High-Throughput Deep Convolutional Neural Networks on Fully Homomorphic Encryption Using Channel-By-Channel Packing

Jung Hee Cheon [1 2] **Minsik Kang** [1] **Taeseong Kim** [1] **Junyoung Jung** [1] **Yongdong Yeo** [1]

Abstract

Secure Machine Learning as a Service is a viable solution where clients seek secure delegation of the ML computation while protecting their sensitive data. We propose an efficient method to securely evaluate deep standard convolutional neural networks based on CKKS fully homomorphic encryption, in the manner of batch inference. In this paper, we introduce a packing method called *Channel-by-Channel Packing* that maximizes the slot compactness and single-instruction-multiple-data capabilities in ciphertexts. Along with further optimizations such as lazy rescaling, lazy Baby-Step Giant-Step, and ciphertext level management, we could significantly reduce the computational cost of standard ResNet inference. Simulation results show that our work has improvements in amortized time by $5.04\times$ (from 79.46s to 15.76s) and $5.20\times$(from 455.56s to 87.60s) for ResNet-20 and ResNet-110, compared to the previous best results, resp. We also got a dramatic reduction in memory usage for rotation keys from several hundred GBs to 6.91GB, which is about $38\times$ smaller than the previous result.

1. Introduction

In a conventional Machine Learning as a Service (MLaaS) scenario, users upload their personal data to cloud servers to make use of a provided service. However, the users would be reluctant to upload their privacy-sensitive data such as medical information to the server, since they might not fully trust the server. Therefore, it is important for MLaaS providers to ensure that they protect the confidentiality and privacy of each individual during MLaaS.

Privacy-Preserving Machine Learning (PPML) provides a tool that can be utilized in this scenario. Fully homomorphic encryption (FHE) has been considered one of the most appropriate tools for PPMLs that achieve both strong security in the cryptographic sense and succinctness of communication. FHE supports computations on encrypted data without decrypting it, which enables MLaaS providers to perform inferences directly on the encrypted personal data of their clients.

Among various FHE schemes, Cheon-Kim-Kim-Song (CKKS) scheme (Cheon et al., 2017) is regarded as the most competent scheme for PPML, since CKKS supports parallel fixed-point arithmetic operations on each encrypted real number in slots, in a single-instruction-multiple-data (SIMD) manner. In order to fully utilize the SIMD properties of CKKS, it is crucial to minimize the unused slots, and also to minimize the operations between the slots inside a single ciphertext. With CKKS, we propose a new methodology which can achieve the following goals, by utilizing SIMD.

One is to implement practical PPML for deep standard CNNs (SCNNs), which is one of the necessities of MLaaS. In particular, we target the ResNet model (He et al., 2016), which is a widely used SCNN that effectively operates on large datasets such as CIFAR-10/100 (Krizhevsky, 2009), and ImageNet (Deng et al., 2009).

The other is batch inference, which is a service that server handles multiple inferences at once. Batch inference is used in situations where the throughput of inference is more important than response latency, such as in the field of user-customized recommendations (AmazonAWS, 2023). This work focuses on actualizing batch inference for secure MLaaS with standard CNNs, by making better use of the SIMD properties of CKKS.

Inference on CNN with HE was previously attempted by several previous works (Gilad-Bachrach et al., 2016; Chou et al., 2018; Dathathri et al., 2019; Al Badawi et al., 2021), but most of the works focused on building HE-friendly networks. Their constructions modify or redesign the CNN model, which requires retraining the network. This not only introduces inefficacy but also falls short of the MLaaS requirement of providing SCNNs. There is a recent line of work (Lee et al., 2021; 2022b;a) that targets precise approx-

[1]Department of Mathematical Sciences, Seoul National University, Seoul, Republic of Korea [2]CryptoLab. Inc., Seoul, Republic of Korea.

imation of pre-trained deep SCNNs. (Lee et al., 2022b) was the first work to completely evaluate the ResNet-20 on RNS-CKKS. The following work (Lee et al., 2022a) proposed an improved packing method that increases slot efficiency, and achieved state-of-the-art results for ResNet-20 and deeper networks, up to ResNet-110. They chose a strategy to pack single data repeatedly into slots in a single ciphertext, which enabled the simultaneous computation of multiple output channels.

Since secure deep SCNNs was not fast enough, the previous work provided batch inference of multiple images to achieve high-throughput by assigning each single-image inference to each CPU thread. Using k threads in batch inference the throughput can be increased k times ideally. But in actual implementation, it is not achievable due to the memory overhead of dealing with large-size ciphertext data. About $25\times$ increase of throughput was possible with 50 threads in (Lee et al., 2022a).

In this work, we show that this limitation can be broken by distributing the tasks of batch inference not only to the CPU threads but also to the slots inside the ciphertext. We propose a more batch-friendly packing, referred to as *Channel-By-Channel Packing* (CBC Packing), boosting up the throughput of secure ResNet inference by over $5\times$ compared to state-of-the-art previous works. The compactness of CBC Packing allows us to process a batch size that is not restricted by the number of threads, breaking the aforementioned throughput limitation. Also, CBC Packing improves the evaluation of Approximate ReLU (AppReLU), the most time-consuming part, by reducing the number of input ciphertexts to deal with and utilizing full-slot SIMD computation.

In CBC Packing, we encode each channel of an image to each ciphertext. CBC Packing enables computations between channels by operations between ciphertexts rather than inter-slot operations of a single ciphertext, reducing slot rotations. So it effectively reduces the cost of sending user-generated public keys, by minimizing the types of key-switching keys used in the slot rotations.

1.1. Our Contribution

We implement and test our improvements on standard ResNet-20/110 architectures on CIFAR-10 dataset with RNS-CKKS homomorphic encryption scheme using the HEaaN library[1] and our proposed CBC Packing.

- From our proposed convolution utilizing CBC Packing structure, we get $6.94\times$ less amortized time for ConvBN, from $12.28s$ to $1.77s$.
- From the compactness of CBC Packing and optimizations on ReLU evaluation, we get $4.78\times$ less amortized time for AppReLU, from $66.76s$ to $13.97s$.
- We reduce the number of required rotation keys to be $5.73\times$ lower than the previous best method (Lee et al., 2022a), from 258 to 45. This corresponds to 6.91GB, which is $37\times$ smaller in memory consumption. Details are in Appendix A.
- With the above improvements, we boost up the amortized speed of secure ResNet-20 inference by $5.04\times$, from 79.46s to 15.76s, and ResNet-110 inference by $5.20\times$, from $455.56s$ to 87.60s, without any compromise on accuracy or security, compared to state-of-the-art previous work (Lee et al., 2022a).

1.2. Technical Contribution

We use the following techniques to boost up the slot efficiency and amortized inference time of the ResNet-20 and ResNet-110 networks.

- We devise CBC Packing to pack multiple images of the same channel into a single ciphertext slot. This enables packing images 100% compactly, compared to 75% compactness (9.38% considering repetitive slots) of the previous work (Lee et al., 2022a). We utilized CBC Packing as Channel-By-Channel Convolution (CBC Conv), which gives *various* advantages on amortized runtime.

  CBC Conv reduces the number of key-switching operations (KSOs) per image by $40.37\times$ (from 1968 to 48.75) and constant multiplications (ptMult) during convolutional layers and the fully-connected (FC64) layer.
- We lowered the number of relinearization (ReLin) by 15.4% when homomorphically evaluating approximate ReLU, using the lazy baby-step giant-step (BSGS) algorithm. We also use $3.30\times$ less AppReLU per image, from 19 to 5.75, by using slots more compactly.
- We used lazy rescaling on each CBC Conv and FC64 layer to lower the overall cost.
- We manually lowered the ciphertext levels after each bootstrapping to minimize the cost and memory usage of computations.

2. Preliminaries

2.1. RNS-CKKS Fully Homomorphic Encryption

In our paper, all the operations inside the CNN architecture are in real arithmetic. CKKS (Cheon et al., 2017) naturally supports fixed-point arithmetic over real or complex numbers and is also capable of handling multiple operations

[1] Open-access on **https://heaan.it**

inside a single ciphertext in a single-instruction-multiple-data (SIMD) manner. The ciphertexts in CKKS are elements of $\mathcal{R}_Q^2$, where $\mathcal{R}_Q = \mathbb{Z}_Q(X)/\langle X^N + 1\rangle$ and N is a power-of-two integer. Each ciphertext encrypts a complex(or real) vector with $N/2$ slots as a message. The basic operations supported by CKKS are presented below. Here, $m_1, m_2 \in \mathbb{C}^{N/2}$ and $\odot$ denotes slot-wise multiplication.

$$\begin{aligned}
&\mathsf{Dec}(\mathsf{ctAdd}(\mathsf{Enc}(m_1), \mathsf{Enc}(m_2))) \simeq m_1 + m_2\\
&\mathsf{Dec}(\mathsf{ptAdd}(\mathsf{Enc}(m_1), m_2)) \simeq m_1 + m_2\\
&\mathsf{Dec}(\mathsf{ctMult}(\mathsf{Enc}(m_1), \mathsf{Enc}(m_2))) \simeq m_1 \odot m_2\\
&\mathsf{Dec}(\mathsf{ptMult}(\mathsf{Enc}(m_1), m_2)) \simeq m_1 \odot m_2\\
&\mathsf{Dec}(\mathsf{Rot}(\mathsf{Enc}(m_1), t))[\mathrm{idx}] \simeq m_1[\mathrm{idx} + t \bmod N/2]
\end{aligned}$$

The message in the ciphertext contains some small error, and this error is accumulated during the homomorphic evaluation of circuits. CKKS manages this error accumulation by introducing a leveled modulus structure. In RNS-CKKS (Cheon et al., 2018b), the ciphertext modulus Q is a product of word-size primes $Q = Q_L := \prod_{i=0}^{L} q_i$ so that the coefficients of the ciphertext polynomials can be decomposed under residue number system (RNS). Every RNS-CKKS ciphertext contains a *level* parameter l which indicates the capacity of remaining homomorphic multiplications on the ciphertext, and each multiplication (ctMult or ptMult) consumes a single level. If the ciphertext level is lower, the number of primes that represent the ciphertext decreases, and both memory and computational costs for homomorphic operations also reduce.

After we consume all the levels of a ciphertext, we can recover the modulus by performing *bootstrapping* (BTS) (Cheon et al., 2018a) operation. The number of sequential multiplications an input ciphertext goes through while evaluating a circuit is referred to as *multiplicative depth*. Without BTS, the multiplicative depth is bounded by the level of the initial ciphertext.

Key switching operation (KSO) is required for ciphertext-ciphertext multiplication (ctMult) and rotation (Rot). KSO accounts for most of the time during ctMult and Rot. These operations are much slower than addition and plaintext multiplication, which do not require KSOs. BTS contains multiple calls to ctMult and Rot, and therefore contains many KSOs. The BTS operation is considered to be the heaviest operation among all components of the RNS-CKKS scheme. Keys required for key switching has a large size, and the size depends on the parameters including degree N, modulus PQ, and a special parameter called dnum, details are in Appendix A.

2.2. Proposed Structure for Homomorphic Evaluation of ResNet on RNS-CKKS

2.2.1. Convolutional Layers and Batch Normalization

Convolution contains an operation multiplying an image vector with slided kernel vector. To accomplish this on ciphertexts, we need to appropriately pack the image and kernel into vectors in $\mathbb{C}^{N/2}$. Gazelle (Juvekar et al., 2018) proposed how to execute a *single-input-single-output* (SISO) convolution on HE schemes with SIMD properties. In the SISO convolution in Gazelle, the encrypted image is rotated instead of the kernel, and $\sum_{i=0}^{i<k\times k} \mathsf{ptMult}(\mathsf{Rot}(\mathsf{Enc}(\mathrm{image}), \mathrm{idx}_i), \mathrm{kernel}_i)$ is calculated, where $k \times k$ denotes the size of the kernel.

(Lee et al., 2022b) adapted the SISO convolution to support strided convolutions. In SISO strided convolution, whenever the image is downsampled, a gap between the pixels is generated, and the gap deteriorates the packing density of the ciphertext. (Lee et al., 2022a) named the packing method in HEAR (Kim et al., 2022) as *multiplexed packing*, and they suggested an improved version of this packing method to fill in the gaps with different channel data after strided convolutions. Our CBC Packing is easy to handle with those gaps, supporting SISO convolutions.

For batch normalization (BN), we use the standard method of integrating BN into the preceding convolutional layer (see Appendix B.3). We refer to the fused convolution as ConvBN.

2.2.2. ReLU Activation Function

Since RNS-CKKS does not support evaluating non-arithmetic operations such as ‘if’ statements, rectified linear unit (ReLU) function has to be approximated by a polynomial.

In evaluating the polynomials for ReLU, we use the odd lazy-BSGS technique from (Lee et al., 2022c) under a monomial basis. This technique lets us minimize the number of KSOs during the evaluation of the approximate ReLU function.

The computational cost and the latency of the BTS are dependent on the encryption parameters and specific choices of the algorithms for BTS. Our ReLU evaluation method is of multiplicative depth 14, and our parameter choice allows depth 9 before each BTS. Therefore, we use BTS twice per each call of ReLU. We also used the *imaginary-removing bootstrapping* (Lee et al., 2022a) for the first of the two BTS to reduce the errors, which made it possible to infer deeper SCNN.

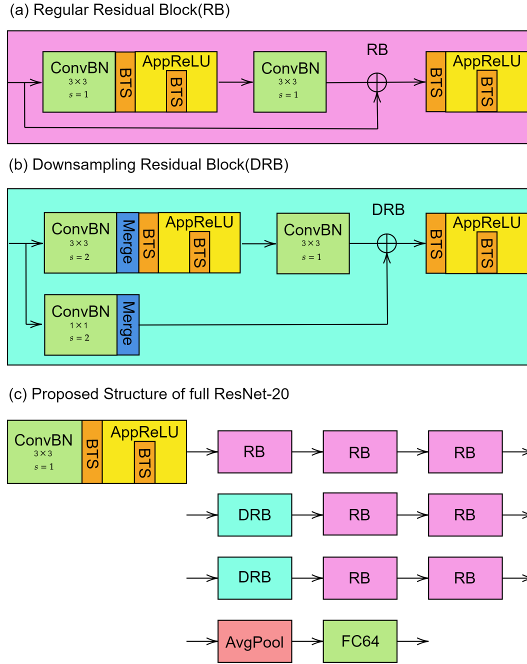

Figure 1. Overall structure of the proposed homomorphic evaluation of ResNet-20.

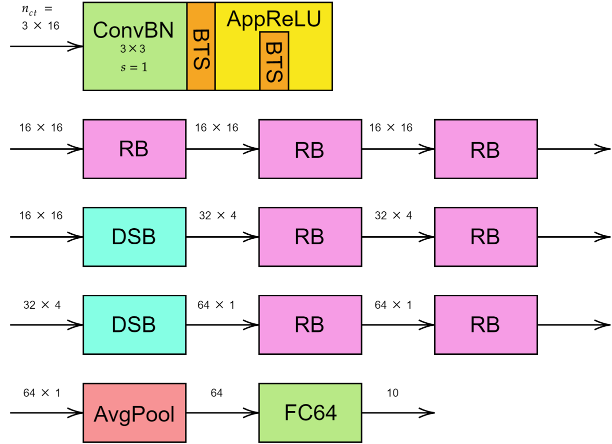

Figure 2. The number of ciphertexts n_{ct} between the components of ResNet-20 for batch inference of 512 images.

2.2.3. OVERALL STRUCTURE

Figure 1 represents the overall structure of our method to evaluate ResNet-20. We call the residual blocks RB for regular residual block (Figure 1(a)), and DRB for downsampling residual block (Figure 1(b)), respectively. ResNet of higher depth shares the same structure with ResNet-20, the only difference is the number of RBs in each row in Figure 1(c). For ResNet-110, 15 RBs are added next to the end of each row. Figure 2 illustrates the number of ciphertexts n_{ct} for the batch inference of 512 CIFAR-10 images. After each downsampling, we perform Merge to fill the gaps. Here, the total number of ciphertexts is reduced by half, which is the same as the number of AppReLU functions to be called after Merge.

3. Channel-By-Channel Packing Method

Past studies (Kim et al., 2022; Lee et al., 2022a) pack multiple channels in one ciphertext to make use of SIMD multiplications between data ciphertexts and kernel plaintexts. This can reduce the number of multiplications but dealing with ciphertexts containing multiple channel-packed data requires a lot of rotations. We must focus on the fact that these multiplications and rotations need heavy KSOs.

So, we propose a new faster method, compared to the previous methods in an amortized sense, by deleting those unnecessary additional rotations and maskings. Furthermore, those advantages make us fully utilize the SIMD and entire slots of a ciphertext.

3.1. Channel-By-Channel Packing

To reduce BTS, (Lee et al., 2022a) used Multiplexed Parallel Packing (MP Packing) so that the input channels can be compactly packed in one ciphertext. But there still can be some waste of ciphertext slots due to the number of input channels. If the input channel is not a power of 2, we have to fill in the rest of the remaining slots with zero which will be referred to as *zero channels*, since the slots of a ciphertext is always a power of 2 (Figure 4(a)). The same problem also arises for output channels. We resolve this problem by splitting every channel into separate ciphertexts.

We propose a more compact packing method, *Channel-By-Channel Packing* (CBC Packing), getting rid of those zero channels. Our proposal may also be applied to other general CNNs if only there are convolutions and downsampling structures. Furthermore, we input more ciphertexts at once when we inference. In the case of CIFAR-10 ResNet inference, 512 images compactly packed in 3×16 ciphertexts will be processed simultaneously. With 3×16 ciphertexts, we can keep the channels of the input images separated before the FC64 layer.

The *Channel-By-Channel Convolution* (CBC Conv), which utilizes the CBC Packing in convolution layers, has an advantage not only for the compactness of packing but also for reducing KSOs and ptMults in every convolution step during the entire inference procedure. We must pay attention to the number of channels which is packed in a single ciphertext.

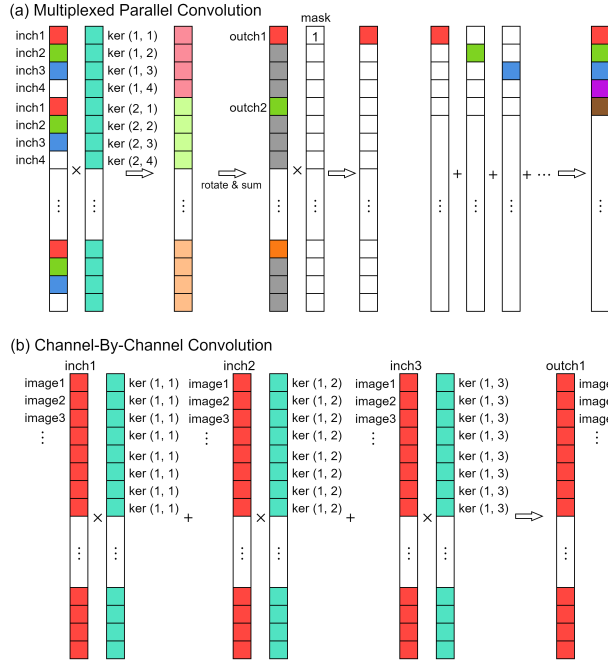

Figure 3. Comparison between Channel-By-Channel Convolution and Multiplexed Parallel Convolution. (input channel=3, stride=1)

Table 1. The amortized number of key-switching operations of the proposed architecture using Channel-by-Channel Convolution compared to the previous state-of-the-art Multiplexed Parallel Convolution (Lee et al., 2022a) in ResNet-20 per image.

Component	proposed	MP Conv
Convolution	48	1,902
Avgpool	0.75	6
FC Layer	0	60
Total	48.75	1,968

The gap denotes how many slots are away from one pixel to another adjacent pixel of a single image, inside the same channel (Figure 4). The gap is multiplied by the stride in every strided convolution. If the gap increases, the valid slots of a ciphertext become sparser without any Gathering or Merge, which will be introduced later.

Rotate-and-sum (RotSum) process for adding values inside a single ciphertext is included in every MP Conv. During the RotSum, slots that do not retain the sum are filled with dummy garbage values. Meanwhile, (Lee et al., 2022a) adopted the simultaneous computation of multiple output channels via utilizing repetitive slot (RS) packing, copying the same image data repeatedly in different slots. As can be seen in Figure 4(a), garbage values are generated by the RotSum process in MP Conv. Therefore, multiplexed

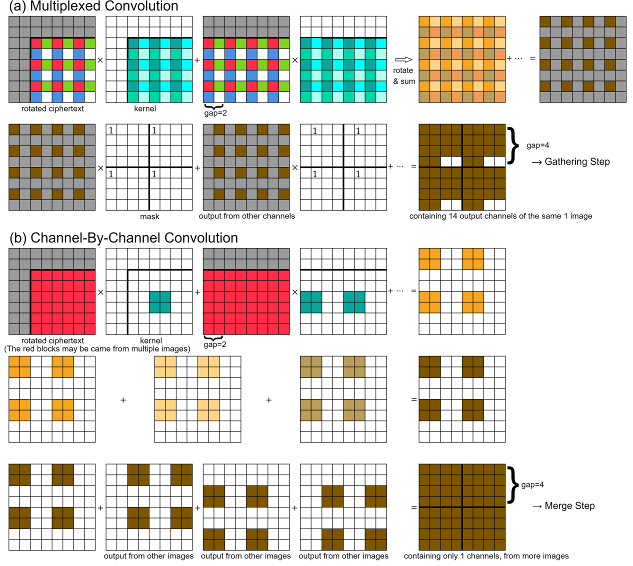

Figure 4. Comparison between Channel-By-Channel Convolution and Multiplexed Convolution. (input channel=3, output channel=14, stride=2)

packing necessitates a Gathering step that assembles sparse valid data of the same ciphertext or different ciphertexts into a more compact single ciphertext in the RS manner. This Gathering requires many maskings (ptMult), rotation, and addition. MP Conv entails the Gathering process for compact packing of output channels, even if the ciphertext passes stride 1 convolution. The proposed CBC Conv does not make garbage values during convolutions, so there is no need to gather the ciphertexts for stride 1 convolutions in Regular Residual Blocks, which is illustrated in Figure 4(b).

If we look more closely, there is another saving of rotations in convolution. If we use the Multiplexed Convolution (Kim et al., 2022), the single output channel ciphertexts should be rotated before the masking as in Figure 4(a) Gathering step. (Lee et al., 2022a) implemented output-compact convolutions (stride ≥ 1) by masking the output ciphertext when they collect valid values after multiplying stride 1 kernels with Gazelle SISO convolution. But in our Merge step, our packing method permits the use of a pre-masked version of kernel plaintexts as can be seen in Figure 4(b). This allows us to remove masking procedures and unnecessarily many additional rotations entirely in the whole ResNet structure.

Note that the operations inside RotSum cannot be parallelized. However, as in the Figure 3, CBC Conv has no RotSum (only sum exists), which allows us to benefit from multi-threading easily. And We need this simple Merge only for stride 2 convolutions in DRBs.

In short, CBC Packing evaluates a batch of multiple images

more compactly and simultaneously and excludes additional rotations, utilizing the SIMD operations to the maximum. Those optimizations result in the advantage that CBC Packing has reduced amortized KSOs from 1968 in (Lee et al., 2022a) to 48.75 in convolutions, average pooling, and fully connected layer (Table 1).

In addition, there are also advantages in the communication cost when receiving the inference output ciphertexts. We have 10 output ciphertexts containing 512 images, compared to 1 ciphertext containing 1 image. Even if we compress them into more compactly packed ciphertexts via RotSum, our proposal needs fewer rotations.

CBC Packing has some demerits in total latency when inferencing a single image. By handling much more images at a time, the total latency became longer, in spite of the much shorter amortized inference runtime. Plus, we process multiple images as one batch so inferencing a single image cannot get the same advantages of amortization as in processing multiple images at a time.

3.2. Reduced Public Key Size

The CBC Packing not only brings the higher-throughput and the compactness of ciphertext but also brings lighter public key size and lower communication cost. The required number of (public) rotation key becomes 45, compared to 258 in (Lee et al., 2022a; Snu-ccl, 2022). There are more details in Appendix A.1.

4. Further Optimizations

4.1. Lazy Rescaling

When we use the SISO convolution method (Juvekar et al., 2018) to evaluate the convolution operations, the computational bottleneck comes from the numerous ptMults between the input ciphertexts and the plaintexts encoded from the pre-trained parameters in the kernel. Throughout the whole procedure of ptMult operation, rescaling accounts for most of the runtime in ptMult and also induces additional errors. Thus it is desirable to postpone rescaling during ptMult until it becomes necessary, to reduce the total number of rescaling.

We utilize the arithmetic structure in convolution and FC64 layer to adopt the lazy rescaling technique. More precisely, we postpone the rescaling during the ptMult between the input ciphertexts and Gazelle kernel plaintexts until the summation. After performing ptMult-without-rescaling of all ciphertexts to be added, we then perform only once rescaling after the summation of them. By adopting lazy rescaling technique for every convolution layer, we reduce the computational burden when evaluating the convolution operation and achieve higher precision of the output. We can

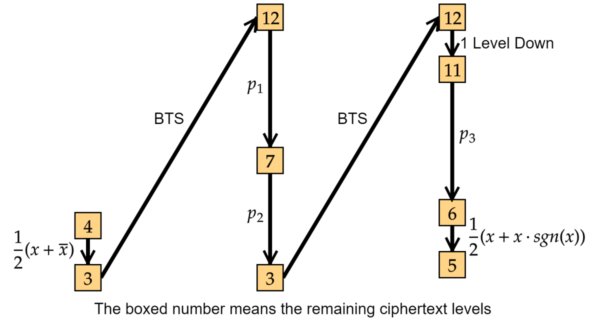

Figure 5. The whole procedure of our method during evaluation AppReLU. We maintain the ciphertext level to 5 before the residual block to optimally manage the ciphertext level. Our method reduces the number of ReLin evaluating AppReLU from 26 to 22 compared to previous work.

also apply the lazy rescaling technique in FC64 similarly.

4.2. ReLU Optimization

To evaluate ReLU activation function on encrypted data, we replace ReLU functions with approximate polynomials for ReLU functions (AppReLUs). We utilize AppReLU polynomial using a composition of minimax approximate polynomials as in (Lee et al., 2021; 2022b;a). We use the odd polynomials p_1, p_2 and p_3 of degrees 15, 15, and 27, respectively, for approximating sign function sgn to capture the 13-bit precision for AppReLU as follows:

$$\mathsf{AppReLU}(x) = x \cdot \frac{1 + sgn(x)}{2},$$
$$sgn(x) \approx (p_3 \circ p_2 \circ p_1)(x),$$
$$|\mathsf{AppReLU}(x) - \mathsf{ReLU}(x)| \leq 2^{-13} \; for \; x \in [-1, 1].$$

We note that the function AppReLU works only for input values in [-1, 1] and we confirmed that the absolute value of all the intermediate values before each ReLU does not exceed 40. The scale-invariance of the ResNet lets us scale the intermediate values down; we can multiply the weights of the very first convolution and the biases of all the convolutions by 1/40.

The performance of homomorphic evaluation of AppReLU mainly depends on the polynomial evaluation algorithm that optimized the number of ctMults, since ctMult involves the relinearization (ReLin) operation, which is the special case of KSO in ctMult. In (Lee et al., 2021), they utilize odd baby-step giant-step (BSGS) algorithm to evaluate the approximate polynomial of sign function using 25 times of ReLin, which results in 26 times usage of ReLins when evaluating AppReLU.

BSGS algorithm can be improved by adopting the lazy ReLin technique during the ctMults in the algorithm, so-called lazy baby-step giant-step (lazy-BSGS) (Lee et al., 2022c). During ctMult, the lazy-BSGS algorithm performs the tensor product of them and then delays the ReLin to be performed until ctMult with other ciphertexts. Thus,the lazy-BSGS reduces the number of ReLins in the homomorphic evaluation of approximate polynomial, which achieves both lower complexity and higher precision in computation compared to the ordinary BSGS algorithm.

We utilize the lazy-BSGS algorithm to homomorphically evaluate the AppReLU in our model. More precisely, we apply the lazy-BSGS algorithm when evaluating the polynomials p_1, p_2 and p_3 homomorphically to reduce the total number of ReLins. Since the available ciphertext levels in our parameters range from 12 to 3 with a total of 9 levels, without additional BTS. We evaluate the polynomial p_1 of degree 15 by a method of consuming 5 levels of ciphertext and then evaluate the polynomial p_2 of degree 15 by the other method of consuming 4 levels. The former requires 6 times ReLin during the evaluation, which is the more optimal and the latter requires 7 times ReLin. We then evaluate the polynomial p_3 of degree 27 by the optimal method which uses 4 levels and 8 times of ReLin. In conclusion, the total number of ReLin required in our method is 22, which reduces 26 to 22 compared to previous work. Figure 5 presents the whole procedure of our method to homomorphically evaluate AppReLU.

4.3. Ciphertext Level Management

We note that we can process the same homomorphic operations with less computational cost under lower modulus. Since the cost of bringing levels down is negligible in RNS-CKKS, we can optimize the computational complexity by maintaining the level of ciphertext as low as possible, if we can adjust the level at which the operations are executed.

Our selection of the parameters in HEaaN requires at least 3 remaining levels before BTS, and if we have more level budget than needed after BTS, we can adjust the level before the heavy computation to optimize the computational cost. The followings are parts of our methodology where we can deploy level optimization: (1) The first ConvBN: Our method for ConvBN costs only 1 level, and the ReLU layers require 4 levels for the input: 1 level for the imaginary-removing BTS and 3 levels before BTS. Originally, the plaintexts are encoded to have 12 levels by default. So, before going through the first convolution, we decrease the level of the encrypted image vectors from 12 down to 5. (2) Other ConvBNs: The second through the last ConvBNs receive outputs of ReLU as inputs. After the second bootstrapping of each ReLU layer, the remaining polynomial evaluation for ReLU consumes 6 levels. Since we have an extra 1 level not being used, we adjust the level after the BTS from 12 to 11 so that the level is consumed tightly, as shown in Figure 5. (3) AvgPool and FC64: In average pooling, we calculate the average of 8×8 data inside a ciphertext. Rather than calling ptMult for the factor $1/64$ and consuming 1 level, we merge the multiplication to the parameters of the FC64 layer. AvgPool and FC64 layer cost only 1 level. So we reduce the level after the last BTS by 5, from 12 down to 7.

5. Implementation Environments

We implement the ResNet-20/110 model on the following computing environment: $2\times$ Intel(R) Xeon(R) Gold 6248 CPU at 2.50GHz with 20 cores (total 40 cores) and 1TB RAM. We conduct our experiments on the HEaaN docker image loaded on Ubuntu-20.04. We execute inference with parameters pre-trained on the CIFAR-10 dataset which is comprised of $50,000$ images for training and $10,000$ images for testing (Krizhevsky, 2009).

5.1. Library

HEaaN (CryptoLab, ver. September 2022) is a homomorphic encryption library supporting the (RNS-)CKKS scheme and its BTS. Since the implementation environment is slightly different from the previous works (Lee et al., 2022a;b), we provide a rough comparison of the latency for basic operations in RNS-CKKS between HEaaN and SEAL library in Table 3.[2][3]

5.2. Parameters

We use a parameter preset **FGb** provided in HEaaN, which has $N = 2^{16}$ polynomial degree and secret key hamming weight 192. The total level of ciphertext provided by **FGb** parameters is 24, of which 15 are assigned to BTS, and users can use the maximum multiplicative depth 9 from 3 to 12. **FGb** parameters have a 58-bit base prime and 42-bit primes for multiplication. The total modulus size including the special primes is 1555-bit, with 128-bit security.

6. Experimental Results

This section presents the results of ResNet-20/110 architecture performance upon our architecture contrast to previous work. In (Lee et al., 2022a), they first implemented ResNet

[2]Specification details: Intel(R) Xeon(R) Silver 4114 CPU at 2.20GHz, single-thread bound to a single core; HEaaN (CryptoLab, ver. September 2022), FGb parameter ($N = 2^{16}$, base prime 58 bit, multiplication prime 42 bit); SEAL v4.1.1 (SEAL, January 2023), custom parameter ($N = 2^{16}$, base prime 51 bit, multiplication prime 46 bit); evaluated at level 5 with modulus size 268bit for HEaaN and 281-bit for SEAL except for BTS.

[3]SEAL BTS time is excerpted from (Lee et al., 2022a).

Table 2. Comparison of total and amortized inference time for ResNet-20/110. Numbers in * marks are calculated from total time and percentage data provided in (Lee et al., 2022a).

Model	ResNet-20								ResNet-110					
	(Lee et al., 2022b) (1 image, 64 threads)		(Lee et al., 2022a) (1 image, single thread)		(Lee et al., 2022a) (50 image, 50 threads)		proposed (512 image, 40 threads)		(Lee et al., 2022a) (1 image, single thread)		(Lee et al., 2022a) (50 image, 50 threads)		proposed (512 image, 40 threads)	
	runtime	amor.	runtime	amor.	runtime	amor.	runtime	amor.	runtime	amor.	-	amor.	runtime	amor.
ConvBN	-	-	351s	-	614.06s*	12.28s*	906.84s	1.77s	1,860s	-	3,190s*	63.80s*	5237.48s	10.25s
AppReLU	-	-	1,908s	-	3,337.95s*	66.76s*	7,154s	13.97s	11,411s	-	19,569s*	391.39s*	39,593s	77.33s
Avg Pool	-	-	2s	-	3.50s*	0.070s*	0.91s	0.002s	2s	-	3.43s*	0.069s*	0.50s	0.001s
FC Layer	-	-	10s	-	17.49s*	0.350s*	6.00s	0.012s	10s	-	17.15s*	0.343s*	5.77s	0.011s
Total	10,602s	-	2,271s	-	3,973s	79.46s	8,067.90s	15.76s	13,282s	-	22,778s	455.56s	44,850.24s	87.60s

Table 3. Evaluation time comparison of HEaaN and SEAL Library

	Rot	ptMult	ptMult w/o Rescale	Rescale	BTS
HEaaN	85ms	21ms	2ms	19ms	23s
SEAL	69ms	17ms	2ms	15ms	140s*

Table 4. Classification accuracy of ResNet20/110 on CIFAR-10.

	model	#test images	backbone	obtained
proposed	ResNet-20	10,000	91.98%	91.96%
	ResNet-110	10,000	93.63%	93.65%
(Lee et al., 2022a)	ResNet-20	10,000	91.52%	91.31%
	ResNet-110	10,000	93.5%	92.95%

models by using one CPU thread with one image and expanded the models to test 50 images with 50 CPU threads by allocating one CPU thread per image. On the other hand, we perform ResNet models to test 512 images at once using 40 CPU threads.

Due to our CBC Conv method, the number of KSOs required in convolutional layers of our model reduces to 48.75, which is $40.37\times$ smaller than $1,968$ in (Lee et al., 2022a) as an amortized sense. When evaluating AppReLU, we require 22 ReLins and 2 BTSs whereas 26 ReLins and one BTS is needed in (Lee et al., 2022a). Our work fully enjoys SIMD operations in RNS-CKKS to reduce the number of AppReLU operations from 19 to 5.75 in an amortized sense, which is $3.30\times$ faster.

6.1. Amortized Runtime

In ResNet-20, our implementation takes 15.76s for one image classification in an amortized sense, which is $5.02\times$ faster than 79.46s in (Lee et al., 2022a). We note that the amortized runtime in convolution layers is 1.79s, which is $6.94\times$ faster compared to previous work. We also accelerate AppReLU, Avg Pool, and FC Layer procedures in our model by $4.78\times$, $39.22\times$, and $29.83\times$, respectively, faster compared to previous work. We also implement ResNet-110 to classify 10,000 images, which results in a $5.20\times$ reduction in amortized runtime.

6.2. Accuracy

We present our classification accuracies for ResNet-20/110 as in Table 4. We confirm that for the both case of ResNet-20 and ResNet-110, our classification of ResNet-20 and ResNet-110 has almost the same accuracy compared to the backbone. More precisely, we note that there is only 0.02% accuracy loss in the classification of ResNet-20 and in the case of ResNet-110, accuracy increases by 0.02%, compared to the backbone.

6.3. Memory Management

Our implementation for ResNet models has RAM usage by 370.28GB at maximum, compared to the 384GB and 512GB for 1 image and 50 images, respectively, in (Snu-ccl, 2022). We note that most of our RAM usage comes from the storage of plaintexts that encodes pre-trained parameters. Thanks to our efficient CBC Packing method, the memory usage for rotation keys is 6.91GB, which is relatively small in the whole RAM usage.

7. Conclusions

We resolved the bottleneck for the CNN models by achieving high-throughput inference in a SIMD manner via our CBC packing algorithm. From the perspective of batch inference, we significantly reduce the amortized runtime of inference, compared to the previous state-of-the-art work (Lee et al., 2022a). We also increase the practicality of our PPML model by downsizing key size for communication with achieving almost the same accuracy compared to the backbone models. We expect that our CBC packing method can be applied to other CNN networks considering their number of downsampling layers, which results in increased efficacy.

References

Al Badawi, A., Jin, C., Lin, J., Mun, C. F., Jie, S. J., Tan, B. H. M., Nan, X., Aung, K. M. M., and Chandrasekhar, V. R. Towards the alexnet moment for homomorphic encryption: HCNN, the first homomorphic cnn on encrypted data with gpus. *IEEE Transactions on Emerging Topics in Computing*, 9(3):1330–1343, 2021. doi: 10.1109/TETC.2020.3014636.

AmazonAWS. Amazon personalize - developer guide; Getting batch recommendations and user segments. https://docs.aws.amazon.com/personalize/latest/dg/getting-started.html, 2023.

Bajard, J.-C., Eynard, J., Hasan, M. A., and Zucca, V. A full RNS variant of FV like somewhat homomorphic encryption schemes. In *Lecture Notes in Computer Science*, pp. 423–442. Springer International Publishing, 2017. doi: 10.1007/978-3-319-69453-5_23. URL https://doi.org/10.1007/978-3-319-69453-5_23.

Cheon, J. H., Kim, A., Kim, M., and Song, Y. Homomorphic encryption for arithmetic of approximate numbers. In Takagi, T. and Peyrin, T. (eds.), *Advances in Cryptology – ASIACRYPT 2017*, pp. 409–437, Cham, 2017. Springer International Publishing. ISBN 978-3-319-70694-8.

Cheon, J. H., Han, K., Kim, A., Kim, M., and Song, Y. Bootstrapping for approximate homomorphic encryption. In Nielsen, J. B. and Rijmen, V. (eds.), *Advances in Cryptology – EUROCRYPT 2018*, pp. 360–384, Cham, 2018a. Springer International Publishing. ISBN 978-3-319-78381-9.

Cheon, J. H., Han, K., Kim, A., Kim, M., and Song, Y. A full RNS variant of approximate homomorphic encryption. *Selected areas in cryptography : ... annual international workshop, SAC ... proceedings. SAC*, 11349:347–368, 2018b.

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.

CryptoLab. HEaaN. https://heaan.it, ver. September 2022.

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.

Deng, J., Dong, W., Socher, R., Li, L.-J., Li, K., and Fei-Fei, L. Imagenet: A large-scale hierarchical image database. In *2009 IEEE Conference on Computer Vision and Pattern Recognition*, pp. 248–255, 2009. doi: 10.1109/CVPR.2009.5206848.

Gentry, C., Halevi, S., and Smart, N. P. Homomorphic evaluation of the AES circuit. In *Lecture Notes in Computer Science*, pp. 850–867. Springer Berlin Heidelberg, 2012. doi: 10.1007/978-3-642-32009-5_49. URL https://doi.org/10.1007/978-3-642-32009-5_49.

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 Balcan, M. F. and Weinberger, K. Q. (eds.), *Proceedings of The 33rd International Conference on Machine Learning*, volume 48 of *Proceedings of Machine Learning Research*, pp. 201–210, New York, New York, USA, 20–22 Jun 2016. PMLR. URL https://proceedings.mlr.press/v48/gilad-bachrach16.html.

Han, K. and Ki, D. Better bootstrapping for approximate homomorphic encryption. In *Topics in Cryptology – CT-RSA 2020*, pp. 364–390. Springer International Publishing, 2020. doi: 10.1007/978-3-030-40186-3_16. URL https://doi.org/10.1007/978-3-030-40186-3_16.

He, K., Zhang, X., Ren, S., and Sun, J. Deep residual learning for image recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2016, Las Vegas, NV, USA, June 27-30, 2016*, pp. 770–778. IEEE Computer Society, 2016. doi: 10.1109/CVPR.2016.90. URL https://doi.org/10.1109/CVPR.2016.90.

Juvekar, C., Vaikuntanathan, V., and Chandrakasan, A. GAZELLE: A low latency framework for secure neural network inference. In *Proceedings of the 27th USENIX Conference on Security Symposium*, SEC'18, pp. 1651–1668, USA, 2018. USENIX Association. ISBN 9781931971461.

Kim, M., Jiang, X., Lauter, K., Ismayilzada, E., and Shams, S. Secure human action recognition by encrypted neural network inference. *Nature Communications*, 13(1), August 2022. doi: 10.1038/s41467-022-32168-5. URL https://doi.org/10.1038/s41467-022-32168-5.

Krizhevsky, A. Learning multiple layers of features from tiny images. pp. 32–33, 2009. URL https://www.cs.toronto.edu/~kriz/learning-features-2009-TR.pdf.

Lee, E., Lee, J.-W., Lee, J., Kim, Y.-S., Kim, Y., No, J.-S., and Choi, W. Low-complexity deep convolutional neural

networks on fully homomorphic encryption using multiplexed parallel convolutions. In Chaudhuri, K., Jegelka, S., Song, L., Szepesvari, C., Niu, G., and Sabato, S. (eds.), *Proceedings of the 39th International Conference on Machine Learning*, volume 162 of *Proceedings of Machine Learning Research*, pp. 12403–12422. PMLR, 17–23 Jul 2022a. URL https://proceedings.mlr.press/v162/lee22e.html.

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*, 2021.

Lee, J.-W., Kang, H., Lee, Y., Choi, W., Eom, J., Deryabin, M., Lee, E., Lee, J., Yoo, D., Kim, Y.-S., and No, J.-S. Privacy-preserving machine learning with fully homomorphic encryption for deep neural network. *IEEE Access*, 10:30039–30054, 2022b. doi: 10.1109/ACCESS.2022.3159694.

Lee, Y., Lee, J.-W., Kim, Y.-S., Kim, Y., No, J.-S., and Kang, H. High-precision bootstrapping for approximate homomorphic encryption by error variance minimization. In Dunkelman, O. and Dziembowski, S. (eds.), *Advances in Cryptology – EUROCRYPT 2022*, pp. 551–580, Cham, 2022c. Springer International Publishing. ISBN 978-3-031-06944-4.

SEAL. Microsoft seal (release 4.1.1). https://github.com/Microsoft/SEAL, January 2023.

Snu-ccl. FHE-MP-CNN. https://github.com/snu-ccl/FHE-MP-CNN, 2022.

A. On the Cost Of Key Switching and the Size of the Key-Switching Keys

In this section, we describe how the modulus and dnum parameter affect key-switching operations in RNS-CKKS.

Let us set $Q = Q_L = \prod_{i=0}^{L} q_i$ as a product of $(L+1)$ pairwise coprime integers and write $Q_l = \prod_{i=0}^{l} q_i$. The ciphertexts consist of polynomials that are in the ring $\mathcal{R}_{Q_l}$, and the key-switching keys are generated in the largest modulus PQ. Here, P is called a *temporary modulus* and used in error control in KSOs. The security of the parameter is affected by the polynomial degree N and the largest modulus size $\log PQ$. For a fixed N and security parameter, the maximum value of $\log PQ$ is determined. Here, The size and composition of $P = \prod_{j=0}^{L'} p_j$ are determined by the dnum value.

In the key switching process of RNS-CKKS, we have to increase the modulus of the ciphertext from Q_l to PQ_l. To represent the values modulo q_i to values modulo p_j, we need to recover the original value from RNS decomposition, by using the Chinese Remainder Theorem (CRT). The CRT introduces an error that is in worst-case proportional to the number of primes in Q. We can control this error by the *temporary modulus technique* (Gentry et al., 2012), and the error term is divided by P in the key-switching process. In the original RNS-CKKS paper (Cheon et al., 2018b), in order to control the error in the key switching process, they had to set the size of P and Q about the same, and the number of available levels L was low.

The work of (Han & Ki, 2020) used the *gadget decomposition method* (Bajard et al., 2017) and introduced the notion of *decomposition number* dnum, to lower the size of P at the cost of the size of the key-switching keys and the computational complexity of the key-switching process. dnum roughly means that we decompose the RNS basis $\{q_0, q_1, \ldots, q_L\}$ into $d = \mathsf{dnum}$ disjoint sub-bases so that the error from CRT is proportional to the size of each sub-basis. The size of P is then required to be larger than the product of primes in each of the sub-basis.

However, In order to switch to the RNS representations for each sub-basis, the total cost of the key-switching process increases, and is of $O((\mathsf{dnum}+1)L)$. Also, the size of the key-switching keys, including the relinearization key and rotation keys, is also proportional to dnum.

A.1. Details on the Parameters used in HEaaN and SEAL Implementations and Communication Costs

Our choice of parameter preset provided in HEaaN (CryptoLab, ver. September 2022) named FGb uses $\mathsf{dnum} = 5$, with P consisting of 5 primes and Q consisting of 25 primes. A single key-switching key size is about 157.3MB for this parameter.

SEAL library supports the 'full dnum' scheme, using P as a single prime. In the work of (Lee et al., 2022a), the parameter choice was 51bit base prime q_0 and 51bit P, and 46bit multiplication primes. The total number of primes is 33, and $\mathsf{dnum} = 32$. Therefore, we can estimate that each key-switching key is of size 157.3MB $\times 32/5 \simeq 1007$MB.

We use the (left) rotation key set of size 45: $\{\pm1, \pm2, 3, \pm4, 5, 6, 7, \pm8, \pm16, \pm24, \pm32, 33, \pm64, 66, 96, \pm128, 160, 192, 224, \pm256, \pm512, \pm768, \pm1024, 2048, 3072, 4096, 5120, 6144, 7168, \pm8192, 16384\}$, of total size about 6.91GB.

The implementation (Snu-ccl, 2022) of the work (Lee et al., 2022a) uses a rotation key set of size 258. The estimated size of all the rotation keys is about 254GB.

For 512 images, our method packs all the images into 48 ciphertexts at level 5, having a total size of 0.29GB. In the method of (Lee et al., 2022a), each ciphertext includes a single image, so 512 ciphertexts at level 31 needs to be sent, which adds up to 15.25GB.

We also note without details that all the communication costs can be halved by using standard compression techniques for RLWE ciphertexts by using seed-generated polynomials in encryptions and sending seeds instead of polynomials.

B. Convolution Algorithms for the Proposed Architecture

B.1. Image Packing

Here, we explain Image Packing algorithm. Because our inference's first step is an image packing, this algorithm is a starting point of our task. This receives imagevector and returns an encrypted image vector. $insert$(a,b) function is a function that inserts elements from a to b at the end of the specified function. And, REncode(α) function is an encoding function that encodes the real part as α and the imaginary part as zero. Since we use the only real part of our ciphertext, REncode is the appropriate encoding method. Finally, each element of the result vector consists of its corresponding message ciphertext. More concrete pseudo-algorithm in Algorithm1.

B.2. Kernel Packing

In this section, we present algorithms to pack kernel values (Algorithm3 for multiplicands and Algorithm2 for summands) into a single plaintext. The algorithm can be repeatedly used in all kinds of convolutions. We can use the same algorithm also for the FC64 layer as KernelPackingMultiplicands(img, m, $g = 1$, $s = 1$, $idx_r = 0$, $idx_c = 0$, l). image_size is 32 for CIFAR-10 images. For 1×1 convolution, the indices (row_index, column_index) = (0,0). For 3×3 convolution, the indices (row_index, column_index) $\in \{-1, 0, 1\} \times \{-1, 0, 1\}$.

B.3. Convolution on Packed Ciphertext

B.3.1. Batch Normalization

For the inference of CNN with batch normalization (BN), The BN layer can be integrated into the preceding convolution layer. For the BN layer and Conv layer without bias $y = \frac{x-\mu}{\sqrt{\nu+\epsilon}} * \gamma + \beta$ and $y = Wx$. We have the fused weight $\overline{W}$ and the fused bias $\bar{b}$ satisfying $y = \overline{W}x + \bar{b} = \frac{\gamma}{\sqrt{\nu+\epsilon}}Wx + (\beta - \frac{\mu\gamma}{\sqrt{\nu+\epsilon}})$. Therefore, we can consider the integrated convolution as having altered weights and biases.

B.3.2. Convolution

We present our convolution algorithm as a part of the CBC Conv. Actually, the special architecture of CBC Conv consists of its pre-masked kernels and Merge step. So the convolution in CBC Conv just follows the description of SISO convolution method as in Gazelle (Juvekar et al., 2018).

C. Approximate ReLU Evaluation

C.1. Lazy Baby-Step Ginat-Step alogirhtm

Here, we explain a lazy Baby-Step Giant-Step (lazy BSGS) algorithm. The original BSGS algorithm is a well-known evaluation algorithm. Although we will explain Odd lazy-BSGS algorithm with a monimial base, if we change the monomial base to another base then we can also use this algorithm in the same manner with only different base elements. In some cases, it is better to use another base element than a power base approximation.

In our cases, we use odd version of lazy-BSGS algorithm. Odd version is not different from the original ones. The only difference is that we use only evaluate with odd degree parts. Our odd lazy-BSGS consists of three parts, which are OddSetUp,OddBabyStep, and OddGiantStep. OddSetUp is a section that pre-calculate encrypted base elements with the given ciphertext. OddBabyStep is an evaluation part for the given encrypted base elements, which are constructed in SetUp if the evaluation polynomial degree is less than something. Giant Step is an evaluation part using a division algorithm with an already-made base element. As stated before, The lazy BSGS is similar to BSGS algorithm. The difference is that we postpone ReLin timing. There is an additional step after multiplication in this algorithm. In general, we add already relinearized objects. If we add two elements without ReLin then we can reserve one ReLin for each multiplication. By using this idea, we can reserve many ReLin in the whole evaluation. This is a crucial idea in the lazy sense BSGS. **Algorithm6** gives us a pseudo-code of our algorithm. we need a ciphertext ct and a polynomial we want to evaluate. Corresponding to its polynomial degree and ct, we can evaluate a polynomial with ct.

In BSGS algorithm, we have to choose two parameters k,l where l is the smallest positive integer and k is an even number satisfying $2^l k > n$, where n is the degree of the given polynomial. Because there is no restricted choosing method of k,l, we can take such parameters corresponding to our cases. By using this choice, we can make a more useful evaluation algorithm in our inference. More detailed story about this, we will explain in the next subsection.

Algorithm 1 ImagePacking

Input: Given image vector **img**
Output: encrypted image vector
procedure ImagePacking(**img**)
  $num \leftarrow$ slot_number
  for $i = 0$; $i < 32$; $i = i + 1$
    input1 $\leftarrow insert($ **img**, $\ldots$, $\mathbf{img}_{3072*i+1024}$)
    input2 $\leftarrow insert($ $\mathbf{img}_{3072*i+1024}$, $\ldots$, $\mathbf{img}_{3072*i+2048}$)
    input3 $\leftarrow insert($ $\mathbf{img}_{3072*i+2048}$, $\ldots$, $\mathbf{img}_{3072*i+3072}$)
  end for
  for $i = 0$; $i < num$; $i = i + 1$
    $\mathbf{msg1}_i \leftarrow$ REncode($\mathbf{input1}_i$)
    $\mathbf{msg2}_i \leftarrow$ REncode($\mathbf{input2}_i$)
    $\mathbf{msg3}_i \leftarrow$ REncode($\mathbf{input3}_i$)
  end for
  for $i = 0$; $i < 3$; $i = i + 1$
    $\mathbf{ct}_{res,i} \leftarrow$ Enc(**msgi**)
  end for
return $\mathbf{ct}_{res}$
end procedure

Algorithm 2 KernelPackingSummands

Input: kernel_value = m, level = l
Output: summand for any convolution.
procedure KernelPackingSummands(m, l)
Initialize message vector msg(num_slots)
Initialize plaintext pt(num_slots)
for $i = 0$;i <num_slots;$i = i + 1$
  $msg[i] = m$
end for
$pt = \mathsf{Encode}(msg, l)$
return pt

Algorithm 3 KernelPackingMultiplicands (in case of 1×1 and 3×3 kernel)

Input: image_size = img, kernel_value = m, gap = g, stride = s, row_index = idx_r, column_index = idx_c, level = l
Output: multiplicand for stride s convolution with gap g.
procedure KernelPackingMultiplicands(img, m, g , s, idx_r, idx_c, l)
Initialize message vector msg(num_slots)
Initialize plaintext pt(num_slots)
if s == 1
  for $i = 0$;i <num_slots;$i = i + 1$
    $msg[i]$ = m
  end for
else
  for $i = 0$; i <num_slots$/(img)^2$; $i = i + 1$
    for $j = 0$; $j < img$; $j = j + 1$
      for $k = 0$; $k < img$; $k = k + 1$
        if $m\%2 * g * s < g * s$ && $l\%2 * g * s < g * s$
          $msg[i * (img)^2 + j * img + k] = m$
        else
          $msg[i * (img)^2 + j * img + k] = 0$
      end for
    end for
  end for
end if
if idx_r == -1
  for $i = 0$; i <num_slots$/(img)^2$; $i = i + 1$
    for $j = 0$; $j < img * g$; $j = j + 1$
      $msg[i * (img)^2 + j]$=0
    end for
  end for
else if idx_r == 1
  for $i = 0$; i <num_slots$/(img)^2$; $i = i + 1$
    for $j = 0$; $j < img * g$; $j = j + 1$
      $msg[i * (img)^2 + ((img)^2 - img * g) + j]$=0
    end for
  end for
end if
if idx_c == -1
  for $i = 0$; i <num_slots$/(img)^2$; $i = i + 1$
    for $j = 0$; $j < img$; $j = j + 1$
      for $k = 0$; $k < g$; $k = k + 1$
      $msg[i * (img)^2 + j * img + k]$=0
      end for
    end for
  end for
else if idx_c == 1
  for $i = 0$; i <num_slots$/(img)^2$; $i = i + 1$
    for $j = 0$; $j < img$; $j = j + 1$
      for $k = 0$; $k < g$; $k = k + 1$
      $msg[i * (img)^2 + j * img + (img - k)]$=0
      end for
    end for
  end for
end if
$pt = \mathsf{Encode}(msg, l)$
return pt

Algorithm 4 ConvBN (in case of 1×1, 3×3 kernel)

```
Input: imagesize = img , gap = g , stride = s, input_channel = in_ch , output_channel = out_ch , ciphertext bundle = ct ,
3-dim output_kernel_bundle = kernel, corresponding summand vector = sum.
Output: convolutioned ciphertext vecotor ct_res adding summands.
procedure ConvBN(img, g , s, in_ch, out_ch, ct,kernel ,sum)
ks ← size(kernel_{0,0})
If ks = 9
  for i = 0 ; i < in_ch ; i = i + 1
    for j = 0 ; j < 3 ; j = j + 1
      for k = 0 ; k < 3 ; k = k + 1
        ct_rot^{i,3*j+k} ← Rot(ct_i , −(j − 1) * img * g − (k − 1) * g)
      end for
    end for
  end for
  for i = 0 ; i < out_ch ; i = i + 1
    for j = 0 ; j < in_ch ; j = j + 1
      ct_temp1 ← multWithoutRescale (ct_rot^{i,0} , kernel_{i,j,0})
      for k = 1 ; k < 9 ; k = k + 1
        ct_temp2 ← multWithoutRescale (ct_rot^{i,k} , kernel_{i,j,k})
        ct_temp1 ← ctAdd(ct_temp1,ct_temp2)
      end for
    if i = 0
      ct_{res,i} ← ct_temp1
    else
      ct_{res,i} ← ctAdd( ct_temp1, ct_{res,i})
    end if
    end for
    ct_{res,i} ← ReScale(ct_{res,i})
  end for
else
  for i = 0 ; i < out_ch ; i = i + 1
    for j = 0 ; j < in_ch ; j = j + 1
      ct_temp ← multWithoutRescale(ct_{res,j} , kernel_{i,j,0})
      If i = 0
        ct_{res,i} ← ct_temp
      else
        ct_{res,i} ← ctAdd(ct_temp, ct_{res,i})
      end if
    end for
    ct_{res,i} ← ReScale( ct_{res,i})
  end for
ct_res ← ptAdd(ct_res,sum)
return ct_res
end procedure
```

Algorithm 6 Odd lazy Baby-Step Giant-Step (Lee et al., 2022c)

Input: A ciphertext ct of x and polynomial degree n with coefficients $\{a_i\}$.
Output: An encryption of given polynomial $p(x)$.
Let k be an even integer and l is the smallest postitive integer satisfying $2^l k > n$.
procedure OddSetUP(ct, k, l)
  for $i = 1$; $i \leq log_2(k)$; $i = i + 1$
    $ct_{2i} \leftarrow$ ctMult(ct_i,ct_i)
  end for
  for $i = 1$; $i < k/2$; $i = i + 1$
    α := $\lfloor log_2(2i+1) \rfloor$
    j := $2^{\alpha-1}$
    $ct_{2i+1} \leftarrow$ ctMult(ct_α, ct_{i-j})
  end for
  for $i = 1$; $i < l$; $i = i + 1$
    $ct_{2^i k} \leftarrow$ ctMult($ct_{2^{i-1}k}$, $ct_{2^{i-1}k)}$
  end for
  return $\{ct_i\}$
end procedure

procedure OddBabyStep($\{a_i\}$, $\{ct_i\}$, k)
  $ct_{res} \leftarrow$ CMult(ct_0 , a_1)
  for $i = 1$; $i < k/2$ and $2 < k$; $i = i + 1$
    $ct_{res} \leftarrow$ ctAdd (ct_{res} , CMult (ct_i, a_{2i+1})
  end for
  return $\sum_i a_i ct_i$
end procedure

procedure OddGiantStep($\{a_i\}$, $\{ct_i\}$, k, l)
if $0 < \lfloor log_2(n/k) \rfloor$ **then**
  OddBabyStep($\{a_i\}$, $\{ct_i\}$, k)
else
  m := $k * log_2(n/k)$
  r := $\{a_0, ..., a_{m-1}\}$
  q := $\{a_m, ..., a_n\}$
  if $(n + 1) - m \leq k$ **then**
    $ct_{ql} \leftarrow$ OddGiantStep(**q**, $\{ct_{2^i k}\}$,k, l)
  else
    $ct_q \leftarrow$ OddGiantStep(**q** , $\{ct_{2^i k}\}$,k, l)
    $ct_{ql} \leftarrow$ ReLin(ct_q)
  end if
  $ct_r \leftarrow$ OddGiantStep(**r** , $\{ct_{2^i k}\}$, k ,l)
  $ct_{res} \leftarrow$ multWithoutRelin(ct_α, ct_{ql})
  $ct_{res} \leftarrow$ ctAdd(ct_{res} , ct_r)
end if
return ct_{res}
end procedure

C.2. ApproxReLU algorithm

Here we explain our ApproxReLU algorithm with a basic polynomial evaluation algorithm. Because OddGiantStep makes some part of polynomial evaluation with the given its coefficients, we can evaluate by using this algorithm. Since we use multWithOutRelin operation in Lazy BSGS and we have to match between objects to operate some particular ones in HEAAN, we have to ReLin before the ending of the evaluation algorithm. As stated before, we can save 4 ReLin times in

one AppReLU.

AppReLU algorithm in $\mathbf{Algoritm7}$ is our polynomial evaluation algorithm with the composition of low-degree polynomials. Since we only use the real part of the message, if we encrypt this message and implement an inferences model, errors will stack on the imaginary part and this makes our model fall into the catastrophic situation((Lee et al., 2022a). So, we use immaginary-BTS the first BTS in AppReLU. After that, we evaluate each polynomial sequentially. As stated in $\mathbf{Algoritm6}$, we need to choose two parameters, k and l corresponding to each polynomial degree. We can handle the trade-off between the number of ReLin and consuming depth. That is, if we want to reduce more consuming depth then, we can choose such parameters satisfying our needs. From this property, we can save more ReLin times than the original lazy BSGS case.

Algorithm 7 ApproxReLU

Input: Encrypted basis elements , given polynomial degree n and its coefficients
Output: An evaluating polynomial with the given ciphertext.
Let k be an even integer and l is the smallest positive integer satisfying $2^l k > n$.

procedure OddEvalPoly(ct, $\{a_i\}$, k, l)
GS_basis ← OddSetUP(ct, k, l)
ct_{poly} ← OddGiantStep($\{a_i\}$, GS_basis, k, l)
$p(ct)$ ← ReLin(ct_{poly})
return $p(ct)$
end procedure

procedure ApproxReLU(ct, $\{a_i\}$,$\{b_i\}$,$\{c_i\}$)
Note that both k_i and l_i are selected corresponding to each polynomial degree and trade-off(need to explain in the above sentence).
ct_{img} ← imaginary_BTS(ct)
$p_1(ct)$ ← OddEvalPoly(ct_{img},$\{a_i\}$, k_1,l_1)
$p_2(p_1(ct))$ ← OddEvalPoly($p_1(ct)$,$\{b_i\}$, k_2,l_2)
$p_2(p_1(ct))$ ← BTS($p_2(p_1(ct))$)
$p_3(p_2(p_1(ct)))$ ← OddEvalPoly($p_2(p_1(ct))$,$\{c_i\}$,k_3,l_3)
return $p_3(p_2(p_1(ct)))$
end procedure

Algorithm 5 MergeCiphertext

Input: ciphertext vector = **ct**, imagesize = img , stride = s and gap = g
Output: rotated sum ciphertext $\mathbf{ct}_{sum}$
Note that $\mathbf{ct}_{res}$ is an initialized ciphertext vector and $\mathbf{ct}_{sum}$ is an initialized ciphertext which is an encryption of zero message.
procedure MergeCiphertext(**ct**, img , s , g)
  for $i = 0$; $i < s$; $i = i + 1$
    for $j = 0$; $j < s$; $j = j + 1$
      $\mathbf{ct}_{res}^{s*i+j} \leftarrow \mathsf{Rot}(\mathbf{ct}^{s*i+j}, -(img * g * i) - j * g)$
      $\mathbf{ct}_{sum} \leftarrow \mathsf{ctAdd}(\mathbf{ct}_{res}^{s*i+j}, \mathbf{ct}_{sum})$
    end for
  end for
return $\mathbf{ct}_{sum}$
end procedure