RESEARCH-ARTICLE

ReSBM: Region-based Scale and Minimal-Level Bootstrapping Management for FHE via Min-Cut

YAN LIU, Ant group, Hangzhou, Zhejiang, China

JIANXIN LAI, Ant group, Hangzhou, Zhejiang, China

LONG LI, Ant group, Hangzhou, Zhejiang, China

TIANXIANG SUI, Ant group, Hangzhou, Zhejiang, China

LINJIE XIAO, Ant group, Hangzhou, Zhejiang, China

PENG YUAN, Ant group, Hangzhou, Zhejiang, China

View all

Open Access Support provided by:

Ant group

Tsinghua University

PDF Download
3669940.3707276.pdf
25 December 2025
Total Citations: 3
Total Downloads: 2518

Published: 30 March 2025

Citation in BibTeX format

ASPLOS '25: 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems
March 30 - April 3, 2025
Rotterdam, Netherlands

Conference Sponsors:
SIGARCH
SIGPLAN
SIGOPS

ASPLOS '25: Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1 (March 2025)
https://doi.org/10.1145/3669940.3707276
ISBN: 9798400706981

ReSBM: Region-based Scale and Minimal-Level Bootstrapping Management for FHE via Min-Cut

Yan Liu
Ant Group
Shanghai, China

Jianxin Lai*
Ant Group
Shanghai, China

Long Li
Ant Group
Shanghai, China

Tianxiang Sui
Ant Group
Shanghai, China

Linjie Xiao
Ant Group
Shenzhen, China

Peng Yuan
Ant Group
Beijing, China

Xiaojing Zhang
Ant Group
Shanghai, China

Qing Zhu
Ant Group
Shanghai, China

Wenguang Chen*
Tsinghua University, Ant Group
Beijing, China

Jingling Xue
UNSW, Ant Group
Sydney, Australia

Abstract

The RNS-CKKS scheme in Fully Homomorphic Encryption (FHE) supports crucial features for privacy-preserving machine learning, such as fixed-point arithmetic and SIMD-style vectorization. Yet, managing the escalation of ciphertext scales from homomorphic multiplications, which risks capacity overflow, along with bootstrapping, presents significant challenges. These complexities are exacerbated by the need to efficiently handle scale and bootstrapping at compile time while ensuring rapid encrypted inference.

In this paper, we present ReSBM, a novel compiler technique that simultaneously optimizes scale and bootstrapping for encrypted inference under RNS-CKKS. By partitioning a program's data flow graph (DFG) into regions with a uniform multiplicative depth of one, ReSBM ensures that placements of Scale Management Operations (SMOs) and bootstraps affect only the latency of a region, not the scales and levels of its live-out ciphertexts. Our region-based approach tackles the NP-hard challenge of optimal bootstrapping placement with hierarchical strategies: (1) optimal intra-region SMO and bootstrapping placement using min-cut, (2) bootstrapping-guided rescaling region identification across a sequence of regions, culminating in tentative bootstrapping at two terminal regions, and (3) minimal-level bootstrap placement across the DFG, elevating ciphertexts only to the necessary minimal level. Validation across a variety of complex models on CPUs shows that ReSBM not only compiles these models more rapidly than a leading method but also boosts encrypted inference efficiency by an average of 12.1% when compared to another leading method. Consequently, ReSBM substantially improves the practical deployment of large models for encrypted inference, surpassing existing methods in terms of both compilation speed and inference performance.

CCS Concepts: • **Software and its engineering** → **Compilers**; • **Security and privacy** → **Cryptography**;

Keywords: FHE, CKKS, RNS-CKKS, Machine Learning, Scale Management, Bootstrapping

ACM Reference Format:
Yan Liu, Jianxin Lai*, Long Li, Tianxiang Sui, Linjie Xiao, Peng Yuan, Xiaojing Zhang, Qing Zhu, Wenguang Chen*, and Jingling Xue. 2025. ReSBM: Region-based Scale and Minimal-Level Bootstrapping Management for FHE via Min-Cut. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1 (ASPLOS '25), March 30–April 3, 2025, Rotterdam, Netherlands.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3669940.3707276

1 Introduction

Fully homomorphic encryption (FHE) [7] facilitates computations on encrypted data, addressing privacy concerns when outsourcing to untrusted servers and enabling secure data exchanges in regulated sectors like healthcare and finance. Various FHE schemes, including [8, 11, 15, 16, 20, 27, 42], have been proposed, with the RNS-CKKS scheme [16] standing out for encrypted inference due to its support for fixed-point arithmetic and SIMD-style vectorization. Consequently, it is broadly supported by FHE libraries such as HEAAN [37], HElib [36], OpenFHE [1], SEAL [34], and ACElib [6].

*Corresponding author.

ASPLOS '25, March 30–April 3, 2025, Rotterdam, Netherlands

ACM ISBN 979-8-4007-0698-1/25/03
https://doi.org/10.1145/3669940.3707276

In the RNS-CKKS scheme, each ciphertext is characterized by a level and a scale. The level indicates the maximum depth of multiplications, both direct and indirect, that the ciphertext can undergo. The scale facilitates the conversion of encrypted complex (real) values into integers, which is crucial for operations like encrypted machine learning inference that predominantly involve real numbers. For example, the real number 1.6 can be represented as 16 at a scale of 10. When ciphertexts undergo operations like multiplication, their scales increase, which can potentially lead to capacity overflow (as detailed in Section 2). For instance, squaring the value of 16 at a scale of 10 results in 256 at a scale of 100.

To facilitate encrypted inference for machine learning models, developers must implement these models using FHE operations provided by established FHE libraries [1, 6, 34, 36, 37]. Manually undertaking this task is challenging, as it requires developers to meticulously balance security, correctness, and performance, a process that is both time-consuming and error-prone, often extending over weeks or months even for experts [24]. To mitigate this burden, a number of FHE compilers [9, 13, 24, 28, 33, 35, 39, 40] have been introduced. These compilers, possibly assisted by DSLs, automatically transform machine learning models into corresponding FHE programs, effectively balancing security, correctness, and performance to expedite the inference process.

In RNS-CKKS, ensuring correctness involves managing ciphertext scales and levels; both additions and multiplications require ciphertexts at matching levels, and additions also need identical scales. Scale Management Operations (SMOs), like rescale (reducing scale and level) and modswitch (lowering only the level) [16], are critical for reducing ciphertext scale and level, thus boosting FHE operation efficiency. Yet, when a ciphertext's level falls to zero, halting further multiplications, bootstrapping [5, 35]—the most expensive FHE operation—is needed to raise its level, which also raises the latency of subsequent operations. Effective scale and bootstrapping management by FHE compilers is vital for ensuring rapid and accurate encrypted inference.

Problem Statement. We aim to develop a compiler approach that optimizes the placement of SMOs and bootstraps for machine learning models using RNS-CKKS on CPUs. This approach must meet all constraints related to scale and bootstrapping management while efficiently compiling models and maximizing encrypted inference efficiency.

Challenges. Given the NP-hard nature of optimal bootstrapping placement [31], efficiently crafting scale and bootstrapping management plans for rapid encrypted inference poses several challenges. Firstly, the interdependence of rescaling and bootstrapping complicates their management; scale reductions often require bootstrapping to maintain ciphertext viability, while bootstrapping raises levels, increasing the latency of subsequent operations. Secondly, a vast search space complicates planning. Thirdly, as bootstrapping is RNS-CKKS's most resource-intensive operation, minimizing its use is crucial to prevent excessive overheads. Lastly, effective scale management must account for bootstrapping outcomes to efficiently reduce FHE operation latency.

Prior Work. Since most FHE compilers [9, 26, 40] lack bootstrapping support, existing scale management approaches are categorized into local or global strategies. Local strategies like EVA's waterline rescaling [9] and PARS [40] focus on inserting SMOs based on immediate operand and result scales, efficiently optimizing compile times but often overlooking overall performance, leading to sub-optimal outcomes. Conversely, global strategies such as HECATE [40] and ELASM [26] target optimal plans by assessing the cumulative impact of SMOs through hill-climbing-based space exploration, but they experience low compile-time efficiency, with LeNet5 taking over 300 seconds to compile [26, 40]. While these methods can automate scale management for smaller FHE programs, their effectiveness is further limited without bootstrapping support [35], crucial for larger applications where bootstrapping significantly impacts operation latencies.

Effective bootstrapping management requires seamless integration with scale management, especially given the NP-hard challenge of optimal bootstrapping placement [31]. Fhelipe [24] employs dynamic programming to devise a bootstrapping plan based on the maximum multiplicative depth, adopting EVA's waterline rescaling method [9]. However, the coordination between scale management and bootstrapping placement is critical for accurate operation latency assessments. DaCapo [35], which conducts liveness analysis for bootstrapping placement, utilizes PARS [40] for scale management. This approach can result in sub-optimal outcomes due to less effective rescaling strategies. Additionally, while exploration-driven methods like HECATE [40] and ELASM [26] offer improved solutions over PARS, they also significantly increase compile times, as noted in [35]. This extension in compile time limits their practicality for smaller models, leading DaCapo to avoid using these methods for bootstrapping management.

This Work. To address the challenges in scale and bootstrapping management and to surmount the limitations of existing solutions, we strategically place SMOs and bootstraps simultaneously within a program. We capitalize on a critical feature of RNS-CKKS [16]: multiplications are the only arithmetic operations that increase ciphertext scales. Each multiplication triggers an increase, while adjustments to scale and level via SMOs or bootstrapping remain unchanged until the next multiplication, as detailed in Table 1. Consequently, we partition the data flow graph (DFG) into regions, each maintaining a uniform multiplicative depth. These regions form the primary units for our analysis and optimization. As illustrated in Figure 1 and explained in Section 3, this approach ensures that placements of SMOs and bootstraps within a region only affect its latency—the time required to execute the operations in the region—without altering the scales and levels of its live-out ciphertexts. Therefore, effective management of rescaling and bootstrapping involves strategically determining the insertion points for these operations within the regions.

We introduce ReSBM, a region-based compiler approach that efficiently coordinates SMO and bootstrapping placements at compile-time to optimize encrypted inference for machine learning models. We address the NP-hard challenge of optimal bootstrapping placement [31] with a hierarchical divide-and-conquer strategy using four key algorithms:

- **Optimal Intra-Region SMO and Bootstrapping Placement** using min-cut for finding insertion points for SMOs (via SMOPLC) and bootstraps (via BTSPLC).
- **Bootstrapping-Guided Rescaling Region Identification for SMO Placement** identifies rescaling regions for SMO placement in a sequence of regions, ending with tentative bootstrapping at both ends. ScaleMgr strategically rescales earlier to accelerate ciphertext scale reduction, minimize levels consumed, and reduce overall sequence latency.
- **Minimal-Level Bootstrapping Placement** across the DFG is enabled by BTSMgr, leveraging ScaleMgr, SMOPLC, and BTSPLC. BTSMgr identifies potential bootstrapping points in terminal regions of each region sequence and selects regions for minimal-level bootstrapping via dynamic programming. For each sequence, it uses ScaleMgr to select rescaling regions for SMO placement, and SMOPLC and BTSPLC to determine the optimal placements of SMOs and bootstraps. Unlike DaCapo [35] and Fhelipe [24], which elevate ciphertexts to the maximum level, our approach only raises ciphertexts to the minimal necessary level. This strategy boosts efficiency and optimizes operations prior to subsequent bootstrapping points.

Contributions. This paper makes four main contributions:

- A region-based compiler approach, ReSBM, that efficiently coordinates SMO and bootstrapping placements, substantially reducing compile times and enhancing encrypted inference efficiency while maintaining accuracy.
- The first min-cut-based scale management solution.
- The first minimal-level bootstrapping solution.
- Comprehensive performance evaluations of ReSBM confirm its effectiveness in meeting design objectives. Validation with a diverse range of complex machine learning models on CPUs demonstrates that ReSBM compiles these models substantially faster than a state-of-the-art method, DaCapo [35], by 4250.2×. Additionally, it improves encrypted inference efficiency by an average of 12.1% over another leading method, Fhelipe [24], with the maximum bootstrapping level set at 16. ReSBM significantly advances the practical deployment of large models for encrypted inference, outperforming current methods in both compilation efficiency and inference performance.

ReSBM is now open-sourced within the ANT-ACE compiler framework [6].

2 Background

We begin with a broad review of representative FHE schemes (Section 2.1), followed by a detailed examination of RNS-CKKS [16] adopted in this work (Section 2.2).

2.1 FHE Schemes

Since Gentry introduced the first FHE scheme [7], which refreshes ciphertexts through bootstrapping by homomorphically executing the decryption function, numerous schemes based on the Learning with Errors (LWE) [30] and Ring Learning with Errors (RLWE) [38] problems have been developed. Notable among these are the BFV [11], BGV [42], and CKKS [15] schemes. Despite these advancements, bootstrapping remains the most computationally intensive operation in FHE. Initially, performing a bootstrapping operation required several hours, but with modern optimizations, this time has been reduced to the order of seconds [12, 21].

CKKS [15] is based on RLWE, while BGV [42] has two variants: one based on LWE and another on RLWE. Both CKKS and the RLWE version of BGV represent ciphertexts as polynomials, with each polynomial's coefficients reduced modulo a user-determined coefficient modulus. However, BGV and CKKS encrypt different types of values. BGV encrypts integers, whereas the CKKS scheme supports fixed-point arithmetic by encrypting complex numbers scaled with a user-defined scale, making it particularly desirable for supporting machine learning applications. Despite encrypting different types of values, both BGV and CKKS support homomorphic addition, multiplication, and Single Instruction Multiple Data (SIMD)-style batching [32, 41]. This batching enhances throughput by packing multiple plaintexts into a single ciphertext, allowing FHE operations to be evaluated element-wise on multiple plaintexts with approximately the same efficiency as on a single plaintext [42].

In FHE, ciphertexts inherently contain noise, which increases slightly during homomorphic additions and significantly during homomorphic multiplications. In the BGV scheme [42], this noise growth is managed using modulus switching, a technique that reduces the coefficient modulus to decrease the noise level. In the CKKS scheme, besides noise growth, the ciphertext scale also increases exponentially during homomorphic multiplications, potentially leading to scale overflow. To address this, a rescaling operation is performed to reduce the ciphertext scale after a certain number of multiplications. This rescaling is analogous to modulus switching in BGV, serving a similar role by reducing both the ciphertext noise and the coefficient modulus.

2.2 RNS-CKKS

RNS-CKKS [16], a widely used variant of the CKKS scheme [15], supports fixed-point arithmetic on encrypted complex numbers, facilitating encrypted machine learning inference. Below, we review the fundamentals of RNS-CKKS to understand the essential aspects of scale and bootstrapping management necessary for its proper operation.

For a power-of-two integer N, we define $R = \mathbb{Z}[X]/(X^N + 1)$ as a ring of integer polynomials where $X^N + 1$ is the *polynomial modulus*. The residue ring, denoted by R_Q, is defined as R/QR, which is the ring R with all polynomial coefficients taken modulo an integer Q (the *coefficient modulus*).

Scale. In CKKS [15], a vector v of $N/2$ complex numbers is encoded into a plaintext $pt \in R$ and encrypted into a ciphertext $ct \in R_Q^2$—a pair of polynomials with cryptographic noise, where Q denotes the coefficient modulus of ct. The values in v are scaled by q to integers during encoding, enabling fixed-point arithmetic. This scaling mimics operations on the original floating-point numbers while controlling precision and noise. Importantly, each ciphertext's associated scale increases with every multiplication. For instance, multiplying two ciphertexts at the same level with scales s_0 and s_1 results in a ciphertext with a scale of $s_0 \times s_1$, as shown in Table 1.

Relinearization. A freshly encrypted ciphertext has two polynomials; multiplying two such ciphertexts results in three. Generally, multiplying ciphertexts with m and n polynomials yields $m + n - 1$ polynomials. Relinearization is used to reduce this number back to two for efficiency [9, 14].

Level. Each ciphertext in CKKS, comprising n polynomials, is tied to a level $level$ and represented as $ct = (ct_0, ..., ct_{n-1}) \in R^n_{Q_{level}}$. Here, $Q_{level} = \prod_{j=0}^{level} q_j$, where $\prod$ denotes the product operation, $q_1, ..., q_{level}$ are coprime integers approximating the scale factor q, and q_0 is a large prime ensuring output precision. The RNS-CKKS variant [16] uses RNS to decompose each polynomial ct_i into $(level + 1)$ smaller ones, $ct_i^0, ..., ct_i^{level}$, with each ct_i^j aligned to its specific modulus q_j. This structure, $ct = (ct_0^{(j)}, ..., ct_{n-1}^{(j)})_{0 \leqslant j \leqslant level} \in \prod_{j=0}^{level} R^n_{q_j}$, enhances computational efficiency while largely preserving precision. Arithmetic operations are performed in smaller rings R_{q_j}, simplifying the overall computational complexity.

Arithmetic and Rotation Operations. RNS-CKKS supports basic arithmetic operations, including AddCP (ciphertext-plaintext addition), AddCC (ciphertext-ciphertext addition), MulCP (ciphertext-plaintext multiplication), and MulCC (ciphertext-ciphertext multiplication). As described in Section 2.1, these operations are all element-wise. For instance, multiplying two ciphertexts encrypting the vectors $[a_0, a_1, a_2, a_3]$ and $[b_0, b_1, b_2, b_3]$ yields a ciphertext encrypting $[a_0 \times b_0,\ a_1 \times b_1,\ a_2 \times b_2,\ a_3 \times b_3]$. The Rotate operation shifts elements within the plaintext of a ciphertext by a specified amount. For example, rotating a ciphertext encrypting $[a_0, a_1, a_2, a_3]$ by one position yields a ciphertext encrypting $[a_1, a_2, a_3, a_0]$. As noted in Table 1, the scales and levels of the resulting ciphertexts typically match those of their operands, except for MulCP and MulCC, which increase the scale.

Table 1. Scales and levels of FHE operation results. s_i and l_i represent the scale and level of the i-th operand ($i \in \{0, 1\}$), s_{res} and l_{res} denote the scale and level of the resultant ciphertext, and l_{bts} indicates the target level for bootstrapping.

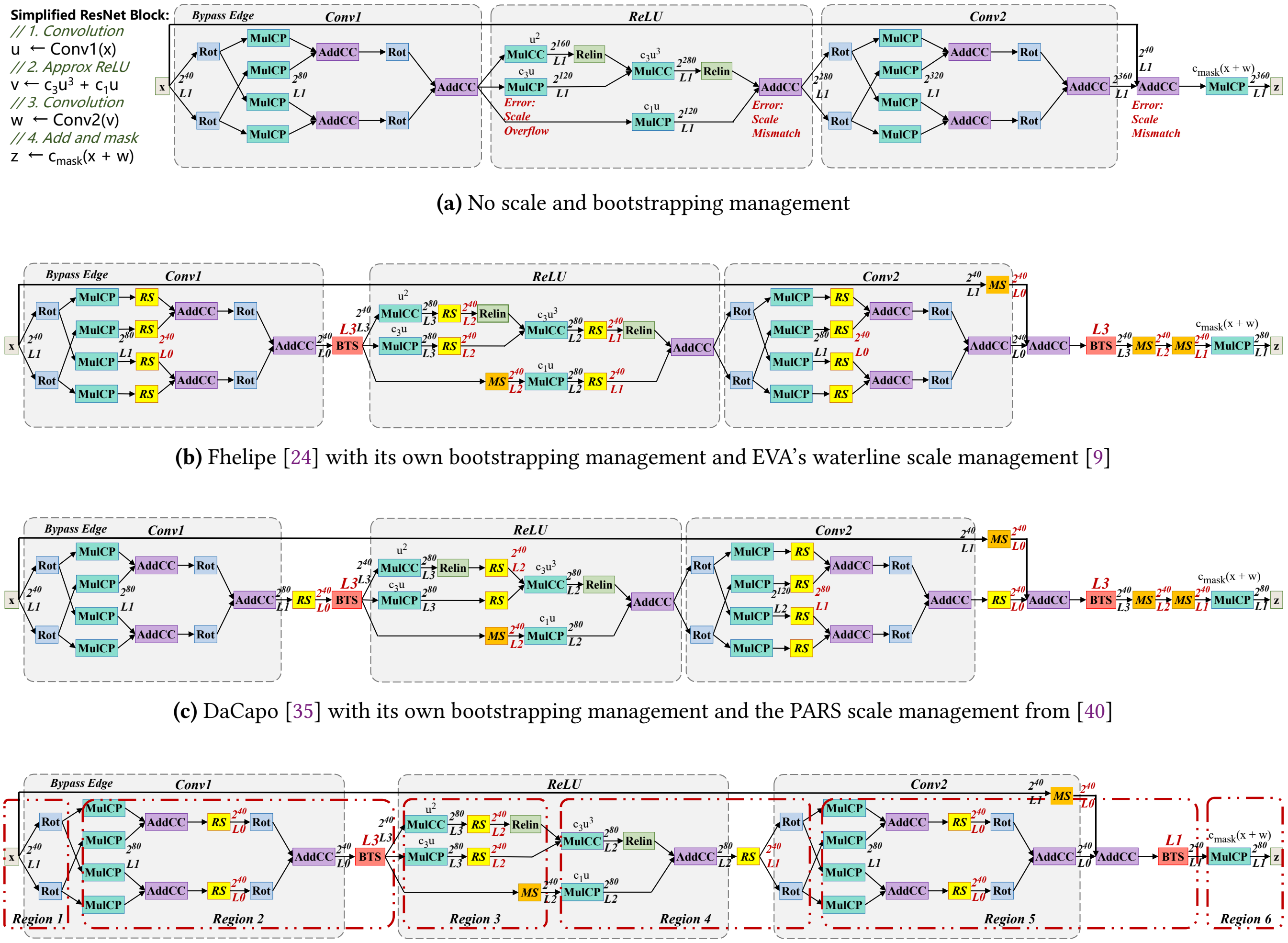
Operation	Constraints	s_{res}	l_{res}
AddCP ct, pt	$s_0 == s_1$ && $l_0 == l_1$	s_0	l_0
AddCC ct_0, ct_1	$s_0 == s_1$ && $l_0 == l_1$	s_0	l_0
MulCP ct, pt	$l_0 == l_1$	$s_0 \times s_1$	l_0
MulCC ct_0, ct_1	$l_0 == l_1$	$s_0 \times s_1$	l_0
Rotate ct, k	–	s_0	l_0
Rescale ct	–	s_0/q	$l_0 - 1$
Modswitch ct	–	s_0	$l_0 - 1$
Bootstrap ct, l_{bts}	–	q	l_{bts}

Operation Constraints. For a ciphertext ct, we write $ct.level$ and $ct.scale$ to indicate its level and scale, respectively. Several constraints govern FHE operations. Firstly, the ciphertext level must be non-negative to prevent modulus depletion: $ct.level \geqslant 0$. Secondly, MulCP and MulCC operations increase scales, which must remain below the coefficient modulus Q_{level} to avoid overflow: $ct.level \geqslant \lceil \log(ct.scale)/\log(q) \rceil - 1$. Finally, RNS-CKKS requires managing scales and levels for binary operations: both additions and multiplications necessitate matching levels, and additions require identical scales as detailed in Table 1 (the first four rows).

To ensure compliance, several scale and bootstrapping management operations are detailed in Table 1 (the last three rows): (1) **Rescale:** Reduces the ciphertext scale by factor q and lowers the level by one, suitable for ciphertexts at least $q \times q_w$, where q_w denotes the waterline [9], a predefined minimum scale. This mitigates RNS-CKKS operation noise [9]. (2) **Modswitch:** Lowers the ciphertext level by one without altering its scale, simply switching from Q_l to Q_{l-1}. (3) **Bootstrapping:** Elevates a ciphertext to a specified level l_{bts} and sets its scale to q. Latency depends only on the target level l_{bts}, not the initial level [5, 35]. This characteristic is utilized in the development of ReSBM to minimize bootstrapping levels and thereby enhance bootstrapping efficiency.

3 Motivation

We motivate our ReSBM approach through a simplified ResNet block, as depicted in Figure 1a, highlighting key differences from the only two existing methods [24, 35] that tackle the same problem addressed by us. In FHE compilers [9, 40], static data-flow graphs (DFGs) or circuits serve as intermediate representations, excluding control-flow elements such as conditionals and loops. In FHE, data-dependent operations like branching cannot be performed on ciphertexts, thus requiring loop counts to be predetermined at compile-time.



```
Simplified ResNet Block:
// 1. Convolution
u ← Conv1(x)
// 2. Approx ReLU
v ← c₃u³ + c₁u
// 3. Convolution
w ← Conv2(v)
// 4. Add and mask
z ← c_mask(x + w)
```

(a) No scale and bootstrapping management

(b) Fhelipe [24] with its own bootstrapping management and EVA's waterline scale management [9]

(c) DaCapo [35] with its own bootstrapping management and the PARS scale management from [40]

(d) ReSBM: this paper's region-based scale and bootstrapping management

Figure 1. Illustrating ReSBM and its distinctions from two existing methods with a simplified ResNet block. The input ciphertext x starts with a scale of 2^{40} at level 1, with both scale factor and waterline set at $q = 2^{40}$. Abbreviations used for operations in Table 1 include RS (Rescale), BTS (Bootstrap), MS (Modswitch), Rot (Rotation), and Relin (Relinearization). Li indicates the level of a ciphertext at i. FHE operations are color-coded to highlight different types.

In Figure 1a, the ResNet block includes an activation function (ReLU) and two convolutions (Conv1 and Conv2). The initial ciphertext x starts with a scale of $s = 2^{40}$ at level $l = 1$. We assume a scale factor and waterline both set at $q = q_w = 2^{40}$, $q_0 = q$, and a maximum bootstrapping level of $l_{\max} = 3$. In the illustrations from Figure 1a to Figure 1d, we annotate the level of a ciphertext at i with L_i. Additionally, FHE operations are color-coded to highlight different types. In RNS-CKKS [16], non-polynomial functions like ReLU are approximated with polynomials [25]; here, we use the cubic polynomial $c_3u^3 + c_1u$ for simplicity. Consistent with EVA [9], weights and biases are encoded at the waterline.

Without scale and bootstrapping management, the program in Figure 1a fails to execute. MulCC and MulCP operations escalate result scales, leading to discrepancies and mismatches that cause runtime errors when adding ciphertexts. Moreover, in operations like ReLU and Conv2 at $L1$, many ciphertext scales surpass the coefficient modulus $Q_1 \approx 2^{80}$, the maximum allowed scale capacity, causing scale overflow.

Effective scale management reduces latency for SMOs and lowers ciphertext levels to speed up FHE operations. Table 2 lists latencies (measured on a CPU@2.70GHz using ACElib's FHE APIs [6]) and worst-case time complexities of key FHE operations. To underscore compile-time optimization opportunities, MulCC and Relinearization are listed separately in Table 2, following EVA's approach [9], although DaCapo

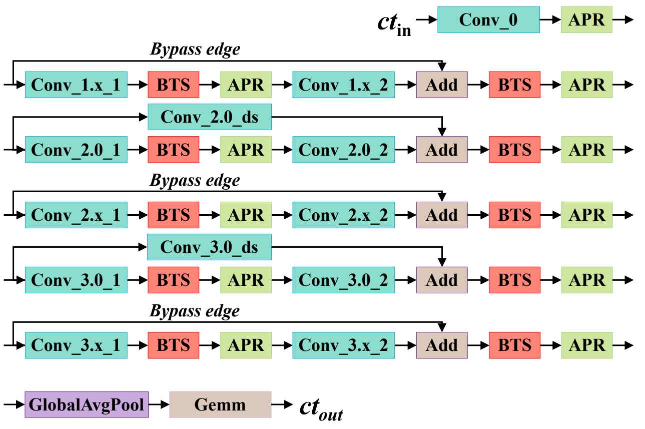


Figure 2. Architecture of ResNet, depicting each ResNet block on a separate line. Abbreviations: APR for approximate ReLU, BTS for bootstrapping, and Conv for convolution.

[35] combines them into a single ciphertext-ciphertext multiplication operation. Modswitch, involving the dropping of one modulo, is the least costly operation. In contrast, bootstrapping is the most time-consuming, followed by rotation and relinearization. Although rescaling is less time-intensive, its latency exceeds those of operations like AddCP, AddCC, MulCP, and MulCC. Thus, strategically placing SMOs is essential to minimize their latency and enhance the efficiency of subsequent, more costly operations such as bootstrapping, relinearization, and rotation.

In machine learning models, as depicted in Figure 2, the output from one block typically serves as the input for the next. Bootstrapping, which elevates ciphertext levels for subsequent multiplications, is essential but costly, far exceeding the expenses of rotation and relinearization (Table 2). Since FHE operations are more efficient at lower ciphertext levels, effective bootstrapping management is crucial to minimize both its own latency and that of subsequent operations [5, 35]. Ideally, bootstrapping should elevate a ciphertext from level 0 ($L0$) only to the minimal necessary level, optimizing overall latency. Bootstrapping to the maximum level ($l_{\max}$) and reducing it with nearly zero-cost modswitch operations is not effective, as higher bootstrapping levels significantly increase costs (Table 2). Therefore, integrating bootstrapping with scale management is key to accurately determining necessary ciphertext levels for each operation.

In Figure 1b to Figure 1d, we compare the scale and bootstrapping management techniques of Fhelipe [24] and DaCapo [35]—the only two compiler methods currently supporting bootstrapping in the literature—with our ReSBM approach. Both Fhelipe and DaCapo elevate ciphertexts to the maximum allowed level $l_{\max}$ (with $l_{\max} = 3$ in this simple example), which may increase latency. Fhelipe employs dynamic programming for bootstrapping decisions alongside EVA's waterline method [9] for scale management, whereas DaCapo uses liveness analysis for bootstrapping and PARS [40] for scale management, with both EVA and PARS serving as local rescaling strategies. Conversely, global strategies such as HECATE [40] and ELASM [26], which rely on hill-climbing space exploration, are impractically time-consuming for large models, taking about 300 seconds even for simple models like LeNet5 [35]. Our ReSBM approach stands out by integrating scale and bootstrapping management to only elevate ciphertexts to the minimal necessary level, marking it as the first approach to do so.

Let us review the solutions implemented by Fhelipe and DaCapo as depicted in Figures 1b and 1c. Both employ local rescaling strategies—EVA by Fhelipe and PARS by DaCapo—centered on immediate operand and result scales. This focus can sometimes cause broader performance impacts and inefficiencies to be overlooked. EVA mandates that the scale of a multiplication result, s_{res}, must be less than $q \times q_w$, while PARS permits $s_{res} \leq q \times q_w^2$ and requires each operand's ciphertext scale, s_{opr}, to stay within $q \times q_w$. As a result, EVA tends to rescale more aggressively, enabling two rotations in Conv2 at $L0$. Conversely, the more conservative rescaling by PARS leads to these rotations occurring at $L1$, unnecessarily increasing latency by 12.9%. Although both methods trigger rescaling at specific threshold levels, EVA's frequent rescaling can sometimes degrade performance. Adopted by Fhelipe, EVA introduces an average of 21.6× more rescaling operations than ReSBM, as detailed in Section 5.

For bootstrapping, both Fhelipe and DaCapo elevate the output ciphertext from the last AddCC in each convolution layer to the maximum level ($l_{\max} = 3$). While the first bootstrap in Conv1 is necessary, the second in Conv2 is excessive, necessitating two modswitch operations to lower it to $L1$ for the final multiplication at the end.

Figure 1d illustrates the solution found by our compiler approach, ReSBM. This region-based strategy coordinates scale and bootstrapping management hierarchically, quickly compiling large machine learning models into efficient FHE programs and boosting encrypted inference efficiency.

In ReSBM, the DFG of a program is divided into a sequence of data-dependent regions, each with a consistent multiplicative depth of one, as illustrated in Figure 1d. Multiplications (MulCC and MulCP), which are the only operations that increase ciphertext scales, are positioned at the beginning of each region. Thus, the number of these regions is equivalent to the program's maximum multiplicative depth plus one. As shown in Figure 1d, ReSBM divides the DFG into six regions, corresponding to the example's multiplicative depth of five.

Once the DFG is segmented into regions, SMOs and bootstraps are exclusively inserted within these regions. Insertions within a region modify ciphertext scales and levels at these points, but such changes persist unchanged throughout the region, impacting only the region's latency, not the region's live-out scales and levels. Treating regions as fundamental units for optimization simplifies the management

Table 2. Time complexities and latencies of RNS-CKKS operations on a CPU@2.70 GHz obtained using ACElib's FHE APIs [6] at varying levels l, with $N = 2^{16}$ (ms).

Operation	**Time Complexity**	$l = 0$	$l = 2$	$l = 4$	$l = 6$	$l = 8$	$l = 10$	$l = 12$	$l = 14$	$l = 16$
AddCP	$O(N \times l)$	0.138	0.575	0.886	1.268	1.714	1.931	2.295	2.807	3.066
AddCC	$O(N \times l)$	0.164	0.548	0.936	1.344	1.690	2.089	2.561	3.089	3.574
MulCP	$O(N \times l)$	–	1.175	1.993	2.746	3.553	4.354	5.175	5.902	6.837
MulCC	$O(N \times l)$	–	2.509	4.237	6.021	7.750	9.280	11.129	13.053	15.638
Rotate	$O(N \times \log(N) \times l^2)$	58.422	77.521	93.799	111.901	130.940	150.321	241.560	243.323	290.575
Relinearization	$O(N \times \log(N) \times l^2)$	–	76.947	93.617	111.819	130.493	149.586	215.768	242.031	262.308
Rescale	$O(N \times \log(N) \times l^2)$	–	9.085	15.107	21.333	27.535	33.792	40.068	46.372	52.744
Bootstrap	–	–	21005	23738	26229	30413	34556	37844	41582	44719
Modswitch	$O(1)$	–								

of scale and bootstrapping to strategic insertions within each respective region. This region-based method utilizes a three-tier divide-and-conquer strategy, streamlining the implementation of our minimal-level bootstrapping approach and the development of efficient, high-performance rescaling and bootstrapping plans for large machine learning models:

- **Optimal Intra-Region SMO and Bootstrapping Placement** utilizes min-cut for inserting SMOs (via SMOPLC) and bootstraps (via BTSPLC) within individual regions. In this motivating example, once Regions 2 and 5 are designated for bootstrapping, BTSPLC finds two bootstrap points (in Conv1 and Conv2), just like Fhelipe and DaCapo. However, SMOPLC's SMO placements differ. For instance, in Region 2, optimal placement of SMOs by SMOPLC markedly reduces total latency (excluding the BTS operation)—from 142.616 ms with Fhelipe and 143.860 ms with DaCapo, to 131.832 ms under ReSBM.
- **Bootstrapping-Guided Rescaling Region Identification for SMO Placement** identifies rescaling regions for SMO placement in a sequence, concluding with tentative bootstrapping at both ends, managed by ScaleMgr. To reduce latency and minimize level consumption, ScaleMgr applies early rescaling to expedite scale reduction. As shown in Figure 1d, rescaling is necessary in every region for the sequence from Region 2 to Region 5, but only in Region 5 for the sequence from Region 5 to Region 6.
- **Minimal-Level Bootstrapping Placement** across the DFG uses BTSMgr to identify regions for minimal-level bootstrapping via dynamic programming. This process examines each sequence of regions, particularly the terminal ones, to optimize SMO and bootstrapping placements and minimize sequence latency. ScaleMgr pinpoints rescaling regions within each sequence, while SMOPLC and BTSPLC fine-tune the placement of SMOs and bootstraps, respectively. In contrast to Fhelipe [24] and DaCapo [35], which elevate ciphertexts to maximum levels, our method keeps levels minimal, boosting bootstrapping efficiency and subsequent operations. Consider Figure 1d: Analysis from Region 2 to Region 5 identifies these as critical bootstrapping points, prompting ScaleMgr to recommend rescaling the entire sequence and setting Region 2's minimal bootstrapping level at $l_{\max} = 3$. For the sequence from Region 5 to Region 6, only level 1 bootstrapping is necessary in Region 5. Both Fhelipe and DaCapo select $l_{\max} = 3$ (Figures 1b and 1c), necessitating two modswitch operations to reduce it to level 1. Although modswitch operations are $O(1)$, bootstrapping the ciphertext to $l_{\max} = 3$ incurs significantly higher overhead than direct bootstrapping to level 1, resulting in inefficiencies (Table 2).

Algorithm 1: The ReSBM compiler approach.

```
input  : $G$ as the DFG of an FHE program
output : $G'$ with the SMOs and bootstraps inserted
1 $R \leftarrow$ BuildRegionedDFG($G$)
2 $P \leftarrow$ BTSMgr ($R$)
3 $G' \leftarrow$ InsertScaleAndBootstrappingPlan($P$)
4 return $G'$
```

4 The ReSBM Approach

ReSBM, as outlined in Algorithm 1, processes the DFG of an FHE program, G, transforming it into an optimized DFG, G', incorporating SMOs and bootstraps to satisfy all scale and bootstrapping requirements (Section 2). Initially, G is segmented into a region-based DFG, R (Section 4.1). In line 2, our bootstrapping manager, BTSMgr, formulates a rescaling and minimal-level bootstrapping plan for R (Section 4.2). This step includes evaluating each sequence of regions within R, planning tentative bootstrapping at both ends, and collaborating with ScaleMgr (Section 4.3) to identify appropriate rescaling regions for SMO placement. This effort also establishes the minimal bootstrapping level at the start relative to the next bootstrapping point at the end region. BTSMgr utilizes SMOPLC to optimize SMO placements in these regions and BTSPLC for bootstraps at the start region (Section 4.4). Finally, in line 3, ReSBM implements this comprehensive plan in G, culminating in the enhanced DFG, G'.

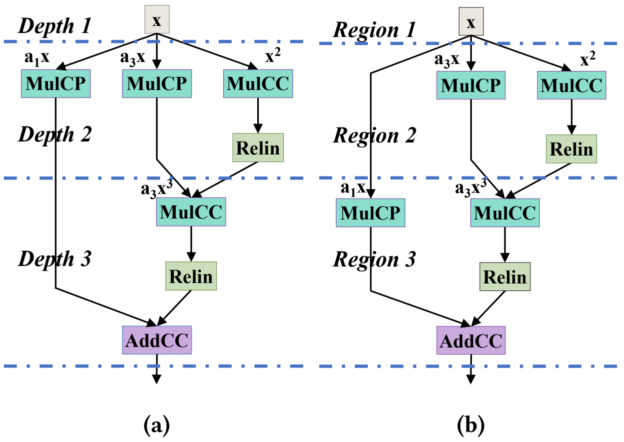


Figure 3. Two region partitions for $a_3x^3 + a_1x$.

4.1 DFG Partitioning

ReSBM partitions a program's DFG into regions, ensuring each region contains MulCC and MulCP multiplications that establish a multiplicative depth of exactly one. Consequently, the number of regions equals the program's maximum multiplicative depth plus one. Moreover, the multiplications within a region are always positioned at its beginning.

In FHE, loops have compile-time known bounds. To streamline analysis and optimization, loops are unrolled only when necessary. A loop is not unrolled if the maximum multiplicative depth in its unrolled version is one; otherwise, it is unrolled. When calculating the latency for an unrolled loop, the latency is adjusted by the number of iterations.

Figure 3 shows two different region partitions for a simple FHE program, $a_3x^3 + a_1x$, with a preference for the one in Figure 3b over Figure 3a. While multiplications at the program's maximum multiplicative depth are uniquely assigned to distinct regions, other nodes not on this critical path have flexibility in their placement. For instance, MulCP for a_1x, calculated at depth 2 but used at depth 3, is placed in the depth 2 region in Figure 3a but in the depth 3 region in Figure 3b. In Figure 3a, a modswitch must be introduced after the MulCP operation to match levels for the final AddCC, but this order is reversed in Figure 3b. This reversal in Figure 3b, which performs MulCP at a lower level, enhances efficiency and performance (as revealed earlier in Table 2).

At line 1 of Algorithm 1, ReSBM applies BuildRegionedDFG to construct a region-based graph R uniquely from a DFG, G, in three clear steps. Initially, the first region, R_0, contains the input ciphertexts from G. Next, the critical path D, representing G's maximum multiplicative depth, is identified, and each multiplication at depth i is placed in region R_i. Finally, two traversals on G allocate other nodes not on D to their appropriate regions. During the forward pass, each node is assigned to the region with the smallest number among its predecessors' regions. In the backward pass, each multiplication node is shifted to the region with the highest number among those containing its successors, ensuring that each region begins with multiplications, as illustrated in Figure 1d. This method of forming regions results in the selection of Figure 3b and the exclusion of Figure 3a.

By design, R is always a sequence of (data-dependent) regions corresponding to G's maximum multiplicative depth (plus one), as illustrated in Figure 1d. We write $R.first$ and $R.last$ for R's first and last regions, respectively.

Algorithm 2: BTSMgr: Bootstrapping manager.

```
   input  : R as a region-based graph of a DFG
   output : Plan as a rescaling and bootstrapping plan
1  $minLAT[R.first] \leftarrow 0$
2  foreach $r \in (R.first, R.last])$ do
3      $minLAT[r] \leftarrow$ DOUBLE_MAX
4  foreach $src \in [R.first, R.last)$ do
5      foreach $dst \in (src, R.last]$ do
6          $RescalingRegions \leftarrow$ ScaleMgr $(src, dst)$
7          $l_{bts} \leftarrow |RescalingRegions \setminus \{src\}|$
8          if $l_{bts} > l_{max}$ then
9              break
10         $\mathcal{L} \leftarrow 0$
11         foreach $r \in RescalingRegions \setminus \{dst\}$ do
12             $RescalingPlan \leftarrow$ SMOPLC$(r)$
13             if $r ==$ "$src$" then
14                 $BTSPlan \leftarrow$ BTSPLC$(src, l_{bts})$
15             $\mathcal{L} \leftarrow \mathcal{L} + r$'s latency after $RescalingPlan$,
                 and $BTSPlan$ (if any), have been applied to $r$
16         $newLAT \leftarrow minLAT[src] + \mathcal{L}$
17         if $newLAT < minLAT[dst]$ then
18             $minLAT[dst] \leftarrow newLAT$
19             $Plan[dst] \leftarrow \{src, BTSPlan, RescalingPlan\}$
20 return Plan
```

4.2 Bootstrapping Management

BTSMgr, as outlined in Algorithm 2, processes a region-based DFG R to devise a rescaling and bootstrapping plan through dynamic programming. In this algorithm, $l_{\max}$ indicates the highest level to which a level 0 ciphertext can be bootstrapped, setting roughly the limit on the distance between two consecutive bootstrapping points in R.

After initialization at lines 1-3, the core functionality of BTSMgr unfolds within two loops at lines 4-5, processing a sequence of regions $[src, dst]$ with src and dst as tentative bootstrapping points. ScaleMgr identifies rescaling regions in $RescalingRegions$ (line 6), while l_{bts} at line 7 denotes levels consumed in $(src, dst]$, excluding src where rescaling precedes bootstrapping (Section 4.4). Lines 8-9 dismiss infeasible bootstrapping pairs. Lines 10-15 utilize SMOPLC and BTSPLC to optimize SMO and bootstrapping placements in *RescalingRegions*, with *dst* excluded as it transitions to a start region in future sequences. At line 15, the latency for region *r* is calculated by summing the latencies of all FHE operations within that region. Lines 16-19 update the plan per dynamic programming standards. Note that bootstrapping is required for the first region in *R* when its input ciphertexts start at level 0, which is managed similarly.

For our motivating example, applying BTSMgr results in the rescaling and bootstrapping plan depicted in Figure 1d. Specifically, there are bootstraps in Region 2 and Region 5. The minimum latency $minLAT[6] = 42509$ is the lowest. If the second bootstrap is moved from Region 5 to Region 4, the overall latency $minLAT[6] = 42754$ increases.

4.3 Scale Management

ScaleMgr, detailed in Algorithm 3, processes a sequence of regions $[src, dst]$ with *src* and *dst* as tentative bootstrapping points, and records rescaling regions in *RescalingRegions*. A ciphertext is eligible for rescaling if its scale is at least $q \times q_w$. As scales increase exponentially with more multiplications, early rescaling reduces latency. Each rescaling lowers the level by one, emphasizing the importance of targeting regions that significantly decrease the output scale of *dst* (its live-out ciphertexts). Between two rescaling options that offer the same reduction in output scale, the earlier one is preferred to enable more operations at a lower level.

We sequentially identify rescaling regions by scanning $[src, dst]$ using two loops at lines 3 and 5. Starting with $region = src$ (line 2), we search for the best rescaling region within $[region, dst]$ during the current round as detailed in lines 5-10. The search halts (lines 7-8) when *SMORegion* is determined to be the best choice (line 11), driven by the accumulation of ciphertext scales from multiplications along the sequence. If further rescaling is needed, the process resumes from $[SMORegion + 1, dst]$ (lines 13-14).

As illustrated in Figure 1d, rescaling is required in every region in [Region 2, Region 5], but only in Region 5 for the sequence [Region 5, Region 6].

Algorithm 3: ScaleMgr: scale management.

```
   input  : [$src, dst$] with bootstrapping at both ends
   output : $RescalingRegions$ as a set of regions
1  $RescalingRegions \leftarrow \emptyset$
2  $region \leftarrow src$
3  while $region \leqslant dst$ do
4      $bestScale \leftarrow INT\_MAX$
5      for $CanRegion \in [region, dst]$ do
6          $scale \leftarrow$ the scale of $dst$'s live-out ciphertexts
             after rescaling has been applied in $CanRegion$
7          if $scale \geqslant bestScale$ then
8              break
9          $SMORegion \leftarrow CanRegion$
10         $bestScale \leftarrow scale$
11     Add $SMORegion$ to $RescalingRegions$
12     $region \leftarrow SMORegion + 1$
13     if $bestScale \leqslant q$ then
14         break
15 return $RescalingRegions$
```

4.4 SMO and Bootstrap Placement via Min-Cut

ReSBM applies SMOPLC, detailed in Algorithm 4, and BTSPLC, outlined in Algorithm 5, to find the optimal placements of SMOs and bootstraps within a region, respectively.

Algorithm 4: SMOPLC: optimal SMO placement.

```
   input  : $G_r = (N_r, E_r)$ as a directed graph for region $r$
   output : $minCut$ as the minimum cut in $G_r$
            (post-transformation)
1  $src \leftarrow$ set of source (MulCC or MulCP) nodes in $G_r$
2  $snk \leftarrow$ set of sink nodes in $G_r$
3  $l \leftarrow$ level of the MulCC/MulCP operations in $G_r$
4  $topoSort \leftarrow$ topological sort of $G_r$
5  foreach $n \in topoSort$ do
6      $\mathcal{L}_n^{SMO} \leftarrow n.freq \times \mathcal{L}[RS][l])$
7      if $n \in src$ then
8          $\mathcal{L}_n^{inc} \leftarrow 0$
9      else
10         $\mathcal{L}_n^{inc} \leftarrow \mathcal{L}[n][l] - \mathcal{L}[n][l-1] + \sum_{m \in pred(n)} \mathcal{L}_m^{inc}$
11     foreach $m \in succ(n)$ do
12         $w_{(n,m)} \leftarrow (\mathcal{L}_n^{SMO} + \mathcal{L}_n^{inc}) / |succ(n)|$
13 $G'_r \leftarrow G_r$ constructed with the transformation above
14 $minCut \leftarrow$ min-cut($src, snk, G'_r$) [29]
15 return $minCut$
```

Given a region represented by an unweighted directed graph, G_r, we use SMOPLC to find a minimum cut for optimal SMO insertion points. This process involves two steps: initially, transforming G_r into a multi-source, multi-sink weighted graph, G'_r (lines 1-13), and subsequently applying a min-cut algorithm [29] to G'_r (line 14). Rescaling at these points minimizes the region's latency. $\mathcal{L}[n][l]$ indicates the latency of operation *n* at level *l*, with RS denoting a rescaling operation. *n.freq* represents the statically known loop count for operation *n* if it is in the loop, and 1 otherwise. *pred* (*succ*) maps a node to its predecessors (successors).

When performing min-cut in G_r, the backedge of a loop is disregarded since any loop containing multiplications has a consistent multiplicative depth of one (Section 4.1). This

Algorithm 5: BTSPLC: optimal bootstrapping placement.

```
   input  : $G_r = (N_r, E_r)$ as a directed graph for region $r$
            $l_{bts}$ as the target bootstrapping level
   output : $minCut$ as the minimum cut in $G_r$
            (post-transformation)
1  $G'_r \leftarrow G_r$ with only level-0 nodes, excluding rescaling
     operations inserted in line 12 of Algorithm 2 and
     their outgoing edges
2  $src \leftarrow$ set of source nodes in $G'_r$
3  $snk \leftarrow$ set of sink nodes in $G'_r$
4  $revTopoSort \leftarrow$ reverse topological sort of $G'_r$
5  foreach $n \in revTopoSort$ do
6      $\mathcal{L}_n^{BTS} \leftarrow n.freq \times \mathcal{L}[BTS][l_{bts}])$
7      if $n \in snk$ then
8          $\mathcal{L}_n^{inc} \leftarrow 0$
9      else
10         $\mathcal{L}_n^{inc} \leftarrow \mathcal{L}[n][l_{bts}] - \mathcal{L}[n][0] + \sum_{m \in succ(n)} \mathcal{L}_m^{inc}$
11     foreach $m \in pred(n)$ do
12         $w_{(m,n)} \leftarrow (\mathcal{L}_n^{BTS} + \mathcal{L}_n^{inc})/|pred(n)|)$
13 $G''_r \leftarrow G'_r$ with the weights created above
14 $minCut \leftarrow$ min-cut$(src, snk, G''_r)$ [29]
15 return $minCut$
```

streamlines the process, achieving the same minimum as if all loops were fully unrolled, thereby enhancing efficiency.

Initially, *src* and *snk* represent the source and sink nodes in G_r, respectively (lines 1-2). Line 3 sets l as the initial level for all multiplications in the region, which is decreased by one via rescaling. From lines 4-10, each node n is analyzed in topological order. The edge weight $w(m, n)$ includes: (1) the latency of a rescaling operation (RS) to be inserted after n (line 6), and (2) the cumulative latency increase of n's direct and indirect predecessors (line 10) compared to when RS is immediately applied after the source nodes (line 8). Lines 11-12 address the necessity of the divisor *succ*, as the RS will be actually added on a new edge (n, m'), creating a new node m' and new edges $\{(m', m) \mid m \in succ(n)\}$. The process concludes with a minimum cut on G'_r (lines 13-15).

We employ BTSPLC from Algorithm 5 to optimize bootstrap placements in region G_r. Like SMOPLC in Algorithm 4, BTSPLC operates in reverse. As outlined in Algorithm 2, when BTSPLC is invoked (line 14), any necessary rescaling has already modified the ciphertext levels in G_r (line 12), preparing for bootstrap insertions. Our ReSBM approach only inserts bootstraps for level-0 ciphertexts post-rescaling (lines 1-2). Line 6 details $\mathcal{L}[n][l_{bts}]$, the latency for a bootstrapping operation n at level l_{bts}, and line 10 measures the latency increase, $\mathcal{L}[n][l_{bts}] - \mathcal{L}[n][0]$, if a bootstrap is placed before n rather than at the region's end (line 8).

Figure 4 demonstrates the minimum cut identified by SMOPLC for SMO placement in Region 2 (Figure 1d), allowing

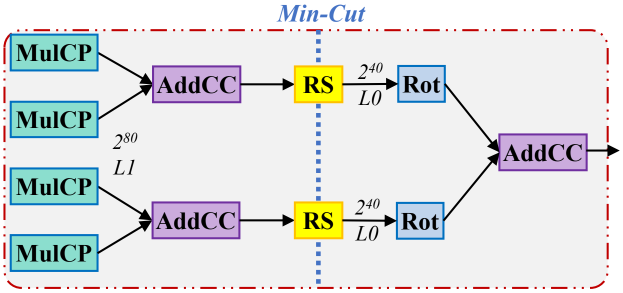


Figure 4. Optimal SMO placement for Region 2 in Figure 1d.

ReSBM to achieve the lowest latency at 131.832 ms. In contrast, Fhelipe [24] and DaCapo [35] (Figures 1b and 1c) employ non-minimum cuts, resulting in higher latencies of 142.616 ms and 143.860 ms, respectively. For bootstrapping, all three methods identified the same insertion point, consistently using the maximum bootstrap level $l_{\max} = 3$.

Moving to Region 5, as depicted in Figure 1d, we observe that for scale management, both Fhelipe and DaCapo yield sub-optimal solutions (Figures 1b and 1c). In terms of bootstrapping, ReSBM employs the minimal level of 1, whereas Fhelipe and DaCapo both utilize the maximum level $l_{\max} = 3$.

Theorem 1. *Given a region, SMOPLC in Algorithm 4 produces a minimum cut that minimizes its total execution time.*

Proof. This arises from how edge weights are set in Algorithm 4. Specifically, the weight for each edge (line 12) within the region represents the combined cost of executing a rescaling operation on that edge and the additional cost incurred relative to performing rescaling at the region's outset (lines 8 and 10). Consequently, the minimum cut identified by SMOPLC minimizes the total weight across this cut, effectively minimizing the overall execution time of the region. □

Theorem 2. *Given a region, BTSPLC in Algorithm 5, when restricted to the region's level-0 nodes (line 1), produces a minimum cut that minimizes its total execution time.*

Proof. Follows from the reasoning in Theorem 1's proof. □

4.5 Time Complexities

Both SMOPLC (Algorithm 4) and BTSPLC (Algorithm 5), primarily governed by a min-cut algorithm, exhibit a worst-case complexity of $O(n^3)$ when applied to a region consisting of n nodes, with n roughly equivalent to the number of edges.

In ScaleMgr (Algorithm 3), the worst case involves selecting all L regions in a sequence $[src, dst]$ for rescaling. Calculating the scale of live-out ciphertexts at *dst* in each region requires $O(n)$ computations, with n nodes per region (line 6). Therefore, the total time complexity is $O(n \times L^2)$.

For BTSMgr (Algorithm 2), the most time-consuming steps are ScaleMgr (line 6), SMOPLC (line 12), and BTSPLC (line 14). Considering a DFG G with R regions and n nodes per region, the time complexity for scale management (line 12) is O($R \times l_{\max} \times T_{sm}$), where $T_{sm} = O(n \times l_{\max}^2)$ covers one call to ScaleMgr. This analysis assumes each region is evaluated $O(l_{\max})$ times (lines 8-9). By caching min-cut results, the complexity for SMOPLC and BTSPLC reduces to O($R \times l_{\max} \times n^3$), where $O(n^3)$ reflects the cost for one min-cut application. Here, $l_{\max}$ indicates multiple processings of the same region due to varying live-in ciphertext levels. Therefore, the total time complexity of BTSMgr (i.e., our ReSBM approach) is O($R \times l_{\max} \times n \times (l_{\max}^2 + n^2)$). Since neither $l_{\max}$ nor n scales with the size of a machine learning model, the overall complexity is linearly related to R, which equals the model's maximum multiplicative depth plus one.

4.6 Limitations of ReSBM

ReSBM partitions a program's DFG into regions, optimizing the placement of SMOs and bootstraps within each region. Although optimal within individual regions, ReSBM is not optimal across the entire DFG due to the NP-hard nature of optimal global bootstrapping placement [31]. As a result, ReSBM may produce sub-optimal plans in scenarios that require cross-region optimization. Nonetheless, these plans can be enhanced through additional compiler optimizations.

Figure 5 shows a sub-optimal plan by ReSBM that can be improved with compiler optimizations, aiming to minimize latency for executing all SMOs and bootstraps. In this example, the ciphertext polynomials $y = a_3x^3$ and $z = a_4((a_1x)^2 + y^4)$ are computed, with x starting at level 0 with a scale of 2^{40} and z as the output, assuming a maximum bootstrapping level of $l_{\max} = 3$ and $q = q_w = q_0 = 2^{40}$. Figure 5a illustrates ReSBM's sub-optimal scale and bootstrapping management plan. Here, y is computed by multiplying a_3x by x^2, and z from $a_4((a_1x)^2 + (y^2)^2)$, requiring three bootstraps: two at level 3 and one at level 2. This plan can be improved by merging the two bootstraps for x, while retaining the BTS at L2 but shifting it to L3 using CSE (Common Subexpression Elimination). Alternatively, applying constant folding to a_1 and a_4, followed by CSE on x^2, allows ReSBM to directly generate an optimal plan, as shown in Figure 5b. Since creating optimal plans is NP-hard [31], integrating ReSBM with compiler optimizations expands optimization opportunities, thereby enhancing its effectiveness.

5 Evaluation

We demonstrate that ReSBM, through its hierarchical design, serves as a practical compiler solution for efficiently managing rescaling and bootstrapping in large machine learning models on CPUs, thus advancing the state of the art. Our evaluation explores the following three research questions:

- **RQ1:** Can ReSBM efficiently compile large models?
- **RQ2:** Does ReSBM improve encrypted inference efficiency?
- **RQ3:** Does ReSBM preserve accuracy in encrypted inference comparable to unencrypted inference?

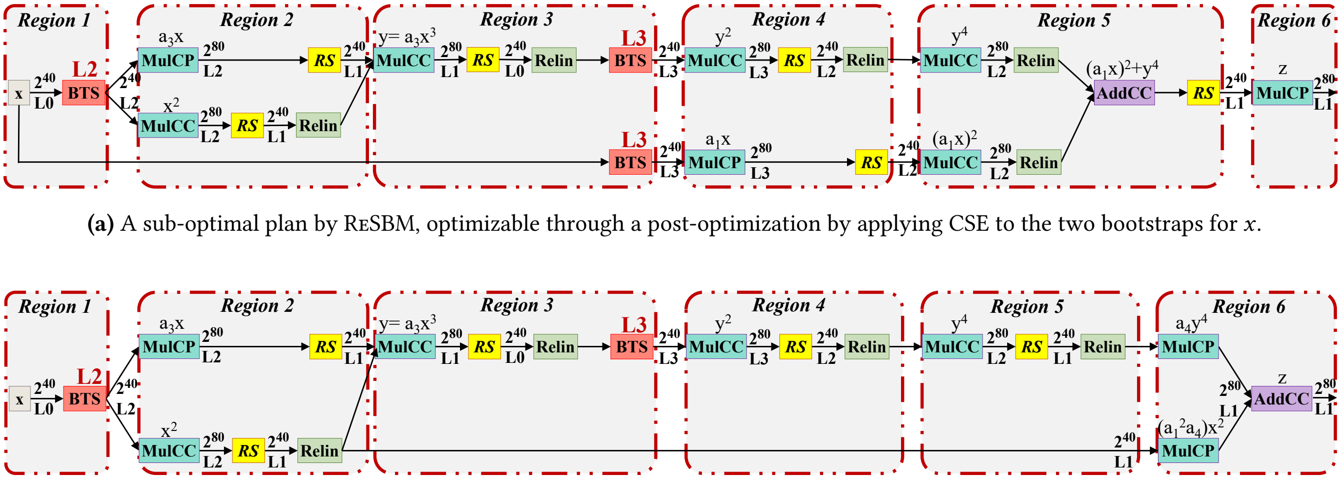
- **Methodology.** Currently, only two compiler approaches for scale and bootstrapping management exist: DaCapo [35] and Fhelipe [24], both of which elevate ciphertexts to their maximum allowed level $l_{\max}$. During the review of this paper, DaCapo had only partially open-sourced its compiler, with its runtime system that supports bootstrapping remaining proprietary. Conversely, Fhelipe was fully open-sourced during the same period but it operates as a DSL-driven FHE compiler. Fhelipe manages bootstrapping based on a DFG's multiplicative depth using dynamic programming and employs EVA [9] for rescaling—an approach that we have integrated into our framework. Our analysis primarily focuses on comparing ReSBM with Fhelipe in terms of encrypted inference efficiency (RQ2). Attempting to replicate DaCapo's method, which uses PARS [40] to determine bootstrapping insertion points based on live-out ciphertexts, was challenging in our framework. Therefore, our comparison with DaCapo is limited to compile times (RQ1) based on their published results but does not extend to encrypted inference efficiency (RQ2), since DaCapo's implementations are on GPUs, while ReSBM is implemented on CPUs.

Additionally, we developed three ReSBM variants for substitution analysis to assess its management strategies: (1) ReSBM$_{\max}$, which raises bootstrapping levels to $l_{\max}$ akin to Fhelipe [24] and DaCapo [35]; (2) ReSBM$_{\text{eva}}$, replacing ReSBM's scale management (ScaleMgr in line 6 and SMOPLC in line 12 of Algorithm 2) with EVA's [9]; and (3) ReSBM$_{\text{pm}}$, merging ReSBM$_{\max}$'s approach with PARS [40] for scale management instead of EVA. ReSBM$_{\max}$ further includes an modswitch optimization to lower levels in excessively bootstrapped ciphertexts, similar to EVA's method (Figure 1b).

We have implemented ReSBM and its three variants as well as Fhelipe in the ANT-ACE FHE compiler [28]. ANT-ACE employs a five-level intermediate representation (IR), including Neural Network (NN) IR, CKKS IR, and Polynomial IR. CKKS IR specifically encapsulates arithmetic operations, rotations, SMOs, and bootstrapping relevant to CKKS, facilitating domain-specific analysis and optimizations. Implementations of ReSBM, its variants, and Fhelipe utilize this CKKS IR.

- **Machine Learning Models.** We utilize a range of deep-learning models, including the ResNet series (20/44/110) [22], AlexNet [3], VGG16 [23], SqueezeNet [19], and MobileNet [18], which are more complex than those typically used in FHE compiler research [9, 24, 26, 40]. As RNS-CKKS [16] does not support non-arithmetic activations like ReLU, we adopt a minimax polynomial approximation [10] with a multiplicative depth of 11, calculating coefficients per [25].

Currently, FHE programs are about 10, 000× slower than their unencrypted counterparts on CPUs [24]. For example, a single encrypted inference for ResNet110 can take about 2 hours. We have dedicated considerable effort to thoroughly evaluating ReSBM and identifying areas for improvement.



(a) A sub-optimal plan by ReSBM, optimizable through a post-optimization by applying CSE to the two bootstraps for x.

(b) An optimal plan by ReSBM, achieved with a pre-optimization.

Figure 5. Sub-optimality of ReSBM in computing ciphertext polynomials $y = a_3x^3$ and $z = a_4((a_1x)^2+y^4)$. The input ciphertext x starts at level 0 with a scale of 2^{40}, and the output is z with $l_{\max} = 3$ and $q = q_w = q_0 = 2^{40}$. (a) shows a sub-optimal plan that can be improved by removing the redundant BTS for x at L3 and shifting the BTS at L2 for x to L3 using CSE. Alternatively, constant folding of a_1 and a_4, followed by CSE on x^2, produces another optimal plan in (b).

- **Experimental Setup.** Experiments were done using Docker (25.0.1) on a Linux server with an Intel Xeon Platinum 8369B CPU @2.70GHz and 512 GB of memory. Models were compiled into their FHE versions using a scale factor of $q = 2^{56}$, a waterline of $q_w = q$, and output precision $q_0 = 2^{60}$. These FHE programs were converted to C using ACElib's FHE APIs [6, 28] and compiled with GCC under "-O2" (10.2.1). The polynomial degree was $N = 2^{16}$ for ResNet20/44/110 and $N = 2^{17}$ for AlexNet, MobileNet, SqueezeNet, and VGG16 to manage larger intermediate results. ACElib's bootstrapping operation has a multiplicative depth of 15, and for security [2], it sets $l_{\max} = 16$, nearly the highest level, matching [35].

We utilized the CIFAR-10 dataset for each model, encoding each input image into a single ciphertext, similar to approaches used in Dacapo [35] and Fhelipe [24]. While our experimental results validate ReSBM against these two existing techniques in this setting, we anticipate that these findings will apply to larger images requiring multiple ciphertexts. Importantly, methods for effectively partitioning larger images for this purpose are not yet established, highlighting an area of ongoing research.

- **RQ1: Compile Times.** Table 3 compares ReSBM's compile times against Fhelipe and DaCapo. DaCapo's compile times, sourced from its publication [35], were recorded on an Intel(R) Core(TM) i7-12700. ReSBM compiles all models in under one second, with compile times peaking at 0.773 seconds for ResNet110. It significantly outperforms DaCapo, being faster by an average of 4250.2×, underscoring the costliness of DaCapo's liveness-analysis-based bootstrapping approach. While Fhelipe compiles models faster than ReSBM, it sacrifices the effectiveness of scale and bootstrapping management solutions, which we discuss below.

- **RQ2: Encrypted Inference Efficency.** As illustrated in Figure 6 (with $l_{\max} = 16$), ReSBM excels over its three variants—ReSBM$_{\text{eva}}$, ReSBM$_{\max}$, and ReSBM$_{\text{pm}}$ —in encrypted inference efficiency across all models. ReSBM consistently outperforms ReSBM$_{\text{eva}}$ by an average of 1.05×, highlighting its superior scale management. It also surpasses ReSBM$_{\max}$ by 1.07×, emphasizing the effectiveness of its minimal-level bootstrapping strategy, and beats ReSBM$_{\text{pm}}$ by 1.21×, underscoring the benefits of its integrated scale and minimal-level bootstrapping management. ReSBM$_{\text{pm}}$'s performance is the lowest among the three variants, with PARS's lazy rescaling contributing additionally to its underperformance by often resulting in operations being performed at higher levels than in ReSBM or EVA, thereby increasing latency (Figure 1c).

Compared to Fhelipe [24], a leading method as depicted in Figure 1b, ReSBM achieves superior performance with an average improvement of 12.1% by excelling in rescaling and bootstrapping efficiencies. For scale management, Table 4 reveals that Fhelipe, utilizing EVA's waterline rescaling, initiates significantly more rescaling operations, ranging from 5.51× (ResNet20) to 69.12× (VGG16), with an average of 21.6×. This demonstrates ReSBM's more effective placement of rescaling operations, enhancing its overall performance.

Let us consider bootstrapping placement. ReSBM, the first compiler to bootstrap ciphertexts only to necessary levels, enhances both bootstrapping and subsequent operations. Table 5 compares bootstrapping decisions between ReSBM and Fhelipe. While both insert the same number of bootstraps per

Table 3. Compile times (in seconds) for ReSBM, Fhelipe [24], and DaCapo [35] on machine learning models. Fhelipe's compile times are obtained in our own implementation. DaCapo's times are from [35], measured on an Intel(R) Core(TM) i7-12700.

Model	ResNet20	ResNet44	ResNet110	AlexNet	VGG16	SqueezeNet	MobileNet
ReSBM	0.128	0.290	0.773	0.050	0.094	0.147	0.185
Fhelipe	0.088	0.200	0.540	0.035	0.067	0.095	0.127
DaCapo	15.8	79.4	–	1042.3	230.1	89.1	222.8

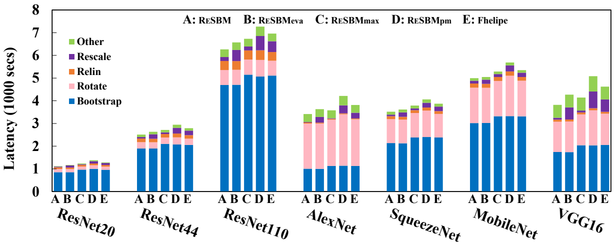


Figure 6. Efficiency of encrypted inference under ReSBM, ReSBM$_{\text{eva}}$, ReSBM$_{\text{max}}$, and Fhelipe ($l_{\text{max}} = 16$).

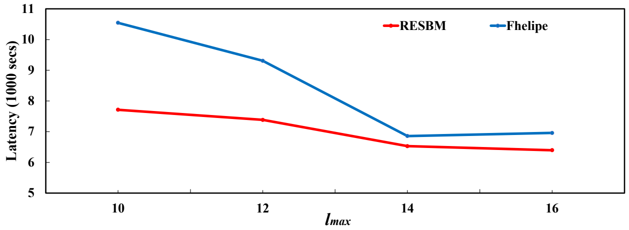


Figure 7. Comparing ReSBM and Fhelipe on encrypted inference latency for ResNet110 at varying l_{max} values.

model, Fhelipe always elevates to $l_{\text{max}} = 16$, whereas ReSBM customizes levels to operational needs as outlined in Algorithm 2 (line 7). According to Table 2, reducing a level from $l_{\text{max}} = 16$ to $l_{\text{max}} = 14$ saves 3137 ms, equivalent to avoiding about 240 costly ciphertext-ciphertext multiplications. Figure 6 demonstrates that ReSBM's improvements range from 7.0% (MobileNet) to 21.0% (VGG16), averaging 12.1%. This showcases how ReSBM's minimal-level management, enhanced by SMOPLC and BTSPLC, effectively optimizes placement and boosts encrypted inference efficiency.

Finally, we assessed how varying l_{max} values affect the efficiency of encrypted inference with ReSBM versus Fhelipe, using ResNet110, the most complex model tested. Trends for the other six models are similar. Lower security levels allow a higher l_{max}, while higher security levels limit it [2]. Reducing l_{max} from 16 to 14, 12, and 10 leads both ReSBM and Fhelipe to progressively insert more bootstraps—110, 112, 174, 217, respectively. While Fhelipe consistently bootstraps to these maximum levels, ReSBM opts for minimal levels, enhancing efficiency (as shown in Table 5 for $l_{\text{max}} = 16$). Figure 7 indicates that decreasing l_{max} prolongs inference times for both methods, yet ReSBM maintains performance advantages over Fhelipe, with improvements of 8.8%, 5.0%, 26.0%, and 36.6%. These performance gains primarily result from ReSBM's successful reduction in bootstrapping levels: 105/110 (between levels 13-15) as indicated in Table 5, 105/112 (at level 14) at $l_{\text{max}} = 14$, 174/174 (between levels 4-12) at $l_{\text{max}} = 12$, and 217/217 (between levels 5-9) at $l_{\text{max}} = 10$.

- **RQ3: Inference Accuracy.** Table 6 shows that ReSBM maintains the accuracy of all models during encrypted inference, closely matching their original pre-trained accuracy. Due to FHE's high computational demands (2 hours for ResNet110), we tested 1,000 images per model. Encrypted inference incurs minor accuracy losses, peaking at 1.7% for ResNet44, with an average loss of 0.3%. This is mainly due to RNS-CKKS's approximate nature and the polynomial approximation of ReLU [25]. For context, expert-crafted and heavily hand-tuned implementations for ResNet20/44/110 [10] show accuracy drops ranging from 0.1% to 0.6%. DaCapo [35] exhibits accuracy variations ranging from -0.1% to 0.2%, whereas Fhelipe [24] reports variations between -1% and 1% for ResNet20. These differences are statistically insignificant between encrypted and unencrypted inference.

6 Related work

Since Gentry introduced the first FHE scheme based on ideal lattices [7], several other schemes such as BGV [42], BFV [11], GSW [8], FHEW [27], TFHE [20], and CKKS [15] have been developed, leveraging the LWE [30] and RLWE [38] problems. Notably, BGV, BFV, and CKKS support SIMD-style batching, packing multiple values into a single ciphertext to enhance throughput. RNS-CKKS [16], a variant of CKKS known for supporting fixed-point arithmetic, is now widely used for encrypted inference in machine learning.

In RNS-CKKS, cryptographic noise in ciphertexts escalates with each operation, particularly multiplications. Managing scale and bootstrapping is crucial to mitigate this noise and ensure encrypted data integrity, significantly impacting the efficiency of homomorphic computations. Essential scale management techniques include EVA [9], HECATE [40], and ELASM [26]. For bootstrapping, methods like DaCapo [35] and Fhelipe [24] optimize placement, while techniques [4, 5, 17, 21] focus on enhancing bootstrapping efficiency.

In scale management, EVA [9] adjusts scales exceeding $q \times q_w$, whereas PARS [40] employs a looser constraint of

Table 4. Number of executed rescaling operation under ReSBM and Fhelipe on machine learning models at $l_{\max} = 16$.

	ResNet20	**ResNet44**	**ResNet110**	**AlexNet**	**VGG16**	**SqueezeNet**	**MobileNet**
ReSBM	2627	6063	15512	610	1026	1458	2035
Fhelipe	14495	33767	86765	28775	70917	14868	16337

Table 5. Bootstrapping levels l selected by ReSBM and Fhelipe on machine learning models at $l_{\max} = 16$.

Model	**ReSBM**								**Fhelipe**
	$l = 16$	$l = 15$	$l = 14$	$l = 13$	$l = 12$	$l = 9$	$l = 6$	$l = 5$	$l = l_{\max} = 16$
ResNet20	1	6	3	6	2	0	0	2	20
ResNet44	1	18	3	18	2	0	0	2	44
ResNet110	1	51	3	51	2	0	0	2	110
AlexNet	0	1	3	2	0	1	0	2	9
VGG16	0	0	7	5	0	0	0	5	17
SqueezeNet	1	0	7	7	2	0	2	0	19
MobileNet	1	0	18	7	0	0	0	4	30

Table 6. Comparing inference accuracy between unencrypted and encrypted models compiled with ReSBM.

Model	**Unencrypted**	**Encrypted**	**Accurray Loss**
ResNet20	90.6%	90.8%	-0.2%
ResNet44	92.5%	90.9%	1.7%
ResNet110	93.9%	93.4%	0.5%
AlexNet	86.7%	86.6%	0.1%
VGG16	94.1%	94.2%	-0.1%
SqueezeNet	93.2%	93.2%	0.0%
MobileNet	90.9%	90.5%	0.4%

$q \times q_w^2$. HECATE [40] and ELASM [26] aim for optimal scale management via extensive space exploration, practical mainly for smaller models due to prolonged compile times as noted in [35]. In contrast, ReSBM introduces an efficient scale management strategy utilizing min-cut analysis.

Optimal bootstrapping placement is NP-hard for $l_{\max} \geqslant 3$ [31] but becomes polynomially solvable for $l_{\max} = 2$. Fhelipe [24] uses dynamic programming for bootstrapping based on a DFG's multiplicative depth, incorporating EVA's waterline rescaling before placement. DaCapo [35] combines liveness-based bootstrapping with PARS scale management [40]. In contrast, ReSBM introduces a minimal-level bootstrapping approach that enhances the efficiency of both bootstrapping and subsequent operations.

Bootstrapping [5], the most costly operation in FHE, consists of four main steps in RNS-CKKS: modulus raising, coeffToSlot (an FHE linear transformation), modular reduction (using a high-degree polynomial of ciphertexts), and slotToCoeff (another FHE linear transformation). The first step increases the ciphertext level, while the subsequent three, involving intensive rotation and relinearization, dominate the bootstrapping latency. Previous studies have optimized bootstrapping by adopting the baby-step-giant-step scheme for linear transformations [21] and the Paterson-Stockmeyer algorithm for polynomial evaluations [4]. Generally, the latency of FHE operations escalates with increasing ciphertext levels. This paper introduces minimum-level bootstrapping for the first time, which reduces the ciphertext level after modulus raising to the minimal level required for subsequent operations, thereby enhancing bootstrapping efficiency.

7 Conclusion

In this paper, we present ReSBM, a novel approach for compiling machine learning models to support encrypted inference using RNS-CKKS—the only FHE scheme that supports fixed-point arithmetic. By dividing a program's DFG into regions, ReSBM efficiently manages scale and bootstrapping hierarchically. It optimizes rescaling and bootstrapping placements in individual regions via min-cut, identifies suitable rescaling regions within sequences to minimize latency and level consumption, and formulates a comprehensive rescaling and bootstrapping plan via dynamic programming.

While ReSBM achieves optimal SMO and bootstrapping placement within individual regions, it does not optimize across the entire DFG due to the NP-hard challenge of global bootstrapping placement. Future enhancements to ReSBM will concentrate on optimizing within bootstrapping regions, further reducing FHE operations post-rescaling, and integrating ReSBM with additional compiler optimizations. Additionally, plans are underway to extend ReSBM to frameworks like PyTorch and platforms such as GPUs.

8 Acknowledgements

We thank all the reviewers for their constructive comments. This research was supported by National Key R&D Program of China (Grant No. 2023YFB4503204).

References

[1] Al Badawi Ahmad, Bates Jack, Bergamaschi Flavio, Cousins David Bruce, Erabelli Saroja, Genise Nicholas, Halevi Shai, Hunt Hamish, Kim Andrey, Lee Yongwoo, Liu Zeyu, Micciancio Daniele, Quah Ian, Polyakov Yuriy, R.V. Saraswathy, Rohloff Kurt, Saylor Jonathan, Suponitsky Dmitriy, Triplett Matthew, Vaikuntanathan Vinod, and Zucca Vincent. 2022. OpenFHE: Open-Source Fully Homomorphic Encryption Library. In *Proceedings of the 10th Workshop on Encrypted Computing & Applied Homomorphic Cryptography* (Los Angeles, CA, USA) *(WAHC'22)*. Association for Computing Machinery, New York, NY, USA, 53–63. https://doi.org/10.1145/3560827.3563379

[2] Martin Albrecht, Melissa Chase, Hao Chen, Jintai Ding, Shafi Goldwasser, Sergey Gorbunov, Shai Halevi, Jeffrey Hoffstein, Kim Laine, Kristin Lauter, Satya Lokam, Daniele Micciancio, Dustin Moody, Travis Morrison, Amit Sahai, and Vinod Vaikuntanathan. 2018. Homomorphic Encryption Security Standard. In *Technical Report. HomomorphicEncryption.org, Toronto, Canada.*

[3] Krizhevsky Alex, Sutskever I., and Hinton G. 2012. ImageNet Classification with Deep Convolutional Neural Networks. *Advances in neural information processing systems* 25, 2 (2012).

[4] Hao Chen, Ilaria Chillotti, and Yongsoo Song. 2019. Improved Bootstrapping for Approximate Homomorphic Encryption. In *In Annual International Conference on the Theory and Applications of Cryptographic Techniques. Cham: Springer International Publishing*. 34–54.

[5] Jung Hee Cheon, Kyoohyung Han, Andrey Kim, Miran Kim, and Yongsoo Song. 2018. Bootstrapping for Approximate Homomorphic Encryption. In *In Advances in Cryptology–EUROCRYPT 2018: 37th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Tel Aviv, Israel, April 29-May 3, Part I 37*. 360–384.

[6] ANT-ACE Compiler. 2024. https://github.com/ant-research/ace-compiler.

[7] Gentry Craig. 2009. Fully homomorphic encryption using ideal lattices. In *Proceedings of the forty-first annual ACM symposium on Theory of computing*. 169–178.

[8] Gentry Craig, Sahai Amit, and Waters Brent. 2013. Homomorphic Encryption from Learning with Errors: Conceptually-Simpler, Asymptotically-Faster, Attribute-Based. In *Advances in Cryptology – CRYPTO 2013*, Canetti Ran and Garay Juan A. (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 75–92.

[9] Roshan Dathathri, Blagovesta Kostova, Olli Saarikivi, Wei Dai, Kim Laine, and Madanlal Musuvathi. 2020. 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*. 546–561. https://doi.org/10.1145/3385412.3386023

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

[11] Junfeng Fan and Frederik Vercauteren. 2012. Somewhat Practical Fully Homomorphic Encryption. Cryptology ePrint Archive, Paper 2012/144. https://eprint.iacr.org/2012/144

[12] Robin Geelen. 2021. *Bootstrapping Algorithms for BGV and FV*. Ph. D. Dissertation. KU Leuven.

[13] Charles Gouert, Dimitris Mouris, and Nektarios Georgios Tsoutsos. 2023. SoK: New Insights into Fully Homomorphic Encryption Libraries via Standardized Benchmarks. In *Proceedings on Privacy Enhancing Technologies*, Vol. 3. 154–172.

[14] Chen Hao. 2017. Optimizing relinearization in circuits for homomorphic encryption. *arXiv preprint arXiv:1711.06319* (2017).

[15] Cheon Jung Hee, Kim Andrey, Kim Miran, and Song Yongsoo. 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.

[16] Cheon Jung Hee, Han Kyoohyung, Kim Andrey, Kim Miran, and Song Yongsoo. 2019. 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, 347–368.

[17] Cheon Jung Hee, Han Kyoohyung, and Hhan Minki. 2018. Faster homomorphic discrete fourier transforms and improved fhe bootstrapping. *Cryptology ePrint Archive* (2018).

[18] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. 2017. MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications. (2017). arXiv:1704.04861

[19] Forrest N. Iandola, Matthew W. Moskewicz, Khalid Ashraf, Song Han, William J. Dally, and Kurt Keutzer. 2016. SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <1MB model size. *CoRR* abs/1602.07360 (2016). arXiv:1602.07360 http://arxiv.org/abs/1602.07360

[20] Chillotti Ilaria, Gama Nicolas, Georgieva Mariya, and Izabachene Malika. 2016. Faster fully homomorphic encryption: Bootstrapping in less than 0.1 seconds. In *Advances in Cryptology–ASIACRYPT 2016: 22nd International Conference on the Theory and Application of Cryptology and Information Security, Hanoi, Vietnam, December 4-8, 2016, Proceedings, Part I 22*. Springer, 3–33.

[21] Bossuat Jean-Philippe, Mouchet Christian, Troncoso-Pastoriza Juan, and Hubaux Jean-Pierre. 2021. Efficient bootstrapping for approximate homomorphic encryption with non-sparse keys. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 587–617.

[22] He Kaiming, Zhang Xiangyu, Ren Shaoqing, and Sun Jian. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 770–778.

[23] Simonyan Karen and Zisserman Andrew. 2014. Very Deep Convolutional Networks for Large-Scale Image Recognition. *Computer Science* (2014).

[24] Aleksandar Krastev, Nikola Samardzic, Simon Langoski, Srinivas Devadas, and Daniel Sanchez. 2024. A Tensor Compiler with Automatic Data Packing for Simple and Efficient Fully Homomorphic Encryption. In *Proc. ACM Program. Lang. 8, PLDI, Article 152, 25 pages.* https://doi.org/10.1145/3656382

[25] Eunsang Lee, Joon-Woo Lee, Jong-Seon No, and Young-Sik Kim. 2021. Minimax Approximation of Sign Function by Composite Polynomial for Homomorphic Comparison. *IEEE Transactions on Dependable and Secure Computing, 19(6)*, 3711–3727.

[26] Yongwoo Lee, Seonyoung Cheon, Dongkwan Kim, Dongyoon Lee, and Hanjun Kim. 2023. ELASM: Error-Latency-Aware Scale Management for Fully Homomorphic Encryption. In *32nd USENIX Security Symposium (USENIX Security 23)*. 4697–4714.

[27] Ducas Léo and Micciancio Daniele. 2015. FHEW: bootstrapping homomorphic encryption in less than a second. In *Annual international conference on the theory and applications of cryptographic techniques*. Springer, 617–640.

[28] Long Li, Jianxin Lai, Peng Yuan, Tianxiang Sui, Yan Liu, Qing Zhu, Xiaojing Zhang, Linjie Xiao, Wenguang Chen, and Jingling Xue. 2025. ANT-ACE: An FHE Compiler Framework for Automating Neural Network Inference. In *2025 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*.

[29] Stoer Mechthild and Wagner Frank. 1997. A Simple Min Cut Algorithm. *Journal of the ACM (JACM)* 44, 4 (1997), 585–591.

[30] Regev Oded. 2009. On Lattices and Learning With Errors and Random Linear Codes and Cryptography. *Proceedings of the Annual ACM*

Symposium on Theory of Computing 56, 6 (2009), 84–93.

[31] Marie Paindavoine and Bastien Vialla. 2015. Minimizing the Number of Bootstrappings in Fully Homomorphic Encryption. In *Selected Areas in Cryptography – SAC 2015*. Springer-Verlag, Berlin, Heidelberg, 25–43. https://doi.org/10.1007/978-3-319-31301-6_2

[32] Malik Raghav, Sheth Kabir, and Kulkarni Milind. 2023. 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*. 118–133.

[33] Dathathri Roshan, Saarikivi Olli, Chen Hao, Laine Kim, Lauter Kristin, Maleki Saeed, Musuvathi Madanlal, and Mytkowicz Todd. 2019. CHET: An Optimizing Compiler for Fully-Homomorphic Neural-Network Inferencing. In *Proceedings of the 40th ACM SIGPLAN conference on programming language design and implementation*. 142–156.

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

[35] Cheon Seonyoung, Lee Yongwoo, Kim Dongkwan, Lee Ju Min, Jung Sunchul, Kim Taekyung, Lee Dongyoon, and Kim Hanjun. 2024. DaCapo: Automatic Bootstrapping Management for Efficient Fully Homomorphic Encryption. https://www.usenix.org/conference/usenixsecurity24/presentation/cheon

[36] Halevi Shai and Shoup Victor. 2014. Algorithms in HElib. In *Advances in Cryptology – CRYPTO 2014*, Juan A. Garay and Rosario Gennaro (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 554–571.

[37] HEAAN software library. 2020. https://github.com/snucrypto/HEAAN.

[38] Lyubashevsky Vadim, Peikert Chris, and Regev Oded. 2013. On Ideal Lattices and Learning with Errors over Rings. *Journal of the Association for Computing Machinery* 60, 6 (2013), 43.1–43.35.

[39] Tommy White, Charles Gouert, Chengmo Yang, and Nektarios Georgios Tsoutsos. 2023. FHE-Booster: Accelerating Fully Homomorphic Execution with Fine-tuned Bootstrapping Scheduling. In *2023 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*. 293–303. https://doi.org/10.1109/HOST55118.2023.10132930

[40] Lee Yongwoo, Heo Seonyeong, Cheon Seonyoung, Jeong Shinnung, Kim Changsu, Kim Eunkyung, Lee Dongyoon, and Kim Hanjun. 2022. HECATE: Performance-Aware Scale Optimization for Homomorphic Encryption Compiler. In *2022 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 193–204. https://doi.org/10.1109/CGO53902.2022.9741265

[41] Zhongcheng Zhang, Ying Li, Yuyang Zhang, Zhenchuan Chen, Jiacheng Zhao, XiaoBing Feng, Huimin Cui, and Jingling Xue. 2025. Qiwu: Exploiting Ciphertext-Level SIMD Parallelism in Homomorphic Encryption Programs. In *2025 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*.

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