mathbf{i}'')$
Return ct''_d

all-one vector in $\mathbb{R}^n$. Considering MULTPARCONV is a linear function, these operations are equivalent to evaluating

$$\begin{aligned} & (\mathbf{c}'' \odot \text{MULTPARCONV}(\mathsf{ct}_\mathsf{x}, BU) \oplus (\mathbf{i}'' - \mathbf{c}'' \cdot \mathbf{m}'')) \odot (\frac{1}{B} \cdot \mathbf{1}) \\ = \ & \mathbf{c}'' \odot \text{MULTPARCONV}(\mathsf{ct}_\mathsf{x}, U) \oplus \frac{1}{B}(\mathbf{i}'' - \mathbf{c}'' \cdot \mathbf{m}''). \end{aligned}$$

Here, if we perform $\text{MULTPARCONV}(\mathsf{ct}_\mathsf{x}, U)$ while replacing the original selecting tensor $\mathsf{Vec}(S'^{(i)})$ by $\mathsf{ParBNConst}(C) \cdot \mathsf{Vec}(S'^{(i)})$, we can perform $\mathbf{c}'' \odot \text{MULTPARCONV}(\mathsf{ct}_\mathsf{x}, U)$ 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.

I. Parameters

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 determined differently for each component such as convolution/batch normalization integration algorithm and downsampling. Table 6 shows the values of parameters that are used in each component of the proposed ResNet structure in Figure 8.

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 6: Parameters that are used in each ConvBN or Downsamp process