HEILP: An ILP-based Scale Management Method for Homomorphic Encryption Compiler

Weidong Yang, Shuya Ji, Jianfei Jiang, Naifeng Jing, Qin Wang, Zhigang Mao and Weiguang Sheng*
Department of Micro/Nano Electronics, Shanghai Jiao Tong University, Shanghai, China
Email: wgshenghit@sjtu.edu.cn

***Abstract*—RNS-CKKS, a fully homomorphic encryption (FHE) scheme, enabling secure computation on encrypted data, has widely be used in statistical analysis and data mining. However, developing RNS-CKKS programs requires substantial knowledge of cryptography, which is unfriendly to non-expert programmers. A critical obstacle is the scale management, which affects the complexity of programming and performance. Different FHE operations impose specific requirements on the scale and level, necessitating programmer intervention to ensure the recoverability of the results. Furthermore, operations at different levels have a significant impact on program performance. Existing methods rely on heuristic insights or iterative methods to manage the scales of ciphertexts. However, these methods lack a holistic understanding of the optimization space, leading to inefficient exploration and suboptimal performance. This work proposes HEILP, the first constrained-optimization-based approach for scale management in FHE. HEILP expresses node scale decision and scale management operation inserting as an integer linear programming model which can be solved with existing mathematical techniques in one shot. Our method creates a more comprehensive optimization space and enables a faster and more efficient exploration. Experimental results demonstrate that HEILP achieves an average performance improvement of 1.72× over existing heuristic method, and outperforms a 1.19× performance improvement with 48.65× faster compilation time compared to the state-of-the-art iteration-based method.**

***Index Terms*—Homomorphic encryption, CKKS, scale management, compilation optimizations, integer linear programming**

I. Introduction

Past years have experienced a remarkable increase in cloud computing service. More and more businesses and organizations migrate their data and services to the cloud, driving a surge in the demand for secure computing. Fully Homomorphic Encryption (FHE) [1] enables computations to be performed on encrypted data without the need for decryption, reducing the risk of data exposure in cloud computing. Among various FHE schemes, RNS-CKKS [2] has shown remarkable performance, especially in machine learning, statistical analysis and data mining. Due to its excellent performance, RNS-CKKS has been widely used in various FHE libraries [3], [4] and compilers [5]–[10].

Despite the benefits of FHE, writing a correct and efficient FHE application remains complex and challenging since existing FHE schemes require in-depth cryptography knowledge such as scale management to achieve desired result recoverability and high performance. RNS-CKKS encodes real numbers as integers with the *scale* representing the position of decimal point. Ciphertext multiplication increases the scale of ciphertexts and the scale overflow would lead to an unrecoverable computation result. Moreover, ciphertexts have an attribute called *level* (detailed in Section II), which affects the latency of operations performed on ciphertexts. To develop a correct and efficient application, programmers should manually insert scale management operations such as *rescale* and *modswitch* to determine the scale and level for ciphertexts while respecting several constraints. This unique programming paradigm requires in-depth cryptographic expertise and presents a significant barrier for non-experts.

To alleviate the programming burden, EVA [6], Hecate [7] and ELASM [8] proposed automatic scale management methods. However, these methods are based on the heuristics or iterative methods, leading to suboptimal performance. EVA [6] inserts scale management operations only for correctness and ignore the impact on latency. Additionally, it employs a separated scale and level analysis, failing to recognize the potential benefits of solving these two issues jointly. Hecate [7] and ELASM [8] achieve higher performance through iteration-based methods to explore the optimization space. Compared with Hecate [7], ELASM [8] can find solutions with better error and latency tradeoff by the fine-grained noise-aware waterline. However, the quality of solutions obtained from iterative methods varies significantly based on multiple factors such as algorithm parameters, initial solutions, and randomness. Besides, they use rewriting rules to insert scale management operations, which limits the solution space.

In this work, we propose HEILP, a constrained-optimization-based scale management method to automatically generate high-performance results for FHE. HEILP expresses the scale management problem as a integer linear programming (ILP) model, which can be solved through current optimization libraries and is advantageous in expression integrity, portability, and solution quality. By defining constraints covering both arithmetic and scale management operations, HEILP establishes a broader solution space and resolves the scale management challenge without stochastic explorations.

The main contributions of our work are as follows:

- We formulate scale management problem for RNS-CKKS as an ILP model that can be solved in one shot. To the best of our knowledge, HEILP is the first constrained-optimization-based approach to tackle scale decisions.
- We integrated this approach into the MLIR [11] framework

This work was supported by National Key R&D Program of China (Grant No. 2023YFB4403500). *Corresponding author: Weiguang Sheng (wgshenghit@sjtu.edu.cn).

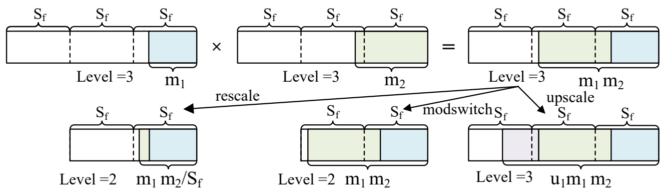

Fig. 1. The diagram of scale, level and scale management operations in RNS-CKKS.

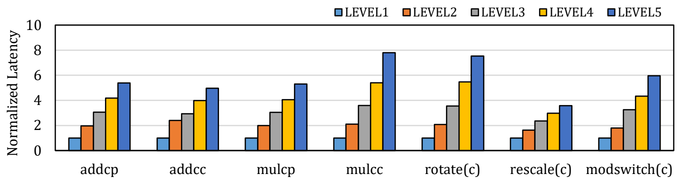

Fig. 2. Latency comparison of RNS-CKKS operations from level 1 to 5.

to achieve automatic scale management.

- Experiments show that HEILP achieves an average performance gain of $1.72\times$ over heuristic-based method and $1.19\times$ performance improvement over iteration-based approach, while reducing time-to-solution by $48.65\times$.

II. Background

A. Homomorphic Encryption and RNS-CKKS

Homomorphic encryption means that there is a homomorphism between operations on plaintexts and operations on ciphertexts, allowing a third party to perform computations on encrypted data. In other words, if we have an operation (e.g., $+$) on the plaintexts then there is an operation $\oplus$ on the ciphertexts so that Dec(Enc($x+y$)) = Dec(Enc(x) $\oplus$ Enc(y)).

RNS-CKKS [2], constructed on the Ring Learning with Errors (RLWE) problem, has attracted widespread interest owing to its excellent performance. To develop an RNS-CKKS program, the following steps should be followed. Firstly, the polynomial modulus degree N and coefficient modulus Q should be determined by programmers according to required security level. Then the essential cryptographic keys are generated, including public, private, relinearization, and rotation keys. The plaintext data is encoded to a polynomial, and subsequently encrypted to a ciphertext. Evaluation supports an arbitrary mix of additions and multiplications for ciphertexts and plaintexts (*mulcc*, *mulcp*, *addcc*, *addcp*). Since *mulcc* increases the number of polynomials, leading to a significant rise in computational overhead and noise, a *relinearization* operation is implicitly added after every *mulcc* to control the number of polynomials. Besides, as RNS-CKKS supports vectorization which allows for the simultaneous processing of multiple data elements within a single ciphertext, it provides *rotate* operations to shift every element within the encrypted vector. Finally, following decryption and decoding, the output message is obtained, as if the computation was performed on unencrypted data, thereby achieving homomorphic encryption.

B. Scale management

As previously mentioned, ciphertexts in RNS-CKKS possess scale and level properties. For example, with the scale is 1000, the real number $m = 0.01$ is encoded as $m' = 10$ (an integer). As depicted in Fig. 1, when ciphertexts with scales $m1$ and $m2$ are multiplied, the scale of the resulting ciphertext is $m1 \times m2$. Coefficient modulus Q represents the scale capacity of a ciphertext, which is the product of small prime modulus (called rescaling factors) S_f. The level l indicates the number of S_f available in Q [8] (can also denote the number of consumed S_f [7]), i.e., $Q \approx {S_f}^l$. As shown in top of Fig. 1, the input ciphertext is divided into three cells, indicating a level of 3. Each cell has a size of S_f, denoting that the scale upper limit of the current ciphertext is ${S_f}^3$.

In the RNS-CKKS scheme, it is essential to keep the scale below ${S_f}^l$. Otherwise, the result would be unrecoverable. To prevent scale overflow, RNS-CKKS provides a *rescale* operation that divides current scale and coefficient modulus Q of a ciphertext by S_f, thereby decreasing the level l by 1. In addition, RNS-CKKS also provides a *modswitch* operation to reduce the level while leaving the scale unchanged. On the other hand, a low scale may lead to irreversible loss of accuracy in the computed result, with the risk of data corruption in extreme cases [6]. Therefore, it is a common practice to ensure that the scale remains above a minimum threshold referred to as the waterline. *Upscale* is to change the scale by multiplying a plaintext without changing the level. Note that since *upscale* is essentially a *mulcp* operation, it can only increase the scale but not decrease the scale. Fig. 1 depicts the impact of the three scale management operations on the scale and level of ciphertexts.

In addition to requirement of correctness, the level l also affects the latency of the operation because it determines the size of ciphertexts [12]. Fig. 2 shows the latency comparison for different RNS-CKKS operations at different operand levels when $N = 2^{15}$ and $S_f = 2^{60}$ [9]. *Mulcc* and *rotate* operations, which dominate the majority of program runtime, are particularly sensitive to changes in the level. In summary, an automated scale management method requires: (a) determining the scale and level for all operations, and (b) inserting appropriate scale management operations, thus generating a correct and efficient FHE program.

III. The HEILP framework

A. HEILP overview

Our proposed HEILP is implemented based on the MLIR framework [11]. Owing to the MLIR, we are able to define customized dialects to offer suitable abstractions and efficiently facilitate optimizations at proper levels. The dialect and optimization passes involved are illustrated in Fig. 3. The passes colored in pink denote those utilized in the current work, while the blue passes indicate those to be incorporated in the future work. In view of the main application scenarios of RNS-CKKS, Python is chosen as the current programming language. Other programming languages such as C will be supported in the future. Besides arithmetic operations, branches and loops with

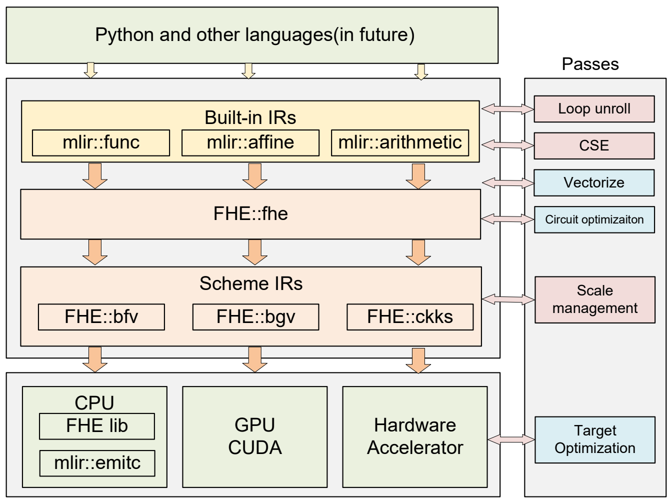

Fig. 3. The block diagram of the overall compilation flow.

plaintexts as conditions are also supported. Ciphertexts can not be used as conditions, otherwise the data of ciphertexts would leak from the control flow [13].

The input program is transformed into the high-level IR which consists of a variety of builtin dialects. These dialects allow high-level program optimization to reduce semantic loss. In this work, loop unrolling is adopted since underlying FHE schemes do not support loop. After loop unrolling, there are other common compilation optimizations such as dead code elimination, constant folding, common sub-expression elimination and so on. The `fhe` dialect defines the operations and the semantics of FHE programs, while abstracting the lower scheme detail such as data type and noise management. Vectorization is a crucial way to improve throughput and the circuit optimization concentrates on reducing the multiplication depth to decrease latency and control noise growth. This work does not focus on these two optimizations, more details can be seen in [14]–[16] and [17]–[19], respectively. Then the program is lowing to scheme-specific dialects including `BFV`, `BGV` and `CKKS`. The scale management optimization which this work focus on is carried on this abstract representation. Finally, the lower-level IR would be lowing into different hardware backends including CPU, GPU and accelerators with hardware-dependent optimization such as [12], [20]. In this work, we target SEAL [3], a common CPU library for FHE. The lower-level IR can also be lowed to `mlir::emitc` to integrate with other programs.

B. Variables and Constants Definition

Constant Parameters: Before the scale management, the program can be viewed as a directed acyclic graph $G(V, E)$. V denotes all nodes in origin programs without additional scale management operations. We use V_{op} to express the node sets of the specific operation op, i.e., $V = V_{add} \cup V_{mul} \cup V_{rotate} \cup V_{input} \cup V_{output}$. Besides, we use $(e1, e2, dst)$ and $(e1, dst)$ to represent an arithmetic node accurately, where $e1$ and $e2$ denote the input edges, and dst denotes the arithmetic node itself. In our notation, J signifies the set of levels, ranging from 1 to L.

TABLE I
List of symbols for ILP model construction

Symbol	Type	Description
s_i	Integer	The scale (logarithmic form) of node i.
$l_{i,j}$	Boolean	Whether node i is at level j.
is_k	Integer	The scale (logarithmic form) of the virtual node of edge k.
$il_{k,j}$	Boolean	Whether the virtual node of edge k is at level j.
$r_{k,j,n}$	Boolean	Whether there is n $rescale$ operations at level j on the edge k.
$m_{k,j,n}$	Boolean	Whether there is n $modswitch$ operations at level j on the edge k.
u_k	Integer	The $upscale$ factor on the edge k.

R is used to express the logarithmic value of rescaling factor, i.e. $2^R = S_f$. $D_{op,j}$ indicates the delay of operation op at level j.

Variable Representation: HEILP formalizes the scale management problem as a constraint-optimization problem. The symbols and description of the variables are displayed in TABLE I. The target of scale management is to determine the scale and level of all arithmetic nodes and insert proper scale management operations to minimize the latency while meeting RNS-CKKS constraints.

The decision variables of the problem are the scale and level of each node (including the input and output nodes) in origin programs, denoted as s_i and $l_{i,j}$. The scale is expressed in logarithmic form to transform variable multiplication to variable addition. On the other hand, the level property is expressed in boolean form to facilitate the operation delay. Fig. 4 (a) shows an example of a multiplication of two ciphertexts and a subsequent $rotate$ operation. For instance, the two input ciphertexts have scales of 2^{20} and 2^{30}, respectively, with both being assigned a level of 3. Note that Fig. 4 (a) is insufficient to generate a correct FHE program. It is necessary to incorporate the scale management operations as depicted in Fig. 4 (c).

For the convenience of subsequent representations, we introduce some auxiliary variables. Assume that the output of each edge has a virtual node (dot circle in Fig. 4 (b)) and there are no arithmetic or scale management operations between the virtual node and the destination node of the edge. This virtual node also has scale and level property (is_k and $il_{k,j}$), representing the input attribute of the destination node.

The scale management operations, denoted as $r_{k,j,n}$,$m_{k,j,n}$ and u_k, are inserted between the virtual node and the source node of the edge. We introduce the order in which the three types of operations are added to an edge. Since $rescale$ has the larger latency than other two operations, reducing its level offers significant performance gains. Therefore, it is placed after $modswitch$ operations which can decrease the level. Furthermore, if $rescale$ is performed before upscale, the precision loss caused by reduced scale is irretrievable. As $upscale$ increases the scale, placing it before $modswitch$ may result in overflow, it is inserted after $modswitch$. Consequently, the final order is $modswitch \rightarrow upscale \rightarrow rescale$. Note that the *downscale* in HECATE [7] is composed of a $upscale$ operation and a $rescale$ operation, which is included in our method. As shown in Fig. 4 (c), on edges (0,2) and (1,2), a $modswitch$ operation is inserted, thereby achieving a level of

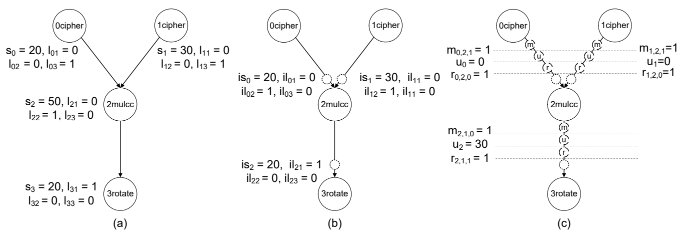

Fig. 4. The diagram of variables in HEILP (not all variables listed).

2 at virtual nodes 0 and 1, while the scale remains unchanged. On edge (2,3), the scale is increased by 2^{30} through an $upscale$ operation, and subsequently reduced by 2^{60} via a $rescale$ operation, which lowers the level by 1.

C. Constraints

After the definition of these variables and constants, we introduce the following constraints to represent the feasible scale management results.

1) Node level allocation Constraint: All nodes, including every node in V and all nodes on edges $k \in E$, can only be assigned to exactly one level.

$$\forall i \in V, \sum_{j \in J} l_{i,j} = 1 \tag{1}$$

$$\forall k \in E, \sum_{j \in J} il_{k,j} = 1 \tag{2}$$

$$\forall k \in E, \sum_{j \in J} \sum_{n \in J} m_{k,j,n} = 1 \tag{3}$$

$$\forall k \in E, \sum_{j \in J} \sum_{n \in J} r_{k,j,n} = 1 \tag{4}$$

2) Node scale upper bound Constraint: In order to avoid scale overflow which leads to unrecoverable results, the scale of nodes (including all arithmetic operation nodes and virtual nodes) should meet the following requirements:

$$\forall i \in V, s_i \leq \sum_{j \in J} j \times l_{i,j} \times R \tag{5}$$

$$\forall k \in E, is_k \leq \sum_{j \in J} j \times il_{k,j} \times R \tag{6}$$

On a given edge, since the worst case occurs at the virtual node, ensuring that the scale and level of the virtual node meet the constraint implies that all nodes along the edge can satisfy this constraint.

3) Node scale lower bound Constraint: As mentioned above, the scale should meet the waterline constrain to keep the relative error lower for better quality of service. Because the minimum scale along an edge is located at either the virtual node or the source node, ensuring that these two nodes fulfill the constraints is sufficient.

$$\forall i \in V, s_i \geq waterline \tag{7}$$

$$\forall k \in K, is_k \geq waterline \tag{8}$$

Note that we can use various waterlines such as coarse-grained fixed waterlines in EVA [6] and fine-grained dynamic waterlines proposed in ELASM [8].

4) Arithmetic Operation Constraint: Different arithmetic operations have varying requirements for the input scale and level, and generate outputs with different rules.

The multiplication operation ($mulcp$ and $mulcc$) requires that the two input virtual nodes are at the same level. After multiplication, the level of the result is unchanged, and the scale becomes the product of the two input scales (in logarithmic representation, it is the sum).

$$\forall (e1, e2, dst) \in V_{mul}, \forall j \in J, il_{e1,j} = il_{e2,j} \tag{9}$$

$$\forall (e1, e2, dst) \in V_{mul}, \forall j \in J, l_{dst,j} = il_{e1,j} \tag{10}$$

$$\forall (e1, e2, dst) \in V_{mul}, s_{dst} = is_{e1} + is_{e2} \tag{11}$$

The addition operation (including $addcp$ and $addcc$) requires that the two input virtual nodes are at the same level and possess identical scale. The level and scale of the calculation result would be unchanged.

$$\forall (e1, e2, dst) \in V_{add}, is_{e1} = is_{e2} \tag{12}$$

$$\forall (e1, e2, dst) \in V_{add}, \forall j \in J, il_{e1,j} = il_{e2,j} \tag{13}$$

$$\forall (e1, e2, dst) \in V_{add}, s_{dst} = is_{e1} \tag{14}$$

$$\forall (e1, e2, dst) \in V_{add}, \forall j \in J, l_{dst,j} = il_{e1,j} \tag{15}$$

The level and scale of the result produced by the $rotate$ operation remain unchanged.

$$\forall (e1, dst) \in V_{rotate}, s_{dst} = is_{e1} \tag{16}$$

$$\forall (e1, dst) \in V_{rotate}, \forall j \in J, l_{dst,j} = il_{e1,j} \tag{17}$$

5) Scale management operation Constraint:

Rescale Operation Constraint: Since only $rescale$ operations can reduce the scale among three operations, its number depends on the difference between the scale of the source node and the scale of the virtual node on an edge.

$$\forall k = (src, dst) \in E, \sum_{j \in J} \sum_{n \in J} r_{k,j,n} \times n \times R \geq s_{src} - is_k \tag{18}$$

$$\forall k = (src, dst) \in E, \sum_{j \in J} \sum_{n \in J} r_{k,j,n} \times n \times R \leq s_{src} - is_k + R - 1 \tag{19}$$

The level of $rescale$ operation is the same as that of the virtual node.

$$\forall k \in E, \sum_{j \in J} il_{k,j} \times j = \sum_{j \in J} \sum_{n \in J} r_{k,j,n} \times j \tag{20}$$

Modswitch Operation Constraint: On a given edge, the sum of the number of $modswitch$ and the number of $rescale$ determine the amount of level reduction.

$$\sum_{j\in J} j \times l_{src,j} - \sum_{j\in J} j \times il_{k,j} = \sum_{j\in J}\sum_{n\in J} r_{k,j,n} \times n + \sum_{j\in J}\sum_{n\in J} m_{k,j,n} \times n$$
$$\forall k = (src, dst) \in E \tag{21}$$

Upscale Operation Constraint: The $upscale$ operation determines the scale increase along an edge.

$$\forall k = (src, dst) \in E, u_k = is_k - s_{src} + \sum_{j\in J}\sum_{n\in J} r_{k,j,n} \times n \times R \tag{22}$$

6) Input and Output Constraint: The input and output nodes have user-defined scales, with the input ciphertexts operating at the maximum level L.

$$\forall i \in V_{input,output}, S_i = scale_i; \tag{23}$$

$$\forall i \in V_{inputcipher}, l_{i,L} = 1; \tag{24}$$

D. Objective Functions

Delay of arithmetical operations: $ALLOP$ is a set consists of $addcc, addcp, mulcc, mulcp, rotate$. The latency of arithmetic operations is

$$D_a = \sum_{op\in ALLOP}\sum_{i\in V_{op}}\sum_{j\in J} D_{op,j} \times l_{i,j} \times num_i \tag{25}$$

Note that we introduce num only for $rotate$ operation. Since rotating vectors by different amounts requires distinct public rotation keys, FHE libraries usually support rotations only by powers of 2 [5]. If the rotation amount is not an integer power of 2, it is necessary to calculate the number of required $rotate$ operations.

Delay of scale management operations: The latency of the $upscale$ operation has been excluded from consideration, since incorporating it necessitates variable multiplication, which would rapidly expand the problem size. Moreover, the latency of $upscale$ is not a significant component of the total latency, making this omission a reasonable simplification.

$$D_s = \sum_{k\in E}\sum_{n\in J}\sum_{j\in J} (D_{mod,n,j} \times m_{k,j,n} + D_{res,n,j} \times r_{k,j,n}) \tag{26}$$

Optimization target: The target of this problem is to minimize:

$$D_{all} = D_a + D_s \tag{27}$$

IV. Experimental results

A. Experimental Setup

To evaluate the efficiency of our approach, we select 10 kernels from [8] and [19] as benchmarks including Sobel filter detection (SF) and Harris corner detection (HCD) in image processing, linear regression (LR), polynomial regression (PR) and multivariate regression (MR) in statistical machine learning, Multilayer Perceptron (MLP) in neural network, retrieval (RT) and set-union (SU) in private information. The number after RT and SU represents the number of key-value pairs. All kernels are already vectorized before experiments. To compare the performance and compilation time, we selected a heuristic method (EVA [6]) and an iteration-based method (ELASM [8]) as baselines. ELASM explores 12000 scale management plan samples just like [8] and its optimization goal is modified to minimize latency. For all three methods, we use the same encryption parameter. The rescaling factor S_f is set to 2^{60} and polynomial modulus N is set to 2^{15} to obtain the 128-bit security level. HEILP uses Gurobi [21] as the solver.

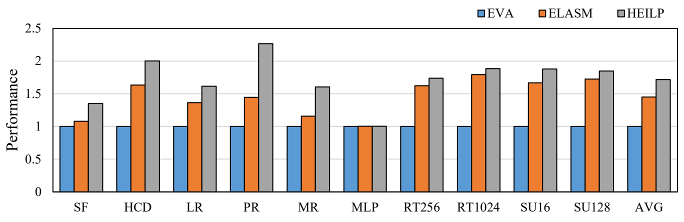

Fig. 5. Performance comparison of different scale management methods.

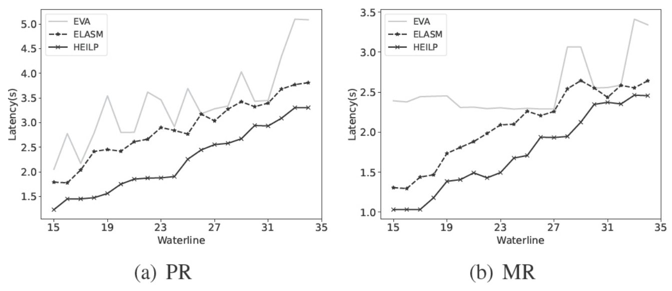

Fig. 6. Performance comparison of different scale management methods for different waterlines.

B. Performance comparison

Fig. 5 shows the performance comparison of different automatic scale management methods with given waterlines. Performance means the reciprocal of the running time for the target task on CPUs. All results are from actual execution, rather than from estimation. Every generated program is executed 50 times to compute the average runtime. Compared with the heuristic method and iteration-based method, the constrained optimization method proposed in this work is easier to find better solutions. Specifically, HEILP can achieve an average performance improvement of 1.72× and 1.19× compared to EVA and ELASM. The most pronounced performance improvements are observed in PR and MR, because previous methods are unable to accurately represent the solution space for these two kernels. Moreover, since the input image in MLP is the user-provided ciphertext and the weight is the plaintext provided by the server, there is almost no $mulcc$ operations in this kernel. This results in a tiny solution space, rendering the performance differences among three methods negligible.

TABLE II
COMPILATION TIME (SECOND) WITH DIFFERENT METHODS

Kernel	EVA [6]	ELASM [8]	HEILP	Speedup
SF	0.0565	8.2701	0.108	76.575×
HCD	0.0619	12.9187	0.2808	46.0068×
LR	0.0647	13.4591	0.2948	45.6550×
PR	0.0817	30.3322	4.1366	7.3326×
MR	0.1094	28.4936	2.125	13.4088×
MLP	0.0916	44.1797	0.242	182.5607×
RT256	0.0557	5.114	0.1683	30.3862×
RT1024	0.0527	5.4431	0.081	67.1988×
SU16	0.0575	5.9592	0.2512	23.7229×
SU128	0.0583	10.592	1.2557	8.4351×
AVG	-	-	-	48.6536×

To demonstrate the scalability of our approach, we compared the performance of three methods under different waterlines for MR and PR, as illustrated in Fig. 6. It can be seen that our method can obtain better performance improvement under almost all waterline parameters, while ELASM may not be able to find better solutions in some cases due to limited search space.

C. Compilation time comparison

Table II shows the compilation time of Fig 5, where the speedup means the speedup of HEILP compared to ELASM. Without expensive undirected exploration, this work achieves 48.65× speedup over ELASM on average. Furthermore, ELASM is unable to sense the size of the solution space, necessitating manual parameter tuning. Otherwise, they might end up conducting unnecessary searches in small spaces, as observed in MLP. Conversely, HEILP effectively models the solution space, facilitating an adaptive solution time with the problem's scale. Additionally, due to the bootstrapping, the depth within a single program region is limited [12], making the expansion of solving time acceptable.

V. Conclusion

This paper proposes an automatic scale management method HELIP for fully homomorphic encryption. HELIP formulates the scale management problem as a constrained-optimization problem that can be solved in a single pass. It creates a comprehensive optimization space to find a better solution than previous methods. The proposed method is integrated into the MLIR framework to generate high-quality solutions. With the comprehensive problem modeling, HEILP achieves 1.72× speedup over the previous heuristic approach.

References

[1] C. Gentry, "Fully homomorphic encryption using ideal lattices," in *Proceedings of the forty-first annual ACM symposium on Theory of computing*, 2009, pp. 169–178.

[2] J. H. Cheon, K. Han, A. Kim, M. Kim, and Y. Song, "A full rns variant of approximate homomorphic encryption," in *Selected Areas in Cryptography–SAC 2018: 25th International Conference, Calgary, AB, Canada, August 15–17, 2018, Revised Selected Papers 25*. Springer, 2019, pp. 347–368.

[3] "Microsoft SEAL (release 4.0)," https://github.com/Microsoft/SEAL, Mar. 2022, microsoft Research, Redmond, WA.

[4] A. Al Badawi, J. Bates, F. Bergamaschi, D. B. Cousins, S. Erabelli, N. Genise, S. Halevi, H. Hunt, A. Kim, Y. Lee *et al.*, "Openfhe: Open-source fully homomorphic encryption library," in *proceedings of the 10th workshop on encrypted computing & applied homomorphic cryptography*, 2022, pp. 53–63.

[5] R. Dathathri, O. Saarikivi, H. Chen, K. Laine, K. Lauter, S. Maleki, M. Musuvathi, and T. 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*, 2019, pp. 142–156.

[6] R. Dathathri, B. Kostova, O. Saarikivi, W. Dai, K. Laine, and M. 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*, 2020, pp. 546–561.

[7] Y. Lee, S. Heo, S. Cheon, S. Jeong, C. Kim, E. Kim, D. Lee, and H. Kim, "Hecate: Performance-aware scale optimization for homomorphic encryption compiler," in *2022 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 2022, pp. 193–204.

[8] Y. Lee, S. Cheon, D. Kim, D. Lee, and H. Kim, "Elasm:error-latency-aware scale management for fully homomorphic encryption," in *32nd USENIX Security Symposium (USENIX Security 23)*, 2023, pp. 4697–4714.

[9] ——, "Performance-aware scale analysis with reserve for homomorphic encryption," in *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, 2024, pp. 302–317.

[10] S. Cheon, Y. Lee, D. Kim, J. M. Lee, S. Jung, T. Kim, D. Lee, and H. Kim, "Dacapo: Automatic bootstrapping management for efficient fully homomorphic encryption," in *33rd USENIX Security Symposium (USENIX Security 24)*, 2024.

[11] C. Lattner, M. Amini, U. Bondhugula, A. Cohen, A. Davis, J. Pienaar, R. Riddle, T. Shpeisman, N. Vasilache, and O. Zinenko, "Mlir: Scaling compiler infrastructure for domain specific computation," in *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 2021, pp. 2–14.

[12] N. Samardzic, A. Feldmann, A. Krastev, S. Devadas, R. Dreslinski, C. Peikert, and D. Sanchez, "F1: A fast and programmable accelerator for fully homomorphic encryption," in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, 2021, pp. 238–252.

[13] S. Carpov, P. Dubrulle, and R. Sirdey, "Armadillo: a compilation chain for privacy preserving applications," in *Proceedings of the 3rd International Workshop on Security in Cloud Computing*, 2015, pp. 13–19.

[14] A. Viand, P. Jattke, M. Haller, and A. Hithnawi, "Heco: Fully homomorphic encryption compiler," in *32nd USENIX Security Symposium (USENIX Security 23)*, 2023, pp. 4715–4732.

[15] R. Malik, K. Sheth, and M. Kulkarni, "Coyote: A compiler for vectorizing encrypted arithmetic circuits," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, 2023, pp. 118–133.

[16] A. Krastev, N. Samardzic, S. Langowski, S. Devadas, and D. Sanchez, "A tensor compiler with automatic data packing for simple and efficient fully homomorphic encryption," *Proceedings of the ACM on Programming Languages*, vol. 8, no. PLDI, pp. 126–150, 2024.

[17] D. Lee, W. Lee, H. Oh, and K. Yi, "Optimizing homomorphic evaluation circuits by program synthesis and term rewriting," in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2020, pp. 503–518.

[18] A. Singh, S. Das, A. Chakraborty, R. Sadhukhan, A. Chatterjee, and D. Mukhopadhyay, "Fheda: Efficient circuit synthesis with reduced bootstrapping for torus fhe," *Cryptology ePrint Archive*, 2023.

[19] R. Recto and A. C. Myers, "A compiler from array programs to vectorized homomorphic encryption," *arXiv preprint arXiv:2311.06142*, 2023.

[20] N. Samardzic, A. Feldmann, A. Krastev, N. Manohar, N. Genise, S. Devadas, K. Eldefrawy, C. Peikert, and D. Sanchez, "Craterlake: a hardware accelerator for efficient unbounded computation on encrypted data," in *Proceedings of the 49th Annual International Symposium on Computer Architecture*, 2022, pp. 173–187.

[21] Gurobi Optimization, LLC, "Gurobi Optimizer Reference Manual," 2024. [Online]. Available: https://www.gurobi.com