DEIM Forum 2023 5b-9-2

# Performance Comparison of Homomorphic Encrypted Convolutional Neural Network Inference between Microsoft SEAL and OpenFHE

Haoyun ZHU†, Takuya SUZUKI††, Houtao HUANG††, and Hayato YAMANA†††

† School of Fundamental Science and Engineering, Waseda University,
3-4-1 Okubo, Shinjuku-ku, Tokyo, 169-8555 Japan
†† Graduate School of Fundamental Science and Engineering, Waseda University,
3-4-1 Okubo, Shinjuku-ku, Tokyo, 169-8555 Japan
††† Faculty of Science and Engineering, Waseda University, 3-4-1 Okubo, Shinjuku-ku, Tokyo, 169-8555 Japan
E-mail: †shuhaoyun@akane.waseda.jp, ††{t-suzuki,tao991,yamana}@yama.info.waseda.ac.jp

**Abstract** Homomorphic encryption (HE) is a promising way to preserve data privacy even during their calculation because HE enables evaluation over ciphertexts. After the first proposal of fully homomorphic encryption (FHE) enabling arbitrary numbers of evaluation over ciphertexts by Craig Gentry in 2009, many HE libraries have appeared that we can use free of charge. Meanwhile, it is not easy to choose the best-fit library for beginners of HE. This paper shows an insight for selecting the library targeting Microsoft SEAL and OpenFHE, two popular HE libraries. We implemented a convolutional neural network (CNN) inference application with these two libraries to compare the inference latency and accuracy. Note that machine learning methods over HE are one of the killer applications of HE. Our experimental results showed that the CNN w/ SEAL had smaller latency, less than 50%, compared to the CNN w/ OpenFHE, while the accuracy was comparable between w/ SEAL and w/ OpenFHE, but slightly better w/ SEAL. In terms of ease of use, OpenFHE is superior because programmers do not have to consider the rescaling required during the calculation. In contrast, adopting automatic rescaling incurs 5x more latency.

**Key words** Privacy-preserving machine learning, Homomorphic encryption, Microsoft SEAL, OpenFHE

## 1 Introduction

In 2009, Craig Gentry [1] first proposed fully homomorphic encryption (FHE), enabling arbitrary numbers of evaluations (additions and multiplications) over ciphertexts. After his proposal, the number of papers related to homomorphic encryption (HE) is increasing every year, benefitting from many kinds of HE libraries, including HElib(注1), TFHE(注2), PALISADE(注3), HEAAN(注4), SEAL(注5), and OpenFHE(注6).

Meanwhile, choosing the best-fit HE library for beginners is difficult; thus, this paper tackles to provide insight into selecting the library targeting Microsoft SEAL [2] and OpenFHE [3], which are two popular FHE libraries. Microsoft SEAL is one of the most widely used HE libraries adopting various HE schemes such as the Brakerski-Fan-Vercauteren (BFV) scheme [4], the BrakerskiGentry-Vaikuntanathan (BGV) scheme [5], and the Cheon-KimKim-Song (CKKS) scheme [6], [7], [8]. Another programmer-friendly new HE library is OpenFHE, a new open-source HE library released in July 2022. OpenFHE combines previously proposed various HE-related schemes, which have been collaboratively constructed by over 26 HE-related research organizations, with merging PALISADE, HElib, and HEAAN. Note that OpenFHE focuses more on the usability of the scheme to promote a wide range of HE usage because the theory of HE and its programming with HE is different from usual programming, which limits the wide range of usages.

This paper compares two important aspects, i.e., latency and accuracy, when comparing the two HE libraries. In the comparison, we implement a convolutional neural network (CNN) inference application to compare the libraries because machine learning is one of the killer HE applications of HE. For example, machine learning as a service (MLaaS) has been gaining more and more attention from businesses as a series of services that provide machine learning tools as a part of cloud computing services. However, MLaaS can be accompanied by significant risks when using users' private data. To preserve the privacy of users' private data, MLaaS with homomorphic encryption (HE) is one of the most reliable solutions. With HE, a user can send the user's ciphertexts to a third party and allows the third party to run a machine learning model, such as a neural network, with the user's ciphertexts without revealing any

(注1): https://github.com/homenc/HElib
(注2): https://github.com/tfhe/tfhe
(注3): https://palisade-crypto.org/
(注4): https://github.com/snucrypto/HEAAN
(注5): https://github.com/microsoft/SEAL
(注6): https://github.com/openfheorg/openfhe-development

underlying private information.

In this paper, we give an insight into selecting a HE library for readers when they want to adopt HE by comparing the latency and the accuracy of CNN inference with SEAL and OpenFHE.

This paper is organized as follows. In Section 2, we explain the background of HE and HE libraries. Section 3 summarizes related work comparing HE libraries and homomorphic encrypted CNN inference. Section 4 shows our performance comparison method. In Section 5, we report and discuss the results of the experiments. Finally, we conclude this paper in Section 6.

## 2 Background

### 2.1 Overview of Homomorphic Encryption (HE)

HE enables performing computations over ciphertexts without using a secret key. HE is classified into the following four types.

- Partially HE (PHE): PHE can be further classified into two types: additive HE which supporting only homomorphic addition; and multiplicative HE supporting only homomorphic multiplications.
- Somewhat HE (SHE): SHE supports several times of homomorphic additions and a few times homomorphic multiplications on a ciphertext; thus, SHE cannot evaluate homomorphic operations with high multiplicative depth.
- Leveled HE (LHE): LHE supports several homomorphic additions and the pre-defined times of homomorphic multiplications, called level as an HE parameter. LHE can evaluate homomorphic operations with higher multiplicative depth than SHE; however, LHE still cannot evaluate homomorphic operations with undefined multiplicative depth, such as training in machine learning. In addition, the higher the level is, the higher the computational cost and the larger the ciphertext size are; therefore, the level should be as small as possible.
- Fully HE (FHE): FHE can perform an arbitrary number of both homomorphic additions and homomorphic multiplications on a ciphertext by applying bootstrap, which takes a long execution time. FHE is constructed by adopting bootstrapping to SHE or LHE; i.e., if bootstrapping is not implemented or executed, the scheme can be called SHE or LHE.

One of the suitable HE schemes for homomorphic encrypted machine learning is the homomorphic encryption for the arithmetic of approximate numbers (HEAAN) scheme, which is also known as the Cheon-Kim-Kim-Song (CKKS) scheme. The first CKKS scheme was proposed by Cheon et al. in 2017 [6] as LHE. Because the CKKS scheme approximates floating-point numbers to fixed-point numbers and adds noise into ciphertexts, the decryption result of a ciphertext is different from the original plaintext. However, the error is sufficiently small by selecting HE parameters carefully; i.e., the error can be negligible. Furthermore, Cheon et al. [8] proposed the residue number system (RNS) variant of the CKKS scheme (called FullRNS-CKKS scheme), which is more efficient than the original CKKS scheme. Although the CKKS scheme is an LHE-based-FHE scheme [7], we focus on the FullRNS-CKKS scheme without adopting bootstrapping because the multiplicative depth in homomorphic encrypted CNN inference is pre-defined.

### 2.2 The FullRNS-CKKS Scheme

We explain the definition of words, HE parameters, and operations of the FullRNS-CKKS scheme. Here, we define $pt$ as a plaintext, $ct$ as a ciphertext, and $t$ as either a plaintext or ciphertext.

- Message: A message is represented as a vector of complex numbers. The FullRNS-CKKS scheme supports packing [9] which encodes or encrypts a vector into a plaintext or ciphertext. Each element in the vector is called slot, and the number of slots is called slot counts.
- Plaintext: Data encoded from a message.
- Ciphertext: Data encrypted from a plaintext.
- Scale $scale(t)$: The bit length of the fraction part. The initial scale of a plaintext and ciphertext is called the scale factor $\Delta$.
- Level $level(t)$: The remaining number of applicable homomorphic multiplication. If the level of a ciphertext is zero, no other homomorphic multiplication can be applied to the ciphertext.
- Degree of polynomial rings $N$: The fixed parameter in a homomorphic encrypted operation. A plaintext or ciphertext $t$ consists of one or more polynomial rings with modulus $Q_{level(t)}$. In the RNS representation, a polynomial ring is divided into several polynomial rings with $\{q_i\}$, where $i \in [0, level(t)]$ and $q_j \approx \Delta$ for $j \in [1, level(t)]$ are satisfied in Microsoft SEAL and OpenFHE.
- Size $size(t)$: The number of polynomial rings in a plaintext or ciphertext without the RNS representation. $size(pt) = 1$ and $size(ct) \geqq 2$ are satisfied.
- Data structure of a plaintext and ciphertext: A 3D array of $size(t) \times level(t) \times N$. Here, we define $\mathbf{t}[i][j][k]$ as an element in $t$, where $0 \leqq i < size(t)$, $0 \leqq j \leqq level(t)$, and $0 \leqq k < N$ are satisfied.
- Number theoretic transform (NTT) form: In some homomorphic operations such as homomorphic multiplication, a negacyclic convolution of two polynomial rings is performed. Because the computational cost of the convolution is high, NTT is applied to a plaintext or ciphertext in normal form to convert it to NTT form and reduce the computational cost of the multiplication from $O(N^2)$ to $O(N \log N)$. Note that a plaintext or ciphertext in NTT form can be converted to that in normal form by applying inverse NTT.
- Homomorphic addition HomAdd$(ct, t)$: We define $ct_{add}$, which is a 3D array of $\max(size(ct), size(t)) \times \min(level(ct), level(t)) \times N$, as a result ciphertext of HomAdd$(ct, t)$ and assume $size(ct) \geqq size(t)$. HomAdd$(ct, t)$ performs $\mathbf{ct_{add}}[i][j][k] \leftarrow \mathbf{ct}[i][j][k] + \mathbf{t}[i][j][k]$, and HomAdd$(ct, t)$ also performs $\mathbf{ct_{add}}[i'][j][k] \leftarrow \mathbf{ct}[i'][j][k]$ if and only if $size(ct) > size(t)$ is satisfied, where $i \in [0, size(t))$, $i' \in [size(t), size(ct))$, $j \in [0, \min(level(ct), level(t))]$, and $k \in [0, N)$ are satisfied. Note that $scale(ct)$ and $scale(t)$ should be the same. In addition, in Microsoft SEAL, the inputs' levels must be the same.

- Homomorphic multiplication HomMul$(ct, t)$: We define $ct_{add}$, which is a 3D array of $(size(ct) + size(t) - 1) \times \min(level(ct), level(t)) \times N$, as a result ciphertext of HomMul$(ct, t)$. Here, we assume that the inputs' are in NTT form and the inputs' sizes are at most two to reduce the computational cost. Then, when $t$ is a ciphertext, HomMul$(ct, t)$ performs $\mathbf{ct_{mul}}[0][j][k] \leftarrow \mathbf{ct}[0][j][k] \times \mathbf{t}[0][j][k]$, $\mathbf{ct_{mul}}[1][j][k] \leftarrow \mathbf{ct}[0][j][k] \times \mathbf{t}[1][j][k] + \mathbf{ct}[1][j][k] \times \mathbf{t}[0][j][k]$, and $\mathbf{ct_{mul}}[2][j][k] \leftarrow \mathbf{ct}[1][j][k] \times \mathbf{t}[1][j][k]$, where $j \in [0, \min(level(ct), level(t))]$ and $k \in [0, N)$ are satisfied. In addition, when $t$ is a plaintext, HomMul$(ct, t)$ performs $\mathbf{ct_{mul}}[i][j][k] \leftarrow \mathbf{ct}[i][j][k] \times \mathbf{t}[0][j][k]$, where $i \in [0, size(ct))$, $j \in [0, \min(level(ct), level(t))]$, and $k \in [0, N)$. $scale(ct_{mul})$ is $scale(ct) \times scale(t)$, and $level(ct_{mul})$ is $\min(level(ct), level(t))$. Note that in Microsoft SEAL, the inputs' levels must be the same.
- Relinearization Relin$(ct)$: The larger the ciphertext's size is, the higher the required computational and spatial costs are. The solution is relinearization which reduces the input's size. $size(\text{Relin}(ct))$ is $size(ct) - 1$ if $size(ct) > 2$ is satisfied.
- Rescaling Rescl$(ct)$: Rescaling reduces the input's scale to keep the bit length for decimal part sufficient. The level of the rescaling output is $level(ct) - 1$.
- Rotation Rotate$(ct, s)$: Rotation permutes the slots of the input circularly with step $s$.

### 2.3 Microsoft SEAL

Microsoft SEAL [2], which is released by Microsoft Research, is an open-source LHE library built in C++ or C#. Microsoft also released EVA [10] which enables building an application with Python. Microsoft SEAL is one of the easiest HE libraries to understand and to use, i.e., user-friendly, because the API is simple. In addition, Microsoft SEAL implements a memory allocator to reduce the overhead of memory allocation and deallocation.

Meanwhile, Microsoft SEAL requires users to understand HE's a basic concept to improve its performance because users must add relinearization and rescaling operations manually. Although EVA adds relinearization and rescaling operations to suitable places, EVA incurs overhead because EVA encodes messages at runtime, even though the encoding can be done in advance. Note that Microsoft SEAL supports not only the FullRNS-CKKS scheme but also the BFV scheme and BGV schemes for modular operations on encrypted integers.

### 2.4 OpenFHE

OpenFHE [11], released in July 2022, is a new open-source FHE library. OpenFHE combines the design ideas of previous FHE projects, such as PALISADE, HElib, and HEAAN, and adds new concepts and ideas. OpenFHE plans to support bootstrapping for all the implemented FHE schemes besides scheme-switching. OpenFHE uses a standard hardware abstraction layer (HAL) to support multiple hardware-accelerated backends. OpenFHE includes both a user-friendly mode and a compiler-friendly mode. With user-friendly mode, all the maintenance operations, such as modulus switching, relinearization, and bootstrapping are automatically invoked. With the compiler-friendly mode, an external compiler makes the above decisions.

OpenFHE combines previously proposed various HE-related schemes, which have been collaboratively constructed by over 26 HE-related research organizations. OpenFHE supports multiple FHE schemes provided by PALISADE, advanced capabilities of the BGV scheme provided by HElib, and the FullRNS-CKKS scheme provided by HEAAN.

### 2.5 Comparison between Microsoft SEAL and OpenFHE in the FullRNS-CKKS Scheme

#### 2.5.1 Memory Allocation

Because the memory allocation and deallocation are executed as kernel functions, their overhead is high. One of the solutions to reduce the overhead is pseudo-allocation and pseudo-deallocation. Microsoft SEAL implements the memory allocator with such pseudo-allocation and pseudo-deallocation; however, OpenFHE does not implement such a memory allocator. Therefore, Microsoft SEAL has less overhead in memory management than OpenFHE.

#### 2.5.2 Adjustment of Scale

When rescaling in the RNS representation, the ciphertext's scale is divided by the last moduli of the ciphertext. In addition, the scale of the ciphertext can be different from the scale factor in the FullRNS-CKKS scheme because all the moduli are coprime and different from the scale factor, even though some moduli are almost the same as the scale factor. Therefore, the scales of inputs of homomorphic addition can be different; however, the scales should be the same, i.e., the scales should be adjusted. Microsoft SEAL does not adjust the scales, i.e., we have to adjust the scales manually; however, OpenFHE adjusts the scales automatically with `FLEXIBLEAUTO`, which is one of the rescaling strategies in OpenFHE. Note that we can adjust the scales by ourselves with `FIXEDMANUAL` in OpenFHE.

## 3 Related Work

We explain the research related to the comparison of HE libraries in Section 3.1. Then, Section 3.2 explains homomorphic encrypted CNN with the FullRNS-CKKS scheme.

### 3.1 Comparison of Homomorphic Encryption Libraries

In 2022, Doan et al. [12] surveyed the implementation of HE schemes. Doan et al. compared the performance of several notable HE schemes covering PHE, SHE, and FHE. Doan et al. compared Microsoft SEAL and PALISADE with homomorphic addition and homomorphic multiplication, followed by demonstrating that Microsoft SEAL was faster than PALISADE. However, the results shown in [12] have two problems: 1) Doan et al. used different parameters between the HE libraries in the comparison, and 2) Doan et al. did not demonstrate the performance of the homomorphic encrypted CNN inference. The latency of an actual application would be different from the latency calculated by the measurement of primitive homomorphic operations; therefore, it is significant to compare the latency of an actual homomorphic encrypted CNN inference among HE libraries.

### 3.2 Homomorphic Encrypted Convolutional Neural Network Inference with the CKKS Scheme

In 2019, Boemer et al. [13] proposed nGraph-HE, which is an extension of Intel's DL graph compiler nGraph. nGraph-HE enables models to be deployed with minimal code changes. In the same year, Boemer et al. [14] proposed nGraph-HE2, the first evaluation of a model with an encrypted ImageNet dataset, which improves nGraph-HE. nGraph-HE2 enables privacy-preserving inference with batch-axis packing and several optimizations, including lazy rescaling to shorten the latency, to implement the FullRNS-CKKS scheme with Microsoft SEAL. As a result, nGraph-HE2 achieved a 3–88x speedup in scalar encoding, a 2.6–4.2x speedup in ciphertext-plaintext scalar addition, and a 2.6x speedup in ciphertext-plaintext scalar multiplication. Furthermore, nGraph-HE2 shortens the latency by 8x on the CryptoNets network by lazy rescaling.

In 2019, Dathathri et al. [15] proposed CHET which is an optimizing compiler for homomorphic encrypted neural network inference. To shorten the inference latency, CHET chooses HW (height-width) layout (channel-wise packing) or CHW (channel-height-width) layout for packing an image to one or more ciphertexts. As a result, CHET achieved higher performance than the expert hand-tuned. In 2020, Dathathri et al. [10] proposed EVA which is an optimizing compiler and runtime for homomorphic encrypted applications. EVA estimates the timing of relinearization and rescaling to shorten the latency; therefore, EVA is user-friendly. EVA achieved a 5.3x speedup compared to CHET.

In 2020, Ishiyama et al. [16] proposed a method to improve the accuracy of CNN inference with HE, which adopted batch-axis packing and different activation functions. Ishiyama et al. achieved accuracies of 99.29% and 81.06% for MNIST and CIFAR-10 datasets,respectively. Furthermore, in 2022, Ishiyama et al. [17], [18] proposed a tuning method of homomorphic encrypted CNN inference, i.e., tuning both latency and accuracy, by changing the channel pruning ratio and activation functions. As a result, Ishiyama et al. successfully tuned the latency from 8.1–12.9 s depending on the accuracy of 66.52–80.96% on the CIFAR-10 dataset. Ishiyama et al. implemented these methods with Microsoft SEAL.

## 4 Comparison Method of Homomorphic Encryption Libraries

### 4.1 Overview

The homomorphic encrypted CNN inference has been accelerated with several optimizations; however, the latency also depends on the implementation. Our goal is to choose the best-fit library for beginners of HE. To achieve our goal, we show an insight for selecting the library targeting Microsoft SEAL and OpenFHE, two popular HE libraries. We implemented a convolutional neural network (CNN) inference application with these two libraries to compare the inference latency and accuracy.

Table 1 CNN model for the MNIST dataset based on [17].

| Layer type | Description | Layer Size |
|---|---|---|
| Convolution | Five filters of size $5 \times 5$ and stride (2, 2) w/o padding | $12 \times 12 \times 5$ |
| Batch normalization | Applying batch normalization | $12 \times 12 \times 5$ |
| Activation | Applying activation function | $12 \times 12 \times 5$ |
| Convolution | 50 filters of size $5 \times 5$ and stride (2, 2) w/o padding | $4 \times 4 \times 50$ |
| Batch normalization | Applying batch normalization | $4 \times 4 \times 50$ |
| Activation | Applying activation function | $4 \times 4 \times 50$ |
| Fully connected | Weighted sum of the entire previous layers w/ 10 filters, each output corresponding to one of the 10 classes | $1 \times 1 \times 10$ |

### 4.2 Datasets

In the experimental evaluation, we use the MNIST dataset [19] and the CIFAR-10 dataset [20], where the MNIST dataset and the CIFAR-10 dataset are common datasets to evaluate CNN with HE. The MNIST dataset consists of images of grayscale $28 \times 28$ pixels, each of which is a single handwritten digit of 0–9, with the ground-truth label. The MNIST dataset consists of 60,000 training data and 10,000 test data. The CIFAR-10 dataset consists of $32 \times 32$ pixels RGB images, each of which is one of 10 different classes, with the ground-truth label. The CIFAR-10 dataset consists of 50,000 training data and 10,000 test data.

### 4.3 Description of the CNN Model

The configuration of the CNN models for the MNIST dataset and CIFAR-10 dataset are the same as those used in [17], [18]. Table 1 and Table 2 list the configuration of the CNN models for the MNIST dataset and CIFAR-10 dataset, respectively.

### 4.4 Activation Function

Table 3 lists the details of the activation functions and non-HE inference accuracy reported in [17] [18]. Here, in the names of the polynomial approximated activation functions, *rg* means a range of approximation, and *deg* means the approximation degree. The range affects only the accuracy; however, the approximation degree affects not only the accuracy but also the latency. Therefore, we select *square*, *swish-rg5-deg2*, and *swish-rg7-deg4* as the activation functions in our experimental evaluation because we use the same trained CNN models used in [17], [18]. Note that we do not adopt channel pruning technique proposed in [17], [18].

### 4.5 Experiment Environment and Condition

We use the source code provided by Ishiyama et al. [18](注7) to execute the homomorphic encrypted CNN inference. Although we adopt Microsoft SEAL 4.0.0 instead of Microsoft SEAL 3.6.6

(注7)：https://github.com/yamanalab/latency-aware-cnn-inference

Table 2 CNN network model for CIFAR-10 (quoted from [17], [18])

| Layer type | Description | Layer Size |
|---|---|---|
| Convolution | 32 filters of size $3 \times 3 \times 3$ and stride (1, 1) with padding. | $32 \times 32 \times 32$ |
| Batch normalization | Applying batch normalization | $16 \times 16 \times 32$ |
| Activation | Applying activation function | $32 \times 32 \times 32$ |
| Average pooling | w/ extent 2 and stride 2 | $16 \times 16 \times 64$ |
| Convolution | 64 filters of size $3 \times 3 \times 32$ and stride (1, 1) with padding | $16 \times 16 \times 64$ |
| Batch normalization | Applying batch normalization | $16 \times 16 \times 64$ |
| Activation | Applying activation function | $16 \times 16 \times 64$ |
| Average pooling | w/ extent 2 and stride 2 | $8 \times 8 \times 64$ |
| Convolution | 128 filters of size $3 \times 3 \times 64$ and stride (1, 1) with padding | $8 \times 8 \times 128$ |
| Batch normalization | Applying batch normalization | $8 \times 8 \times 128$ |
| Activation | Applying activation function | $8 \times 8 \times 128$ |
| Average pooling | w/ extent 2 and stride 2 | $4 \times 4 \times 128$ |
| Global average pooing | Applying global average pooling | $1 \times 1 \times 256$ |
| Fully connected | Weighted sum of the entire previous layer with 10 filters | $1 \times 1 \times 10$ |

which Ishiyama et al. used, we do not change any codes for the version up. However, we modified the source code for OpenFHE, i.e use the operation methods in OpenFHE instead of SEAL. We set `CMAKE_BUILD_TYPE` to `Release` mode in CMake to build Microsoft SEAL, OpenFHE, and our source code.

Based on the methods of Ishiyama et al. [17], [18], we adopt the optimization to merge coefficients of a pair of layers to reduce the level consumption. We merge a pair of the convolutional layer and the batch normalization layer and a pair of the activation function layer and the average pooling layer. Furthermore, if the absolute value of a coefficient $w$ of a layer is less than the threshold $\varepsilon$, the coefficient is rounded; i.e., we set $w$ to $\varepsilon$ when $0.0 \leqq w < \varepsilon$ or set $w$ to $-\varepsilon$ when $-\varepsilon < w < 0.0$, where the threshold $\varepsilon$ is set to $10^{-6}$ for *swish-rg7-deg4* and is set to $10^{-7}$ for *square* and *swish-rg5-deg2*.

Table 4 lists the specification of the server used in the evaluation. In addition, we configure the HE parameters as shown in Table 5. In OpenFHE, we adopt `FLEXIBLEAUTO` for the rescaling strategy.

### 4.6 Experiment Setup

We adopt parallelization among homomorphic operations using OpenMP with static scheduling to shorten the latency, where the static scheduling decides the chunk size before executing the for-loops so that the overhead by chunking is smaller than the other scheduling mode. The number of threads we used is 1, 18, 36, or 72. To evaluate scalability, we use only NUMA-node0 for 1 or 18 threads, use only NUMA-node0 and NUMA-node1 for 36 threads and use all for 72 threads by specifying the NUMA nodes using the `numactl` command. Although OpenFHE supports parallelization in a homomorphic operation, i.e., fine-grained parallelization, we adopt parallelization only among homomorphic operations, i.e., coarse-grained parallelization, because coarse-grained parallelization incurs lower overhead than fine-grained parallelization.

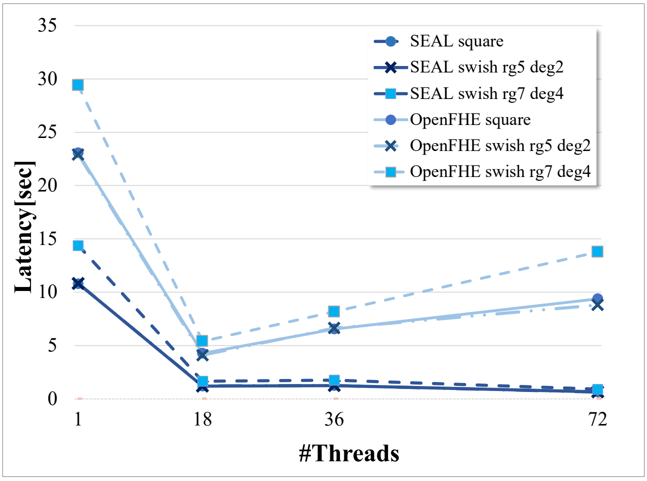

Figure 1 Latency v.s. the number of threads with the MNIST dataset.

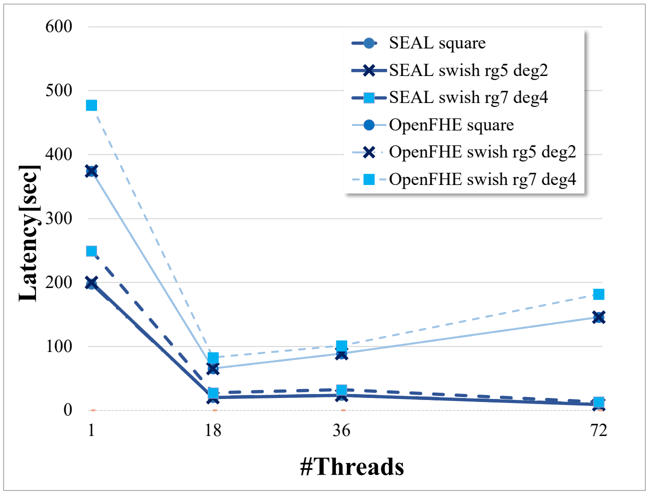

Figure 2 Latency v.s. the number of threads with the CIFAR-10 dataset.

To obtain the average latency, we execute 72 times inference, followed by averaging them for each parameter set and dataset. In addition, to obtain the average accuracy, we execute 1,000 times inference using 72 threads with SEAL and 18 threads with OpenFHE, followed by averaging.

## 5 Result and Discussion

Table 6 shows the latency and accuracy of the MNIST and CIFAR-10 datasets. Besides, Fig. 1 and 2 illustrate the latency versus the number of threads.

Table 7 compares the latency of primitive operations in Microsoft SEAL and OpenFHE under the same conditions. Table 8 and 9 compare the latencies of multiplication and activation functions with FIXEDMANUAL and FLEXI- BLEAUTO mode with OpenFHE, respectively.

### 5.1 Latency Comparison

Table 6 shows that OpenFHE was almost twice slower as Microsoft SEAL with one thread, three times slower with 18 threads, four times slower with 36 threads, and 8 times slower with 72

Table 3 Details of activation functions and inference accuracy over plaintext with MNIST dataset and CIFAR-10 dataset reported in Table 5.3, Table 5.5, and Table 5.6 in [17]

| Activation function name | Range for x-axis | Appoximation degree | Function | Accuracy[%] | |
|---|---|---|---|---|---|
| | | | | MNIST | CIFAR-10 |
| *square* | N/A | N/A | $x^2$ | 99.35 | 79.17 |
| *swish-rg5-deg2* | $[-5, 5]$ | 2 | $0.1 \quad x^2 + 0.5x + 0.24$ | 99.48 | 79.53 |
| *swish-rg7-deg4* | $[-7, 7]$ | 4 | $-0.001328x^4 + 0.128x^2 + 0.5x + 0.1773$ | 99.52 | 80.91 |

Table 4 The server specification and software versions for the experiment.

| | | |
|---|---|---|
| CPU | Model number | Intel Xeon E7-8880 v3 |
| | Frequency (base) | 2.30 GHz |
| | Frequency (turbo) | 3.10 GHz |
| | #cores/CPU | 18 |
| | Hyper-threading | Disabled |
| | L1i cache capacity/core | 32 KiB |
| | L1d cache capacity/core | 32 KiB |
| | L2 cache capacity/core | 256 KiB |
| | L3 cache capacity/CPU | 45 MiB |
| | #CPUs | 4 |
| Memory | Capacity | 3 TB |
| #NUMA nodes | | 4 |
| OS | | CentOS 7.6.1810 |
| Library | Microsoft SEAL | 4.0.0 |
| | OpenFHE | 1.0.0 |
| Compiler & API | g++ | 7.4.0 |
| | OpenMP | 4.5 |
| | Eigen | 3.4.0 |

Table 5 HE Parameters

| Dataset | Function | Level | Degree of polynomial rings $N$ | Slot counts | Scale factor $\Delta$ |
|---|---|---|---|---|---|
| MNIST | *square* | 5 | 16,384 | 8,192 | $2^{30}$ |
| | *swish-rg5-deg2* | 5 | | | |
| | *swish-rg7-deg4* | 7 | | | |
| CIFAR-10 | *square* | 8 | | | |
| | *swish-rg5-deg2* | 11 | | | |
| | *swish-rg7-deg4* | 11 | | | |

threads. We estimate the following two reasons.

The first reason is the different latencies in primitive operations between Microsoft SEAL and OpenFHE. Table 7 shows the latency of each primitive operation in the FullRNS-CKKS scheme with Microsoft SEAL and OpenFHE, where we tested 5 times executions using sample codes to calculate the average latency under the same conditions and parameters. As shown in Table 7, the operation in SEAL is faster than in OpenFHE. Note that, unlike OpendFHE, Microsoft SEAL does not support direct computation with integers, to compute, you need to change an integer into plaintext.

The second reason is that OpenFHE needs more kernel function calls to allocate and deallocate memory than SEAL, which may result in a bottleneck of OpenFHE. To confirm the memory consumption in the inference process for MNIST and CIFAR-10 datasets,

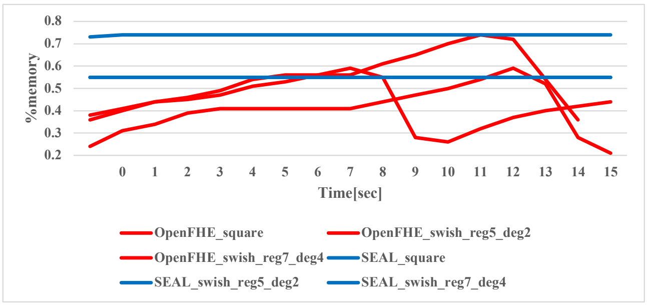

Figure 3 Memory allocation transition w/ SEAL and OpenFHE (MNIST dataset)

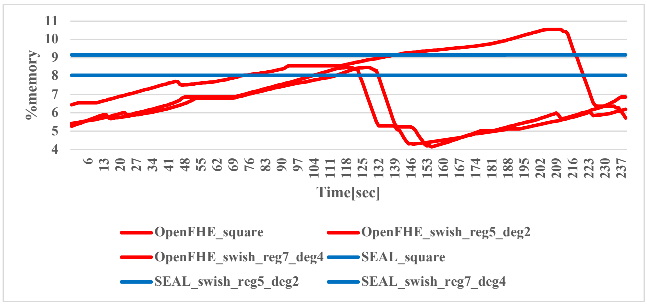

Figure 4 Memory allocation transition w/ SEAL and OpenFHE (CIFAR-10 dataset)

we measured the transition of the physical memory consumption. We recorded the memory consumption during the first 15 seconds and 4 minutes after initiating the second inference of MNIST and CIFAR-10 datasets, respectively. Note that we measured the second inference’s memory consumption to eliminate uncertain behaviors usually occurring at the first inference because of the cache mechanism. Figs. 3 and 4 show the transitions of memory consumption with different activation functions in 72 threads. As shown in Figs. 3 and 4 , Microsoft SEAL keeps memory consumption at the same level, while OpenFHE changes the memory consumption dynamically. From these results, it is clear that since Microsoft SEAL has built-in memory management, Microsoft SEAL does not release the allocated memory to reuse, which results in less overhead compared to OpenFHE.

### 5.1.1 Latency v.s. the Number of Threads

As shown in Fig. 1 and 2, using 72 threads achieved the lowest latency in Microsoft SEAL; however, using 18 threads achieved the lowest latency in the MNIST dataset using OpenFHE, 18 threads and 36 threads in OpenFHE achieved the lowest latency with the square function and the swish function, respectively in the CIFAR-

Table 6 The result w/ MNIST dataset and CIFAR-10 dataset (FIXEDMANUAL mode in OpenFHE)

| Activation function | #threads | MNIST dataset | | | | CIFAR-10 dataset | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | Latency [sec] | | Accuracy [%] | | Latency [sec] | | Accuracy [%] | |
| | | SEAL | OpenFHE | SEAL | OpenFHE | SEAL | OpenFHE | SEAL | OpenFHE |
| *square* | 1 | 11.15 | 23.07 | 99.2 | 99.2 | 201.1 | 374.76 | 80.3 | 78.8 |
| | 18 | 1.20 | 4.28 | | | 20.1 | 68.01 | | |
| | 36 | 1.24 | 6.55 | | | 23.6 | 85.94 | | |
| | 72 | 0.64 | 9.38 | | | 9.6 | 125.19 | | |
| *swish-rg5-deg2* | 1 | 11.08 | 22.91 | 99.3 | 99.3 | 200.2 | 374.37 | 80.3 | 80.1 |
| | 18 | 1.20 | 4.10 | | | 20.2 | 65.52 | | |
| | 36 | 1.24 | 6.63 | | | 23.6 | 88.77 | | |
| | 72 | 0.65 | 8.81 | | | 9.6 | 145.83 | | |
| *swish-rg7-deg4* | 1 | 15.01 | 29.40 | 99.4 | 99.4 | 249.5 | 477.46 | 81.4 | 81.5 |
| | 18 | 1.64 | 5.40 | | | 28.1 | 82.66 | | |
| | 36 | 1.74 | 8.17 | | | 32.3 | 101.20 | | |
| | 72 | 0.74 | 13.77 | | | 13.3 | 181.65 | | |

Table 7 Latency of primitive operations

| Dataset | Level | Operation | Latency[ms] | |
|---|---|---|---|---|
| | | | SEAL | OpenFHE |
| MNIST | 5 | Addition | 0.465 | 4.059 |
| | | Multiplication | 6.181 | 15.298 |
| | | Rotation | 31.450 | 455.506 |
| | | Rescaling | 6.763 | 28.802 |
| | | Relinearization | 0.030 | 0.023 |
| | 7 | Addition | 0.379 | 1.922 |
| | | Multiplication | 5.344 | 6.644 |
| | | Rotation | 36.095 | 102.314 |
| | | Rescaling | 5.794 | 96.606 |
| | | Relinearization | 0.004 | 0.076 |
| Cifar-10 | 8 | Addition | 0.639 | 4.251 |
| | | Multiplication | 6.969 | 7.329 |
| | | Rotation | 53.200 | 128.671 |
| | | Rescaling | 8.612 | 26.781 |
| | | Relinearization | 0.015 | 0.031 |
| | 11 | Addition | 2.556 | 2.417 |
| | | Multiplication | 16.192 | 7.798 |
| | | Rotation | 127.979 | 160.022 |
| | | Rescaling | 17.557 | 50.683 |
| | | Relinearization | 0.047 | 0.080 |

Table 8 Multiplication latency in `FIXEDMANUAL` mode and `FLEXIBLEAUTO` mode w/ OpenFHE

| Dataset | Level | Latency[sec] | |
|---|---|---|---|
| | | FLEXIBLEAUTO | FLEXIBLEAUTO |
| MNIST | 5 | 0.308 | 0.795 |
| | 7 | 0.404 | 0.670 |
| Cifar-10 | 8 | 0.427 | 0.729 |
| | 11 | 0.566 | 0.575 |

Table 9 Activation function latency in `FIXEDMANUAL` mode and `FLEXIBLEAUTO` mode w/ OpenFHE

| Activation function | #threads | Latency[sec] | |
|---|---|---|---|
| | | `FIXEDMANUAL` | `FLEXIBLEAUTO` |
| Square | 1 | 23.07 | 132.28 |
| | 18 | 4.28 | 10.26 |
| | 36 | 6.55 | 8.86 |
| | 72 | 9.38 | 9.66 |
| swish-rg5-deg2 | 1 | 22.91 | 134.36 |
| | 18 | 4.10 | 10.18 |
| | 36 | 6.63 | 7.80 |
| | 72 | 8.81 | 9.22 |
| swish-rg7-deg4 | 1 | 29.40 | 116.75 |
| | 18 | 5.40 | 9.98 |
| | 36 | 8.17 | 9.04 |
| | 72 | 13.77 | 13.75 |

10 dataset. As we mentioned in our experimental setup (Section 4.6), we used fewer NUMA nodes on the larger number of threads. The reason why using more threads results in longer latency with OpenFHE is that the latency reduced by the increment of threads does not offset the latency increased by the memory allocation and deallocation due to the lack of memory management capability of OpenFHE.

### 5.1.2 Mode comparison in OpenFHE

Table 9 shows the difference in the latency between `FIXEDMANUAL` and `FLEXIBLEAUTO` in OpenFHE, where `FLEXIBLEAUTO` is easier to use than `FIXEDMANUAL` because rescaling is performed automatically by OpenFHE with `FLEXIBLEAUTO`. As shown in Table 9, `FIXEDMANUAL` performs better, i.e., smaller latency, than `FLEXIBLEAUTO` in this case. The reason is that `FLEXIBLEAUTO` is designed so that OpenFHE automatically applies rescaling just before homomorphic multiplication, except for the first homomorphic multiplication. Specifically, we measured the multiplication latency in FIXEDMANUAL and FLEXIBLEAUTO mode with OpenFHE by executing a multiplication 5 times to have Table 8. As shown in Table 8, FLEXIBLEAUTO consumes more time to apply automatic rescaling than FIXEDMANUAL.

## 5.2 Comparison of the Accuracy

As shown in Table 6, in the inferences of 1,000 images, OpenFHE shows the same accuracy as Microsoft SEAL w/ the MNIST dataset,

Table 10 Accuracy w/ Microsoft SEAL, FIXEDMANUAL mode and FLEXIBLEAUTO mode

| Activation function | Accuracy | | |
|---|---|---|---|
| | Microsoft SEAL | OpenFHE FIXEDMANUAL | OpenFHE FLEXIBLEAUTO |
| square | 78% | 77% | 78% |
| swish-rg5-deg2 | 81% | 78% | 81% |
| swish-rg7-deg4 | 83% | 83% | 83% |

and lower accuracy in the CIFAR-10 dataset except for the *swish-rg7-deg4*. Although we used the same computational process in SEAL and OpenFHE, the `FIXEDMANUAL` mode in OpenFHE will incur an additional precision loss of around 3-4 bits. Besides, Table 10 compares the accuracy between Microsoft SEAL and `FIXEDMANUAL` mode and `FLEXIBLEAUTO` mode in OpenFHE in 100 inferences. As shown in Table 10, OpenFHE’s `FLEXIBLEAUTO` mode achieves the same precisions as SEAL. This fact implies the existence of another reason that our setting in `FIXEDMANUAL` mode is not the best solution in OpenFHE, even if the setting is the same as Microsoft SEAL.

## 6 Conclusion

This paper focused on the differences between two homomorphic encryption libraries, Microsoft SEAL and OpenFHE, to reveal the essential concepts and implementations to improve the performance of homomorphic encryption CNN inference. In our experimental results, Microsoft SEAL performs better than OpenFHE in terms of latency because Microsoft SEAL has built-in memory management. OpenFHE shows the same accuracy as Microsoft SEAL in the MNIST dataset and lower accuracy in the Cifar-10 dataset except for the *swish-rg7-deg4*.

OpenFHE is capable of automatically scaling numbers and supporting direct calculations with integers. Using SEAL is the best choice for programs customized by FHE experts or compilers for specific applications. For beginners of FHE or people who want to use FHE without sufficient knowledge, OpenFHE is a better approach even though it incurs 10x longer latency than SEAL.

One of the prospects is to apply memory management in OpenFHE and evaluate its performance of the homomorphic encrypted CNN inference. More importantly, we also need to evaluate the performance of OpenFHE and Microsoft SEAL using different CNN models or other applications.

## References

[1] C Gentry. A fully homomorphic encryption scheme. phd thesis, stanford university, 2009. 2009.

[2] Microsoft SEAL (release 4.0). `https://github.com/Microsoft/SEAL`, March 2022. Microsoft Research, Redmond, WA.

[3] Ahmad Al Badawi, Jack Bates, Flavio Bergamaschi, David Bruce Cousins, Saroja Erabelli, Nicholas Genise, Shai Halevi, Hamish Hunt, Andrey Kim, Yongwoo Lee, et al. Openfhe: Open-source fully homomorphic encryption library. In *Proceedings of the 10th Workshop on Encrypted Computing & Applied Homomorphic Cryptography*, pages pp.53–63, 2022.

[4] Junfeng Fan and Frederik Vercauteren. Somewhat practical fully homomorphic encryption. *IACR Cryptol. ePrint Arch. 2012:144*, 2012.

[5] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. (leveled) fully homomorphic encryption without bootstrapping. *ACM Transactions on Computation Theory (TOCT)*, 6(3):pp.1–36, 2014.

[6] Jung Hee Cheon, Andrey Kim, Miran Kim, and Yongsoo Song. Homomorphic encryption for arithmetic of approximate numbers. In *Proceedings of the Advances in Cryptology – ASIACRYPT 2017,LNCS,vol.10624,pp.409–437*. Springer, 2017.

[7] Jung Hee Cheon, Kyoohyung Han, Andrey Kim, Miran Kim, and Yongsoo Song. Bootstrapping for approximate homomorphic encryption. In *Proceedings of the Advances in Cryptology – EUROCRYPT 2018, LNCS, vol. 10820, pp. 360–384*. Springer, 2018.

[8] Jung Hee Cheon, Kyoohyung Han, Andrey Kim, Miran Kim, and Yongsoo Song. A full rns variant of approximate homomorphic encryption. In *Proceedings of the Selected Areas in Cryptography – SAC 2018, LNCS, vol.11349, pp. 347–368*. Springer, 2019.

[9] Nigel P Smart and Frederik Vercauteren. Fully homomorphic simd operations. *Designs, codes and cryptography*, 71(1):pp.57–81, 2014.

[10] Roshan Dathathri, Blagovesta Kostova, Olli Saarikivi, Wei Dai, Kim Laine, and Madan Musuvathi. Eva: An encrypted vector arithmetic language and compiler for efficient homomorphic computation. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages pp.546–561, 2020.

[11] Openfhe brings new encryption tools to developers. https://www.darkreading.com/dr-tech/openfhe-brings-new-encryption-tools-to-developers, 2022.

[12] Thi Van Thao Doan, Mohamed-Lamine Messai, Gérald Gavin, and Jérôme Darmon. A survey on implementations of homomorphic encryption schemes, 2022. Preprint, doi: 10.21203/rs.3.rs-2018739/v2.

[13] Fabian Boemer, Yixing Lao, Rosario Cammarota, and Casimir Wierzynski. Ngraph-he: a graph compiler for deep learning on homomorphically encrypted data. In *Proceedings of the 16th ACM International Conference on Computing Frontiers*, pages pp.3–13, 2019.

[14] Fabian Boemer, Anamaria Costache, Rosario Cammarota, and Casimir Wierzynski. Ngraph-he2: A high-throughput framework for neural network inference on encrypted data. In *Proceedings of the 7th ACM Workshop on Encrypted Computing & Applied Homomorphic Cryptography*, pages pp.45–56, 2019.

[15] Roshan Dathathri, Olli Saarikivi, Hao Chen, Kim Laine, Kristin Lauter, Saeed Maleki, Madanlal Musuvathi, and Todd Mytkowicz. Chet: an optimizing compiler for fully-homomorphic neural-network inferencing. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages pp.142–156, 2019.

[16] Takumi Ishiyama, Takuya Suzuki, and Hayato Yamana. Highly accurate cnn inference using approximate activation functions over homomorphic encryption. In *Proceedings of the 2020 IEEE International Conference on Big Data (Big Data)*, pages pp.3989–3995. IEEE, 2020.

[17] 石山琢己, 鈴木拓也, and 山名早人. 準同型暗号上での畳み込みニューラルネットワーク推論に対する channel pruning の適用. In *第 14 回データ工学と情報マネジメントに関するフォーラム (DEIM2022)*, number J33-4, pages pp.1–8, 2022.

[18] Takumi Ishiyama, Takuya Suzuki, and Hayato Yamana. Latency-aware inference on convolutional neural network over homomorphic encryption. In *Proceedings of the 24th International Conference on Information Integration and Web Intelligence (iiWAS 2022), LNCS, vol.13635, pp.324–337*.

[19] Yann Lecun, Leon. Bottou, Yoshua. Bengio, and Patrick. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):pp.2278–2324, 1998.

[20] Alex Krizhevsky. Learning multiple layers of features from tiny images, 2009. `https://www.cs.toronto.edu/~kriz/learning-features-2009-TR.pdf`.