FlexHE: a Flexible Kernel Generation Framework for Homomorphic Encryption-Based Private Inference

Jiangrui Yu[1,2], Wenxuan Zeng[1,5], Tianshi Xu[2,1], Renze Chen[6], Yun Liang[2,4], Runsheng Wang[2,3,4], Ru Huang[2,3,4], Meng Li[1,2,4*]

[1]Institute for Artificial Intelligence & [2]School of Integrated Circuits, Peking University, Beijing, China

[3]Institute of Electronic Design Automation, Peking University, Wuxi, China

[4]Beijing Advanced Innovation Center for Integrated Circuits, Beijing, China

[5]School of Software and Microelectronics, Peking University, Beijing, China

[6]School of Computer Science, Peking University, Beijing, China

Abstract

Secure two-party computation (2PC) based on homomorphic encryption (HE) achieves formal data privacy protection and gets increasing adoption for private deep neural network (DNN) inference. As modern HE schemes usually operate on polynomials, existing works rely on manually-designed HE kernels for representative DNN operations. However, this is not only unscalable considering the diverse operator types, shapes, polynomial orders, etc, but also misses important optimization opportunities. In this paper, we introduce FlexHE, a flexible kernel generation framework to enable automatic generation and optimization of HE kernels for 2PC-based private inference. Given a high-level description of DNN operations, FlexHE can systematically define the HE kernel design space considering various optimization dimensions, including loop tiling, reordering, etc. We also analyze the communication and computation impact of different optimization dimensions for design space reduction. To search for the best kernel design, a two-level optimization problem is formulated and iteratively solved with an integer linear programming (ILP) formulation. With extensive experimental results, we not only demonstrate a better coverage of DNN operations including depth-wise Conv3D and dilated Conv3D, but also achieve more than 100×, 7.9×, and 4.2× latency reduction compared to prior-art HElayers, Cheetah, and Falcon, respectively.

Keywords: Homomorphic Encryption, Private Inference, Encoding and Kernel Generation, Integer Linear Programming

1 Introduction

The last decade has witnessed the rapid evolution of deep learning (DL) and its increasing adoption in privacy-sensitive applications, including medical diagnosis [13], face recognition [30], financial system [14], etc. Privacy has emerged as a major concern when applying DL in these real-world applications. Therefore, there is a growing demand for privacy-preserving DL [10, 17, 18, 23, 28].

Secure two-party computation (2PC) based on homomorphic encryption (HE) has recently been proposed to protect data privacy with a formal privacy guarantee and has attracted a lot of attention [12, 19–22]. The 2PC framework protects the privacy of both model weights held by the server and the input data owned by the client. Through the execution of a series of cryptography protocols, the client can eventually learn the final inference results but nothing else can be derived. Meanwhile, the server knows nothing about the client's input [20–22].

HE-based 2PC frameworks enjoy the benefits of better communication efficiency compared to other alternative 2PC frameworks but face significant challenges with HE-based computation [11]. Specifically, deep neural networks (DNNs) process high-dimensional tensors while HE computes over polynomials, whose coefficients are usually regarded as a one-dimensional vector. As shown in Figure 2, the mapping from tensors to polynomials, denoted as encoding (or packing) [12], directly determines the computation correctness and efficiency. Therefore, a central question with HE-based 2PC frameworks is how to design encoding algorithms for high-performance DNN kernel generation.

Most of the existing researches [10, 11, 28] focus on developing hand-optimized encoding algorithms, where the developers need to manually design high-performance protocols by analyzing the computation pattern and the trade-offs of different optimization options. This approach, however, suffers from important drawbacks. On the one hand, manual optimization is unscalable considering the diverse sets of DNN operation types, shapes, HE parameters, and system-level communication and computation capability, etc. On the other hand, this approach is also error-prone and can miss important optimization opportunities, leading to sub-optimal encoding solutions. Even though some recent works have initiated the study of HE compilers, they primarily focus on graph-level optimization [7] and are developed for end-to-end HE-based computation [1, 15], which suffers from limited support for non-linear activation functions and high computation overhead.

To overcome the drawbacks above, in this paper, we propose FlexHE, a flexible encoding and kernel generation framework for HE-based 2PC. Given a high-level description of DNN operations, FlexHE can automatically analyze and generate the kernel design space. We also systematically analyze different optimization dimensions, including tiling, loop reordering, elements reordering, etc, and their impact on communication and computation complexity to reduce the design space and improve optimization convergence. Then, a two-level optimization problem is formulated to enable automatic encoding optimization. Our contributions can be summarized as below:

*Corresponding Author, meng.li@pku.edu.cn

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ICCAD '24, October 27–31, 2024, New York, NY, USA

© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ACM ISBN 979-8-4007-1077-3/24/10
https://doi.org/10.1145/3676536.3676739

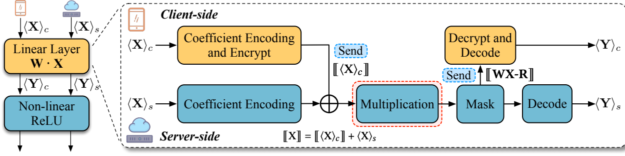

Figure 1. Two-party computation framework with linear and non-linear layers. The blue dashed box represents communication latency while the red one represents computation latency.

- We systematically analyze the optimization space for HE encoding and propose FlexHE to enable flexible kernel generation and encoding optimization for DNN operators.
- We propose static analysis in the front-end of FlexHE to automatically analyze and generate the encoding optimization space.
- We propose a two-level optimization formulation in the back end of FlexHE to optimize for the total latency efficiently.
- Compared to prior-art baselines, including HElayers[1], Cheetah [11], and Falcon [28], we not only realize a much better DNN operator coverage but also achieve more than 100×, 7.9×, and 4.2× latency reduction respectively.

2 Background

2.1 Threat Model

We focus on efficient privacy-preserving DNN inference, which involves two parties, i.e., the server and the client. The client initially holds private data, while the server holds a model. We assume the attackers to be semi-honest and hence, they will follow the predefined protocols but attempt to learn more information than permitted. Following [10, 11, 21, 22, 28], we assume that the model architecture, including the number and types of layers, their dimensions, and bit widths, are openly known to both the server and the client. 2PC ensures that by the end of the inference process, the client is only allowed to learn the model's architecture and inference result while the server gains no information about the client's input.

2.2 Notations

In this paper, we use $[N]$ to denote the set $\{0, 1, 2, ..., N-1\}$, bold upper-case letters such as $\mathbf{W}$ to represent multi-dimensional tensors, lower-case bold letters like $\mathbf{m}$ to represent vectors, and $\mathbf{m}[j]$ to denote the j-th element of the vector $\mathbf{m}$. We use lower-case letters with "hat" such as $\hat{p}$ to represent a polynomial, and $\hat{p}[j]$ to denote the jth coefficient of the polynomial $\hat{p}$. We denote by $\hat{a} \cdot \hat{b}$ the multiplication of two polynomials. For a power-of-two number N and $q > 0$, we write $\mathbb{A}_{N,q}$ to denote the set of integer polynomials $\mathbb{A}_{N,q} = \mathbb{Z}_q[X]/(X^N + 1)$. We denote $[\![\mathbf{M}]\!]$ as the HE ciphertexts on tensor $\mathbf{M}$.

2.3 HE-based 2PC Framework

We review the strengths and basic flow of the 2PC framework. 2PC frameworks that combine arithmetic sharing (ArSS) and HE has been widely studied in [10–12] due to their high flexibility and high accuracy when supporting different non-linear activation functions [9, 16]. Compared to the end-to-end FHE framework, the 2PC framework has higher accuracy due to the 2PC's ability to realize accurate non-linear functions, while FHE requires polynomial approximation, which suffers from prohibitively accuracy degradation [8]. Although some HE schemes, such as TFHE [6], can implement arbitrary functions, they still suffer from low computational efficiency [25]. Moreover, when combined with other FHE schemes like CKKS [5], switching between schemes incurs a large accuracy drop [3]. What's more, the 2PC framework can repack the ciphertext during computation, thereby eliminating the need for expensive bootstrapping operations. Thus, we focus on the 2PC framework in this paper.

As described in Section 1, the computation of DNNs requires encoding to convert high-dimensional tensors into one-dimensional coefficient vectors. To achieve this, there are two widely used encoding schemes, including SIMD [12] and coefficient encoding [10, 23, 28]. SIMD encoding allows element-wise addition, multiplications, and rotations on encrypted vectors. However, this encoding scheme incurs large computation overhead to compute multiply-and-accumulate operations, which requires many expensive rotation operations. On the other hand, coefficient encoding puts elements in plaintext polynomial coefficients directly, thus allowing element-wise addition and convolution between two vectors. Note convolution can be used to implement multiply-and-accumulate efficiently. Moreover, SIMD encoding restricts plaintext modulus to $2kN + 1$ where k is an integer and N is the polynomial degree. Coefficient encoding allows for a larger set of plaintext modulus like power-of-2, which benefits the non-linear functions implemented by 2PC [11]. Therefore, the coefficient encoding scheme is usually much more computation-efficient compared to SIMD encoding for DNN operators [10, 11] and is the focus of our work.

As illustrated in Figure 1, each intermediate activation tensor is additively shared between the server and the client as $\langle \mathbf{X} \rangle_c$ and $\langle \mathbf{X} \rangle_s$, respectively, with $\mathbf{X} \equiv \langle \mathbf{X} \rangle_c + \langle \mathbf{X} \rangle_s$. Both the server and the client encode their inputs into plaintext polynomials following a pre-determined protocol. The client then encrypts its plaintext to generate $[\![\langle \mathbf{X} \rangle_c]\!]$ and sends it to the server. After receiving the client's share, the server reconstructs $\mathbf{X}$ as $[\![\mathbf{X}]\!] = [\![\langle \mathbf{X} \rangle_c]\!] + \langle \mathbf{X} \rangle_s$. The server then homomorphically computes $\mathbf{W}\,[\![\mathbf{X}]\!] - \mathbf{R}$, obtaining $[\![\mathbf{Y}]\!]_c = [\![\mathbf{WX} - \mathbf{R}]\!]$, where $\mathbf{R}$ is a randomly generated mask. Finally, $[\![\mathbf{Y}]\!]_c$ is sent back to the client as the client's share, and the server sets $\langle \mathbf{Y} \rangle_s = \mathbf{R}$ as the server's share.

From the discussion above, it is clear that the total latency of the linear layer consists of two parts: **computation** and **communication**. The computation latency is the time to perform homomorphic computations, while communication latency is the time to send the secret shares. Different encoding methods affect these two parts differently, resulting in different efficiency.

2.4 Related Works

As shown in Table 1, previous works on HE kernel generation mainly fall into the following two categories.

Hand-optimized encoding algorithms Hand-optimized coefficient encoding methods are currently the primary means to achieve correct and efficient encoding. For instance, Cheetah [11] proposes an efficient encoding algorithm for 2D convolutions (Conv2D), while Iron [10] and Falcon [28] further optimize general matrix multiplications (GEMM) and depth-wise Conv2D (DEP). Although these methods achieve good performance, they usually rely on fixed encoding rules and optimize in a relatively small space, like the number of channels packed within one polynomial. Through

Table 1. Qualitative comparison with previous works. "Comm." means "Communication", "Mem." means "Memory", and "Lat." means "Latency".

Methods	Hand-optimized [10, 11, 28]	SIMD-compiler [1, 7]	FlexHE (ours)
Encoding method	Coefficient	SIMD	Coefficient
Framework	HE-based 2PC	End-to-end FHE	HE-based 2PC
Supported Kernels	GEMM [10] Conv2D [11] DEP [28]	GEMM Conv2D MatMul	Any conform to Section 4.2
Optization Target	Minimize Comm.	Minimize Mem./ Lat.	Minimize Lat.
Kernel Generation	✗	✓	✓
Optization for Environment	✗	✗	✓

our evaluation, under different communication and computation environments, these works may be suboptimal. Moreover, hand-optimized encoding algorithms are error-prone and labor-intensive, making it hard to introduce new operators.

SIMD-encoding based compiler This category includes SIMD-encoding based compilers such as CHET [7], HElayers [1], etc. These compilers focus on end-to-end FHE frameworks with SIMD encoding and mainly optimize for noise growth, the number of bootstrapping, etc., which is not the bottleneck in a 2PC framework. Moreover, they only support limited operators like GEMM and Conv2D. Our work, however, focuses on the HE-based 2PC framework with coefficient encoding, which faces different challenges, and we hope to support a much wider range of operators. Therefore, a framework capable of automatically generating and optimizing encoding methods is needed.

3 Motivations

Motivation 1 Since HE requires encoding tensors into polynomial coefficients and only supports polynomial-level arithmetic, manually generating encoding algorithms can be error-prone and labor-intensive due to the big gap between tensor operations and polynomial multiplications. Thus, ***correctness*** is the first aspect we need to focus on.

We use a simple example of GEMM in Figure 2 to illustrate. As can be observed, we pack $\mathbf{X}$ and $\mathbf{W}$ into $\hat{x}$ and $\hat{w}$, and this encoding allows the results to be extracted in certain coefficients of the resulting polynomial. However, as shown at the bottom of Figure 2, if we slightly modify the encoding of elements $\mathbf{X}[1,0]$ and $\mathbf{X}[1,1]$, the final results get corrupted.

Motivation 2 Besides the correctness aspect, ***optimization*** is even harder considering various DNN operators, numerous encoding methods, and varied computation and communication conditions. We give two examples to show this aspect as follows.

First, different encoding methods lead to different performance. In Figure 3(a), we use three different encoding methods to encode Conv2D and choose three different input shapes under the same environment 4CPU+WAN for illustration. Different encodings have varied numbers of input and output channels in one polynomial. While the difference is small, their performance impact is noticeable.

Second, different computation and communication environments further add complexity. In Figure 3(b), we compare three different environments combined with the above three encodings and apply them to C2. As can be observed, for these three environments, the optimal computation and communication latency are achieved by different encoding schemes.

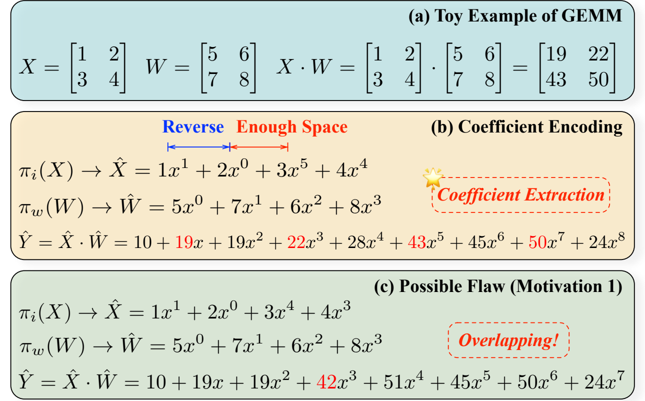

Figure 2. Toy example of GEMM with coefficient encoding and the possible flaw. (a) Multiplication of matrix $\mathbf{X}$ and $\mathbf{W}$. (b) Packing of GEMM. $\mathbf{W}$ is packed in column-major order and $\mathbf{X}$ is packed in row-major order reversely; each row is left enough space to avoid overlapping. (c) Possible flaws for coefficient packing. The space between two rows of $\mathbf{X}$ is not enough, resulting in overlapping with dummy elements. The "42" in red color should be "22", but gets corrupted by $4x^3 * 5x^0$ after shortening the spacing.

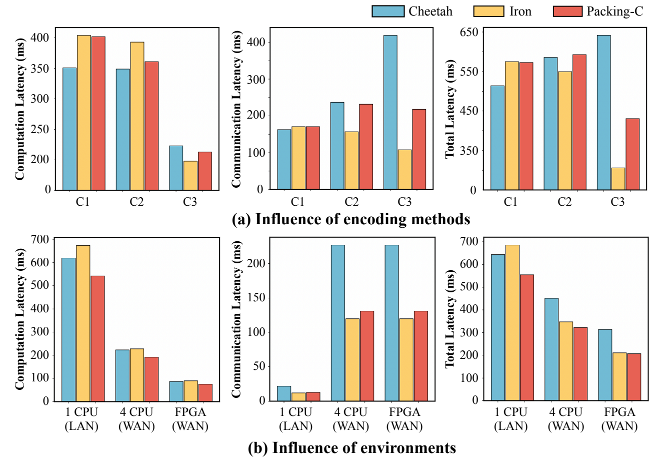

Figure 3. Efficiency differences among various (a) encoding methods and (b) environments. Cheetah [11] packs as many input channels as possible in one polynomial. Iron [10] guarantees the ratio of output and input channels in one polynomial equals the plaintext conv. Packing C always packs two input channels in one polynomial. C1, C2, and C3 represent input shapes of (28, 128, 128), (14, 256, 256), (7, 256, 512) in the order of (H/W, C, K).

From the above examples, it is clear that generating new encoding methods correctly and optimizing them presents huge challenges for developers. Inspired by these motivations, in this paper, we propose to use an automatic kernel generation framework to generate correct and optimized encoding methods for efficient 2PC-based private inference. We summarize the challenges of designing such a kernel generation framework as follows:

Challenge 1 The search space for encoding is extremely large, yet the correct encoding takes up only a small fraction of this space.

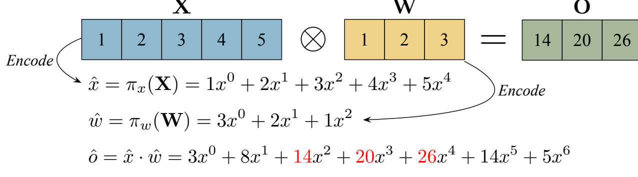

Figure 4. Example of Conv1D encoding.

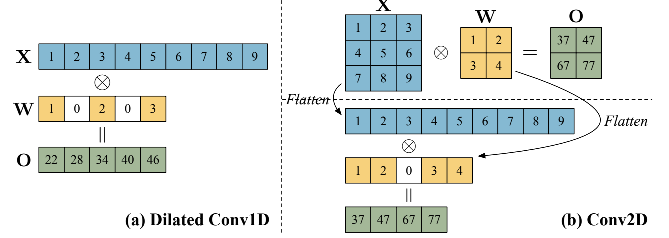

Figure 5. Example of the computation process of (a) Conv2D and (b) dilated Conv1D.

Therefore, we must prune the search space to those where most encoding methods are correct.

Challenge 2 The variety of encoding primitives, parameters, and environmental settings makes the optimization space still very large and hard to explore. Thus, we need to find an efficient way to improve the exploration efficiency.

Corresponding to these two challenges, we first give a detailed analysis of coefficient encoding and then formulate the optimization space where most encodings are correct in Section 4. In Section 5, we introduce our kernel generation framework based on previous analysis and formulate the problem as a two-level optimization process to solve it more efficiently.

4 Analytical Modeling

Due to the extremely large encoding space, and the huge portion of incorrect encodings, it is impossible to search in this space. Therefore, in this section, we first give a detailed analysis of coefficient encoding (Section 4.1) to show its ability to support multi-dimensional convolutions, and then show how to prune the encoding space into a subspace where most encodings are correct (Section 4.2).

We use the Einstein summation convention to represent tensor computation for ease of illustration. The subscripts that appear on the right side but don't appear on the left side indicate reduction. For example, one-dimensional convolution (Conv1D) is denoted as $\mathbf{o}_i = \mathbf{x}_{i+k}\mathbf{w}_k$ where $\mathbf{o}$, $\mathbf{x}$ and $\mathbf{w}$ are vectors.

4.1 Analysis for Coefficient Encoding

To identify the correct encoding methods, we need to first derive the functionalities provided by coefficient encoding. We start with a simple example, i.e., Conv1D. We observe that polynomial multiplication performs negacyclic convolution between their coefficients. Thus to perform Conv1D between two input vectors $\mathbf{x}$ and $\mathbf{w}$ with coefficient encoding, we can simply encode one vector in the polynomial as its original sequence and encode another in reverse order. We now give the definitions of the encoding $\pi_x(\mathbf{x}) : \mathbb{Z}_p^{N1} \rightarrow \mathbb{A}_{N,p}$ and $\pi_w(\mathbf{w}) : \mathbb{Z}_p^{N_2} \rightarrow \mathbb{A}_{N,p}$ and assume $N_1 N_2 \leq N$ for simplicity.

$$\begin{aligned} \hat{x} &= \pi_x(\mathbf{x}) \text{ where } \hat{x}[i] = \mathbf{x}[i], \\ \hat{w} &= \pi_w(\mathbf{w}) \text{ where } \hat{w}[k] = \mathbf{w}[N_2 - k], \end{aligned} \tag{1}$$

where $i \in [N_1]$, $k \in [N_2]$ and all other coefficients of $\hat{x}$ and $\hat{w}$ are set to 0. The polynomial multiplication $\hat{o} = \hat{x} * \hat{w}$ directly gives the result of one-dimensional convolution in certain coefficients. We present a simple example in Figure 4.

Proposition 4.1. *Given two vectors* $\mathbf{x}$ *and* $\mathbf{w}$*, we encode them into polynomials* $\hat{x} = \pi_x(\mathbf{x})$ *and* $\hat{w} = \pi_w(\mathbf{w})$ *as Equation (1) shows, then the convolution between two vectors* $\mathbf{o} = Conv1D(\mathbf{x}, \mathbf{w})$ *can be computed via the polynomial product* $\hat{o} = \hat{x} \cdot \hat{w}$ *over ring* $\mathbb{A}_{N,p}$.

Then, we observe that for a strided convolution, such as $\mathbf{o}_i = \mathbf{x}_{2i+k} * \mathbf{w}_k$, it could be handled as a regular convolution and then only pick the relevant elements. Similarly, for dilated convolution like $\mathbf{o}_i = \mathbf{x}_{i+2k} * \mathbf{w}_k$, we insert zeros between the weight elements as shown in Figure 5 (a).

We observe that multi-dimensional convolution (ConvND) can be converted into Conv1D. Given two tensors $\mathbf{X}$ and $\mathbf{W}$, ConvND can be computed as $\mathbf{O}_{i_1,i_2,\ldots,i_n} = \mathbf{X}_{i_1+k_1,i_2+k_2,\ldots,i_n+k_n}\mathbf{W}_{k_1,k_2,\ldots,k_n}$. If we flatten $\mathbf{X}$ as a vector directly, and flatten $\mathbf{W}$ as a vector with a proper number of zeros inserted between its elements, the ConvND can be computed by the Conv1D of the flattened vectors. We now define the flattening function: $\mathbf{x} = F_x(\mathbf{X}) : \mathbb{Z}_p^{N_1 \times N_2 \times \ldots \times N_n} \rightarrow \mathbb{Z}_p^{N_t}$ where $N_t = \Pi_i N_i$ and $\mathbf{w} = F_w(\mathbf{W}) : \mathbb{Z}_p^{M_1 \times M_2 \times \ldots \times M_n} \rightarrow \mathbb{Z}_p^{M_t}$ where $M_t = \Pi_i M_i$. Here we assume $M_i \leq N_i$ for simplicity.

$$\begin{aligned} \mathbf{x}[i_1 * N_2 * \ldots * N_n + \ldots + i_n] &= \mathbf{X}[i_1, i_2, \ldots, i_n], \\ \mathbf{w}[j_1 * N_2 * \ldots * N_n + \ldots + j_n] &= \mathbf{W}[j_1, j_2, \ldots, j_n], \end{aligned} \tag{2}$$

where all other coefficients are set to zero. The Conv1D of $\mathbf{x}$ and $\mathbf{w}$ directly gives the multi-dimensional convolution in some of its elements. We present an example in Figure 5(b) and omit the formal proof due to space limit.

Proposition 4.2. *Given two multi-dimensional tensors* $\mathbf{X}$ *and* $\mathbf{W}$*, the multi-dimensional convolution between the two tensors can be evaluated by the one-dimensional convolution between* $\mathbf{x} = F_x(\mathbf{X})$ *and* $\mathbf{w} = F_w(\mathbf{W})$ *as Equation (2) shows.*

Remark From the above analysis, we observe that coefficient encoding provides functionalities of multi-dimensional convolution with arbitrary strides and dilations between tensors. This makes it suitable for deep learning operations as ConvND performs multiply-and-accumulate operations in nature and thus eliminates the need for the expensive rotation used in SIMD encoding for accumulation.

4.2 Analysis for DL Operators

Based on functionalities provided by coefficient encodings, in this section, *we want to show most DL operators can be decomposed to several ConvNDs between the slice of* $\mathbf{X}$ *and the slice of* $\mathbf{W}$. From proposition 4.2, the encoding of each slice can then be analytically determined, and thus each slice can be treated as a whole block. Therefore, the whole encoding space is pruned to a subspace where the encoding of each slice is guaranteed to be correct and we only need to optimize the arrangements of these blocks, leading to much better optimization efficiency.

We represent tensor computations as $\mathbf{O} = op(\mathbf{X}, \mathbf{W})$, where $\mathbf{X}$ is the activation tensor, $\mathbf{W}$ is the weight tensor, and $\mathbf{O}$ is the output tensor. We categorize the indices in tensor operations into

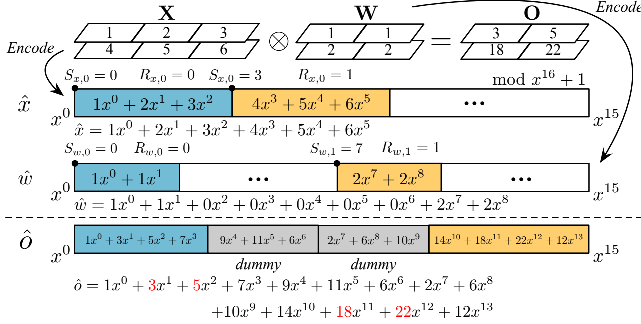

Figure 6. Example of depth-wise Conv1D encoding. Each channel of **X** is considered as a whole block and encoded consecutively into the polynomial. Each channel of **W** is considered as a whole block and is encoded with enough spacing. Elements inside the block are encoded reversely based on proposition 4.1.

four groups — m, i, j, and k — based on the tensor they appear in. Specifically, the index m appears in all three tensors, i appears in **X, O**, j appears in **O, W**, and k appears in **X, W**. We still adopt the Einstein summation convention as mentioned in Section 4.1. For instance, depth-wise convolution can be written as $\mathbf{O}_{m_1,i_1,i_2} = \mathbf{X}_{m_1,i_1+k_1,i_2+k_2}\mathbf{W}_{m_1,k_1,k_2}$.

We observe that for most DL operations, there are two situations: 1) the indices appear individually in each dimension, including GEMM ($\mathbf{O}_{i_1,j_1} = \mathbf{X}_{i_1,k_1}\mathbf{W}_{k_1,j_1}$), where each dimension has only one indice, and matrix-vector multiplication (GEMV). 2) some dimensions have linear combinations of i or j with k, including Conv1D ($\mathbf{i_1} = \mathbf{X}_{i_1+k_1}\mathbf{W}_{k_1}$), where i_1 and k_1 appear in the first dim in **X**, ConvND ,dilated Conv, and Group Conv. All the above computations can be generated with our proposed framework. We now show that if a certain computation can be represented in the above format, then, it can be decomposed into several ConvND kernels.

We start from a simplified scenario that only involves indices i and k, e.g., $\mathbf{O}_{i_1,i_2} = \mathbf{X}_{k_1+i_1,k_2+i_2,k_3}\mathbf{W}_{k_1,k_2,k_3}$. Following Proposition 4.2, it can be naturally mapped to a ConvND operation between **X** and **W**. When the computation further involves indices m and j, e.g., a depth-wise Conv ($\mathbf{O}_{m_1,i_1} = \mathbf{X}_{m_1,i_1+k_1}\mathbf{W}_{m_1,k_1}$), as shown in Figure 6, for each specific value of m_1, e.g., $m_1 = 0$, the computation becomes $\mathbf{O}_{0,i_1} = \mathbf{X}_{0,i_1+k_1}\mathbf{W}_{0,k_1}$, which is the simplified scenario. This indicates *we can regard the complex computation as a combination of simplified scenarios*. Therefore, we can first encode each slice as separately a block and then, combine them together. Note we need to leave enough space to guarantee different slices are not interfering each other.

From the above discussion, each slice of **X** indexed by m and each slice of **W** indexed by m and j is encoded as a whole block. Thus, we only need to determine the arrangements of each block, including the sequence of blocks, denoted as $R_{x,m}$ and $R_{w,m,j}$ for **X** indexed by m and **W** indexed by m, j respectively, and the “starting position”, i.e. the slots with the lowest degree that this block takes up. We use the sequence of a block to index its starting position and denote as $S_{x,R_{x,m}}$ and $S_{w,R_{w,m,j}}$ respectively. An example is provided in Figure 6.

Remark From the above analysis, we derive that for most DNN operators, their computation can be decomposed to several ConvNDs between their slices, and thus the correct encoding of each slice is given in proposition 4.2 analytically and only the arrangements, i.e. the sequence and starting position of each slice remain undetermined.

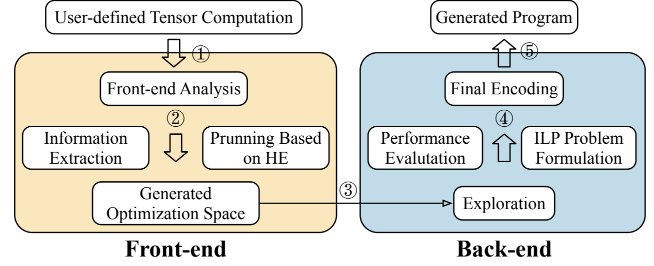

Figure 7. Overview of proposed FlexHE workflow.

5 Framework Design of FlexHE

5.1 Overview

Figure 7 illustrates the overall workflow of FlexHE. Users first define the tensor computations in mathematical forms using Python, and then register them as optimization tasks within FlexHE (①). The workflow of FlexHE can be divided into two parts: front-end analysis (Section 5.2) and back-end optimization (Section 5.3). The front-end analysis first extracts useful information, e.g., the number of indices and the category of each indices from the user input, and then aggregates the information to generate an optimization space based on Section 4 (②). The back-end search engine systematically explores the entire space to identify the most efficient encoding method (③ and ④). We formulate the search as a two-level optimization problem with the inner loop optimization formulated as an integer linear programming (ILP) problem. FlexHE leverages analytical cost modeling to evaluate each sample to improve search efficiency. Finally, based on the optimized encoding solution, FlexHE generates the low-level program (⑤).

5.2 Front-end Analysis

To derive the encoding method, we need to characterize tensor computations first. Therefore, FlexHE initially extracts useful information from the input program, including each indice with its range and the category of each index (i.e. in which tensor it appears). Figure 8(a) and (b) presents an example using depth-wise Conv1D.

Then, FlexHE generates the optimization space with the extracted information and some encoding primitives, including tiling, loop reordering, and elements reordering. Tiling primitive tiles the range of each indice into a smaller one such that the subtensor fits into the single polynomial. Loop reordering primitive changes the order of the indices, and thus influences the final encoding. Elements reordering primitive reorders the sequence of the elements encoded into the polynomial. We also observe that, though there are many other primitives like loop unrolling, they do not affect the encoding and, thus, are not considered.

The optimization space is then generated by systematically enumerating these primitives in a specific order. Each point in the optimization space is encoded using a vector, and each value in the vector represents a specific choice of encoding primitive or parameter. Figure 8(c) and (d) shows an example of the encoding primitives and a possible point in the optimization space.

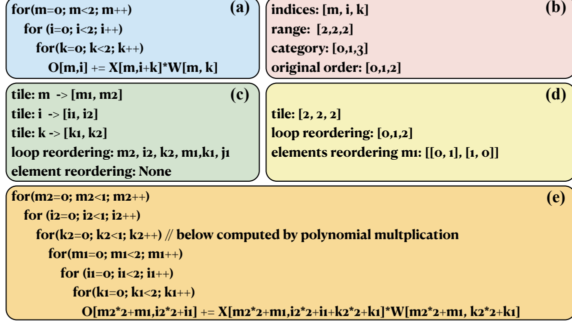

Figure 8. Depth-wise Conv1D example for information extraction. (a) User input program. (b) Extracted information, including the range of three indices, the category they belong to (0 for m, 1 for i, 2 for j, and 3 for k), and the original order of the loops (m, i, k represented by 0, 1, 2). (c) One possible encoding primitive. Three loops are all tiled to two levels; the outer loop and inner loop are all kept in the original sequence. Elements along m indice can be reordered. (d) Vectorization of the encoding primitives in (c). There are in total three indices m, i, and j in the user program. The tiling for each index is 2, 2, 2, which represents the range of the indices in inner loops (m1, i1, k1), and results in only one tile, i.e. the original tensor. Inner loops are kept in the original sequence. The m1 indice can be permuted, including two permutations [0, 1] and [1, 0]. (e) Diagram to show the loop structure after applying the encoding primitives. Inner loops (i1, j1, k1) are computed through polynomial multiplication.

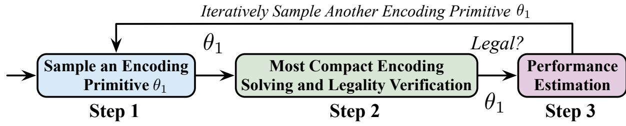

Figure 9. Diagram for exploration steps.

To enable efficient exploration of the huge optimization space, we propose to prune the points that are less likely to generate a good packing method. Here, we consider the following strategies:

1) Limit the depth of loop tiling to two: the polynomial abstraction divides the whole problem into two levels, the inner one represents polynomial multiplication while the outer one represents addition between different resultant polynomials;
2) limit the number of loop reordering: loop reordering for the outer loop does not affect the encoding. Though it may influence the data movement between cache and memory, it is hardware primitive and is not the focus of our paper;
3) limit the number of element reordering: we choose not to reorder the j index since it does not influence the performance and we only permute the m index.

5.3 Back-end Optimization

During the back-end optimization, FlexHE explores the optimization space to minimize total latency. The optimization target is formulated as

$$\textbf{min.}\ \text{Total_Latency}(\theta_1, \theta_2), \tag{3}$$

Table 2. Variables of encodings.

Notation	Description
L_x, L_w	Block length
$R_{x,m}, R_{w,m,j}$	Sequence for each block
$S_{x,a}, S_{w,a}$	Starting position for block with sequence a
N_x, N_w	Total number of blocks
N	Degree of polynomial

where θ_1 represents the encoding primitives including tiles, loop reordering, and elements reordering in Figure 8(d) and θ_2 represents the encoding of one polynomial listed in Table 2. For a specific encoding primitive, there exist many different encodings. Considering the polynomial length, some encodings will make the result exceed the polynomial length, which wraps around and corrupts the results and is thus illegal. Therefore, we need to verify, for a specific encoding primitive, if the most compact encoding is legal. We first sample an encoding primitive θ_1; find the most compact encoding under this primitive, and check if it is legal. The search process is then formulated as a two-level optimization problem with the outer one optimizing encoding primitives while the inner one finds the most compact encoding and verifies if it is legal.

Exploration process As shown in Figure 9, the exploration process begins with selecting θ_1 in the optimization space. This point is then used to establish the specific ILP problem (detailed in the next section) to find the most compact encoding. An ILP solver is then employed to solve this problem and check if it is legal. Upon solving, we acquire the exact encoding method. If the most compact encoding is legal, then an analytical estimator is used to derive the estimated cost under this encoding primitives. Otherwise, we would set the Latency as infinity. Afterward, we sample another point and iteratively optimize for the optimized encodings. We exploit a Bayesian optimization method supported by [2] to search for the optimal latency.

The analytical modeling is given as follows without proof due to limited space. The communication latency is estimated as:

$$(\Pi_{mik} \frac{N_m}{Tile_m} \frac{N_i}{Tile_i} \frac{N_k}{Tile_k} + \Pi_{mij} \frac{N_m}{Tile_m} \frac{N_i}{Tile_i} \frac{N_j}{Tile_j}) \ell_{comm},$$

where N_m, N_i, N_j, N_k is the range of each indices and $Tile_m$, $Tile_i$, $Tile_j$, $Tile_k$ represents the tiling of each indice. ℓ_{comm} represents the communication cost of one single polynomial. The computation is estimated as

$$\Pi_{ijmk} \frac{N_i}{Tile_i} \frac{N_j}{Tile_j} \frac{N_m}{Tile_m} \frac{N_k}{Tile_k} \ell_{comp},$$

where ℓ_{comp} represents the computation cost of one single polynomial multiplication. An example is given in Figure 10.

ILP problem formulation We then illustrate how to find the most compact encoding with an example in Figure 10. The encoding primitive is given in Figure 8(d).

We show the optimization procedure in Figure 10. All the variables are listed in Table 2. We first sample a sequence for the blocks. As changing the sequence of **X** blocks is equivalent to changing the sequence of **W** blocks, we only permute **W** blocks. Then, to find the most compact encoding, we minimize the slots taken up by the encodings as shown in Figure 10 (⑤).

$$\textbf{min.}\quad S_{x,N_x-1} + S_{w,N_w-1} + L_x + L_w. \tag{4}$$

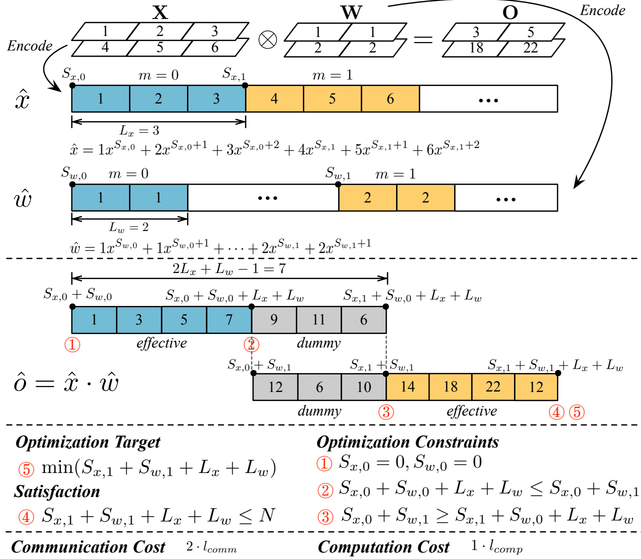

Figure 10. Examples for depth-wise Conv1D optimization. There are two blocks for **X** and **W** indexed by m, marked with blue block and yellow block, respectively. Polynomial multiplication convolves every block in $\hat{x}$ with $\hat{w}$ and results in an effective part that contains elements of **O** (convolved with the block with the same m) and a dummy part (convolved with the block with different m). The optimization target minimizes the total slots in the resulting polynomial (④). Constraint ② avoids overlapping between the blue effective block with the second result block. Constraint ③ avoids overlapping between the green effective block with the first block. The communication cost in this case is $2 \cdot \ell_{comm}$, including 1 input polynomial and 1 output polynomial. The computation cost in this case is $1 \cdot \ell_{comp}$, which corresponds to 1 polynomial multiplication.

Without loss of generality, let the first block of activation and weight start from 0, i.e. $S_{x,0} = 0$ and $S_{w,0} = 0$. A polynomial multiplication convolves every activation tensor block with the weight tensor block, producing an output block with length $N_x \cdot L_x + L_w - 1$ (e.g., $2L_x + L_w - 1$ in Figure 10) and a starting position $S_{w,R_{w,m,j}} + S_{x,0}$ (e.g., $S_{w,0} + S_{x,0}$ and $S_{w,1} + S_{x,0}$ in Figure 10). However, not all the elements in a block are useful. Only the part that is the result between blocks with the same m indices is effective and contains needed results in its coefficient, whose length is $L_x + L_w - 1$ and starts from $S_{w,R_{w,m,j}} + S_{x,R_{x,m}}$. For simplicity, we omit the -1 in the block length as it is usually far larger than 1.

To derive a correct packing, we need enough spacing to avoid overlapping between blocks. First, we need to ensure the effective block does not overlap with the block behind it in Figure 10 (②).

$$S_{x,R_{x,m}} + S_{w,R_{w,m,j}} + L_x + L_w \leq S_{x,0} + S_{w,R_{w,m,j}+1} \quad \forall\, m, j. \tag{5}$$

Second, we need to ensure the effective block doesn't overlap with the block in front of it as shown in Figure 10 (③).

$$S_{x,R_{x,m}} + S_{w,R_{w,m,j}} \geq S_{x,N_x-1} + S_{w,R_{w,m,j}-1} + L_x + L_w \quad \forall\, m, j.$$

Finally, we check if the result can be fit into one polynomial (④).

$$S_{x,N_x-1} + S_{w,N_w-1} + L_x + L_w \leq N. \tag{6}$$

Afterward, we sample another sequence for **W** and keep iterating until all sequences are touched or correct encoding is found.

6 Experimental Results

6.1 Experiment Setups

FlexHE is built on the top of the SEAL library [24], EMP toolkit [26], EzPC framework [4], and OpenCheetah [23] in C++. Consistent with [19, 21–23], we simulate a network setting via Linux Traffic Control. The bandwidth is set to 384MBps for LAN and 44MBps for WAN. The round-trip latency is about 0.3ms for LAN and 40ms for WAN. All the experiments are evaluated on an Intel Xeon CPU@2.2 GHz with 256GB RAM. We evaluate FlexHE on a variety of tensor computations, including matrix-vector multiply (GEMV), matrix-matrix multiply (GEMM), N-dimensional convolution (ConvND), depth-wise convolution (DEP), dilated convolution (DIL), and transposed N-dimensional convolution (TND). In our experiments, we use two environments, i.e., 1CPU+LAN and 4CPU+WAN as the computation-bound and communication-bound situations, respectively.

6.2 Micro-benckmark Evaluation

We compare the performance of FlexHE with both hand-optimized encodings, including Iron [10], Cheetah [11], Falcon [28], and SIMD compiler HElayers [1] for 11 different common operators. HElayers and hand-optimized encodings both lack support for Conv3D, DIL, T1D, T2D, and T3D, so we implement them based on Conv2D.

Results and analysis From Figure 11, we have the following observations: 1) FlexHE outperforms HElayers in all benchmarks under two environments. FlexHE achieves 19.9∼288.8× speedup under 1CPU+LAN and 1.1∼185.2× speedup under 4CPU+WAN. We believe the reasons for such significant speedups are due to the huge exploration space, unified optimization for environments, and the rotation-free advantage of coefficient encoding. 2) FlexHE achieves comparable and even higher performance over hand-optimized encodings on almost all benchmarks and achieves noticeably higher speedups on Conv3D, GRP, and T3D. FlexHE achieves 1.5∼1.7× speedup under 1CPU+LAN and 1.4∼1.5× speedup under 4CPU+WAN; 3) we also notice that FlexHE is unable to optimize for GEMM. The reason is that the selected test case is too lightweight such that the estimated computation latency is largely disturbed by other costs like thread launch time. We verify that the estimated cost of Iron encoding is indeed larger using our analytical estimator. However, even disturbed by inaccurate estimation, FlexHE still achieves comparable performance.

6.3 Evaluation for New Operators

We evaluate FlexHE on two uncommon operators under 4CPU+WAN, i.e., dilated Conv3D [27] (abbreviated as DiC3D) and depth-wise Conv3D [29] (abbreviated as DpC3D), which lack library support. We compare these two operators against the hand-implemented operators using the Conv2D kernel provided by hand-optimized encodings and HElayers.

Results and analysis As shown in Figure 12, we make the following conclusions: 1) For DiC3D, FlexHE achieves 1.4∼7.8× speedup compared to Cheetah and 168.7∼435.7× speedup compared to HElayers; 2) For DpC3D, FlexHE achieves 1.49∼4.17× speedup compared to Falcon and 5.69∼15.26× speedup compared to HElayers.

6.4 End-to-end Evaluation

We perform the end-to-end evaluation of ResNet-50 on the Imagenet dataset against Cheetah and HElayers. Since the 1×1 kernel is not

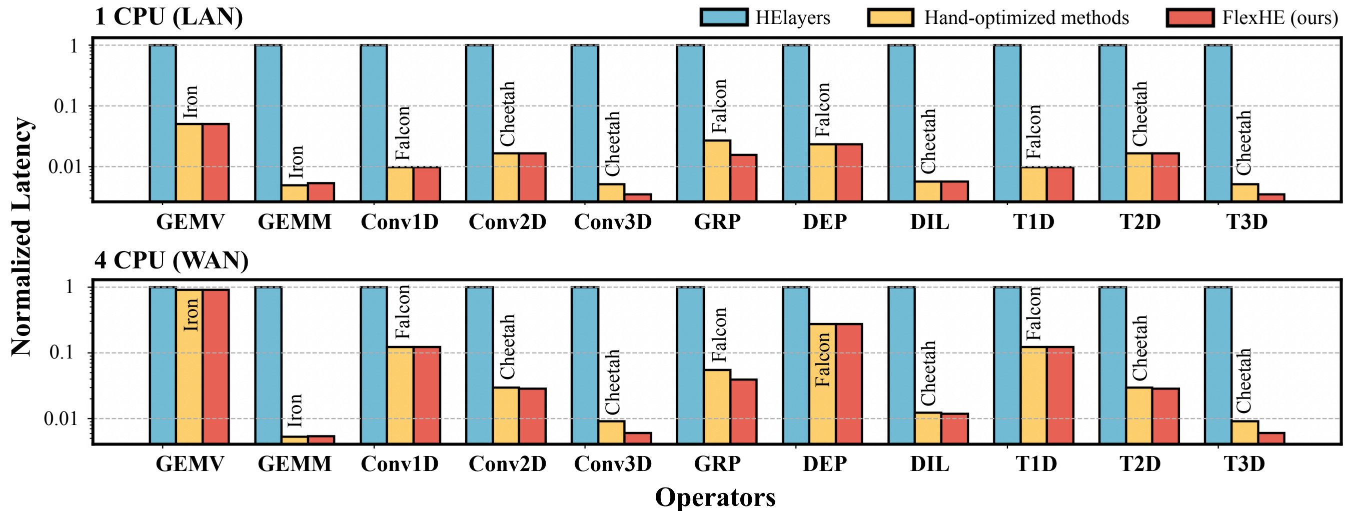

Figure 11. Micro-benchmark with HElayers and hand-optimized methods on different operators under different environments.

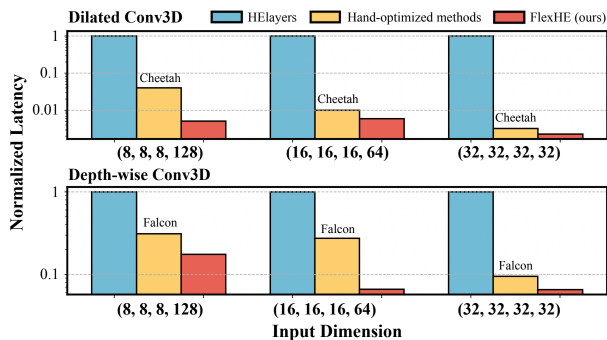

Figure 12. Benchmark with HElayers and hand-optimized methods on dilated Conv3D and depth-wise Conv3D.

supported by HElayers, we implement the point-wise convolutions with linear kernels.

Results and Analysis As shown in Table 3, we have the following observations: 1) Under 4CPU+WAN, FlexHE achieves 6.2× and 1.2× total latency reduction compared to HElayers and Cheetah, respectively; 2) under 1CPU+LAN, FlexHE even achieves a more significant improvement on total latency with 11× and 1.1× reduction, respectively; 3) compared with HElayers, we notice that the efficiency improvement of end-to-end inference is not as large as the Conv2D as shown in Section 6.2. This is because there are many point-wise convolutions in ResNet-50, which are implemented by linear kernels with a lower speedup, especially under 4CPU+WAN.

6.5 Optimization Performance

We also show the exploration process of Conv3D and GRP in Figure 13. We run 1500 trails in total in about 15 minutes.

Results and analysis As can be observed, the exploration process converges quickly for both operations. The reason for fast convergence is two-fold: 1) the whole encoding space is reduced to a smaller space where most points are legal points so that most samples are correct encodings; 2) inner ILP optimization can solve for dense encoding efficiently and eliminate searching for spacing, which is inefficient.

Table 3. End-to-end evaluation on ResNet-50.

Environment	Method	End-to-end Lat. (s)	Comm. (MB)
4CPU+WAN	HElayers	683.96	627.80
	Cheetah	132.03	1312.1
	FlexHE (ours)	110.71	539.65
1CPU+LAN	HElayers	3032.1	627.80
	Cheetah	300.05	1312.1
	FlexHE (ours)	271.94	539.65

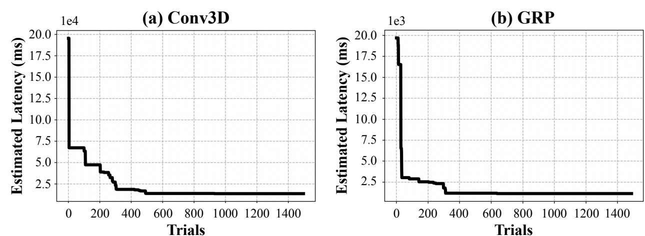

Figure 13. Performance v.s. exploration trails.

7 Conclusion

In this work, we propose FlexHE, a flexible kernel generation framework to enable automatic generation and optimization of HE kernels for 2PC-based private inference. FlexHE automatically defines the kernel design space based on a high-level description of DNN operations. FlexHE formulates a two-level optimization problem to search for the latency-efficient encoding. With extensive experiments, FlexHE demonstrates better coverage and higher latency reduction compared to prior-art HElayers, Cheetah, and Falcon.

References

[1] Ehud Aharoni, Allon Adir, Moran Baruch, Nir Drucker, Gilad Ezov, Ariel Farkash, Lev Greenberg, Ramy Masalha, Guy Moshkowich, Dov Murik, Hayim Shaul, and Omri Soceanu. 2023. HeLayers: A Tile Tensors Framework for Large Neural Networks on Encrypted Data. *Proceedings on Privacy Enhancing Technologies* 2023, 1 (Jan. 2023), 325–342. https://doi.org/10.56553/popets-2023-0020

[2] Takuya Akiba, Shotaro Sano, Toshihiko Yanase, Takeru Ohta, and Masanori Koyama. 2019. Optuna: A Next-generation Hyperparameter Optimization Framework. arXiv:1907.10902 [cs.LG]

[3] Song Bian, Zhou Zhang, Haowen Pan, Ran Mao, Zian Zhao, Yier Jin, and Zhenyu Guan. 2023. HE3DB: An Efficient and Elastic Encrypted Database Via Arithmetic-And-Logic Fully Homomorphic Encryption. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*. 2930–2944.

[4] Nishanth Chandran, Divya Gupta, Aseem Rastogi, Rahul Sharma, and Shardul Tripathi. 2019. EzPC: Programmable and efficient secure two-party computation for machine learning. In *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 496–511.

[5] Jung Hee Cheon, Andrey Kim, Miran Kim, and Yongsoo Song. 2017. Homomorphic encryption for arithmetic of approximate numbers. In *Advances in Cryptology–ASIACRYPT 2017: 23rd International Conference on the Theory and Applications of Cryptology and Information Security, Hong Kong, China, December 3-7, 2017, Proceedings, Part I 23*. Springer, 409–437.

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

[7] Roshan Dathathri, Olli Saarikivi, Hao Chen, Kim Laine, Kristin Lauter, Saeed Maleki, Madanlal Musuvathi, and Todd Mytkowicz. 2019. CHET: an optimizing compiler for fully-homomorphic neural-network inferencing. In *ACM SIGPLAN PLDI*. 142–156.

[8] Keke Gai, Meikang Qiu, Yujun Li, and Xiao-Yang Liu. 2017. Advanced fully homomorphic encryption scheme over real numbers. In *2017 IEEE 4th international conference on cyber security and cloud computing (CSCloud)*. IEEE, 64–69.

[9] Karthik Garimella, Zahra Ghodsi, Nandan Kumar Jha, Siddharth Garg, and Brandon Reagen. 2023. Characterizing and Optimizing End-to-End Systems for Private Inference. In *ACM ASPLOS (ASPLOS '23)*. ACM. https://doi.org/10.1145/3582016.3582065

[10] Meng Hao, Hongwei Li, Hanxiao Chen, Pengzhi Xing, Guowen Xu, and Tianwei Zhang. 2022. Iron: Private inference on transformers. *Advances in Neural Information Processing Systems* 35 (2022), 15718–15731.

[11] Zhicong Huang, Wen-jie Lu, Cheng Hong, and Jiansheng Ding. 2022. Cheetah: Lean and fast secure Two-Party deep neural network inference. In *31st USENIX Security Symposium (USENIX Security 22)*. 809–826.

[12] Chiraag Juvekar, Vinod Vaikuntanathan, and Anantha Chandrakasan. 2018. Gazelle: A Low Latency Framework for Secure Neural Network Inference. *arXiv:1801.05507 [cs]* (2018). http://arxiv.org/abs/1801.05507

[13] Igor Kononenko. 2001. Machine learning for medical diagnosis: history, state of the art and perspective. *Artificial Intelligence in medicine* 23, 1 (2001), 89–109.

[14] Neeraj Kumar, Ritu Chauhan, and Gaurav Dubey. 2020. Applicability of financial system using deep learning techniques. In *Ambient Communications and Computer Systems: RACCCS 2019*. Springer, 135–146.

[15] Eunsang Lee, Joon-Woo Lee, Junghyun Lee, Young-Sik Kim, Yongjune Kim, Jong-Seon No, and Woosuk Choi. 2022. Low-complexity deep convolutional neural networks on fully homomorphic encryption using multiplexed parallel convolutions. In *International Conference on Machine Learning*. PMLR, 12403–12422.

[16] Jian Liu, Mika Juuti, Yao Lu, and N. Asokan. 2017. Oblivious Neural Network Predictions via MiniONN Transformations. *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security* (2017). https://api.semanticscholar.org/CorpusID:3617652

[17] Ximeng Liu, Lehui Xie, Yaopeng Wang, Jian Zou, Jinbo Xiong, Zuobin Ying, and Athanasios V Vasilakos. 2020. Privacy and security issues in deep learning: A survey. *IEEE Access* 9 (2020), 4566–4593.

[18] Fatemehsadat Mireshghallah, Mohammadkazem Taram, Praneeth Vepakomma, Abhishek Singh, Ramesh Raskar, and Hadi Esmaeilzadeh. 2020. Privacy in deep learning: A survey. *arXiv preprint arXiv:2004.12254* (2020).

[19] Pratyush Mishra, Ryan Lehmkuhl, Akshayaram Srinivasan, Wenting Zheng, and Raluca Ada Popa. 2020. Delphi: A Cryptographic Inference Service for Neural Networks. In *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, 2505–2522. https://www.usenix.org/conference/usenixsecurity20/presentation/mishra

[20] Payman Mohassel and Yupeng Zhang. 2017. Secureml: A system for scalable privacy-preserving machine learning. In *2017 IEEE symposium on security and privacy (SP)*. IEEE, 19–38.

[21] Deevashwer Rathee, Mayank Rathee, Rahul Kranti Kiran Goli, Divya Gupta, Rahul Sharma, Nishanth Chandran, and Aseem Rastogi. 2021. Sirnn: A math library for secure rnn inference. In *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1003–1020.

[22] Deevashwer Rathee, Mayank Rathee, Nishant Kumar, Nishanth Chandran, Divya Gupta, Aseem Rastogi, and Rahul Sharma. 2020. CrypTFlow2: Practical 2-party secure inference. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*. 325–342.

[23] Brandon Reagen, Woo-Seok Choi, Yeongil Ko, Vincent T. Lee, Hsien-Hsin S. Lee, Gu-Yeon Wei, and David Brooks. 2021. Cheetah: Optimizing and Accelerating Homomorphic Encryption for Private Inference. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. 26–39. https://doi.org/10.1109/HPCA51647.2021.00013

[24] SEAL 2023. Microsoft SEAL (release 4.1). https://github.com/Microsoft/SEAL. Microsoft Research, Redmond, WA..

[25] Andrei Stoian, Jordan Frery, Roman Bredehoft, Luis Montero, Celia Kherfallah, and Benoit Chevallier-Mames. 2023. Deep neural networks for encrypted inference with tfhe. In *International Symposium on Cyber Security, Cryptology, and Machine Learning*. Springer, 493–500.

[26] X. Wang, A. J. Malozemoff, and J. Katz. 2016. EMP-toolkit: Efficient MultiParty computation toolkit. https://github.com/emp-toolkit.

[27] Zijian Wang, Yaoru Sun, Qianzi Shen, and Lei Cao. 2019. Dilated 3D Convolutional Neural Networks for Brain MRI Data Classification. *IEEE Access* 7 (2019), 134388–134398. https://doi.org/10.1109/ACCESS.2019.2941912

[28] Tianshi Xu, Meng Li, Runsheng Wang, and Ru Huang. 2023. Falcon: Accelerating Homomorphically Encrypted Convolutions for Efficient Private Mobile Network Inference. *arXiv preprint arXiv:2308.13189* (2023).

[29] Rongtian Ye, Fangyu Liu, and Liqiang Zhang. 2018. 3D Depthwise Convolution: Reducing Model Parameters in 3D Vision Tasks. arXiv:1808.01556 [cs.CV]

[30] Wenyi Zhao, Rama Chellappa, P Jonathon Phillips, and Azriel Rosenfeld. 2003. Face recognition: A literature survey. *ACM computing surveys (CSUR)* 35, 4 (2003), 399–458.