Loop-Aware Bootstrapping Management for Efficient Fully Homomorphic Encryption

Seonyoung Cheon

Department of Electronic and Electrical Engineering

Graduate School

Yonsei University

Loop-Aware Bootstrapping Management for Efficient Fully Homomorphic Encryption

Advisor : Prof. Hanjun Kim

A Dissertation Submitted
to the Department of Electronic and Electrical Engineering
and the Committee of the Graduate School
of Yonsei University in Partial Fulfillment of the
Requirements for the Degree of
Doctor of Philosophy in Electronic and Electrical Engineering

Seonyoung Cheon

August 2026

Loop-Aware Bootstrapping Management for Efficient Fully Homomorphic Encryption

This Certifies that the Dissertation of Cheon Seon Young is Approved

Committee Chair KIM HANJUN
2026-07-08 21:04:31

Committee Member 이용우
2026-07-06 15:18:00

Committee Member Ro Won Woo
2026-07-07 13:17:38

Committee Member Song William Jinho
2026-07-02 09:53:40

Committee Member 이동윤
2026-07-02 23:59:41

Department of ELECTRICAL AND ELECTRONIC ENGINEERING
Graduate School
Yonsei University
August 2026

감사의 글

박사 학위 과정 동안 지도해주신 김한준 교수님께 가장 먼저 깊은 감사의 인사를 전하고 싶습니다. 부족했던 제가 박사과정을 마칠 수 있었던 것은 6년 동안 아낌없이 지도해주시고, 자유롭게 연구할 수 있는 좋은 환경을 만들어주신 교수님 덕분입니다. 연구의 방향을 잡지 못해 헤매던 순간에도 기다려주시고 조언해주신 덕분에 조금씩 연구자로 성장할 수 있었습니다. 늘 부족한 저를 믿고 이끌어주신 교수님께 진심으로 감사드립니다.

박사 학위 심사위원을 맡아주신 송진호 교수님과 노원우 교수님, 그리고 그동안 논문 지도와 학위 심사를 함께해주신 이동윤 교수님과 이용우 교수님께도 깊이 감사드립니다.

송승빈 박사님, 이용우 교수님, 정신녕 박사님, 동관오빠, 재호오빠, 희림오빠, 근우오빠, 성우오빠, 현준오빠, 주민, 해은, 호윤, 현호, 찬, 건모, 성호, 재민 모두 감사드립니다. 여러분들과 학위과정을 함께 할 수 있어서 정말 즐거웠습니다. 연구하면서 훌륭한 분들의 도움 덕분에 학위논문을 완성하고 박사과정을 마무리할 수 있었습니다. 뿐만 아니라 학회에 함께 참여하고, 신촌에서 놀고, 같이 커피를 마시던 순간들까지도 소중한 기억으로 남아있습니다. 평생 잊지 못할 추억 만들어주셔서 다들 너무 감사드립니다.

대부분의 시간을 연구실에서 보냈지만, 연구실 밖에서 늘 편안함과 힘이 되어준 가족들과 친구들에게도 깊이 감사드립니다. 제 선택을 항상 존중해주시고 격려해주신 가족들에게 감사드립니다. 10년 넘게 만나면서 언제나 웃음을 주었던 친구들 가은, 한나, 민지, 유진에게 고마움을 전합니다. 박사과정의 고민과 어려움을 함께 공유할 수 있었고 서로 이해해주던 민정, 고산, 고재영에게도 감사의 말을 전합니다. 대학교 때부터 지금까지 함께하며 즐거운 추억을 많이 만들어준 공학 9반 친구들에게도 감사드립니다.

마지막으로, 박사과정 동안 기쁜 순간과 힘든 순간을 가장 가까이에서 함께해준 남자친구 여동현에게 진심으로 감사드립니다.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ALGORITHMS

ABSTRACT

Loop-Aware Bootstrapping Management for Efficient Fully Homomorphic Encryption

Fully homomorphic encryption (FHE) enables computation directly on encrypted data without decryption, making it a promising approach for privacy-preserving machine learning in untrusted environments such as cloud computing. Among FHE schemes, RNS-CKKS is particularly well suited for machine learning because it supports fixed-point arithmetic and SIMD parallelism. However, repeated multiplications in RNS-CKKS consume the finite levels of a ciphertext. To continue computation beyond this limit, bootstrapping must recover consumed levels, but it is the most expensive operation in RNS-CKKS. Moreover, bootstrapping placement affects both the number of bootstrapping operations and the cost of subsequent FHE operations through scale management. Thus, optimizing placement requires considering overall program performance, making manual placement difficult and often inefficient.

This dissertation presents FORTE, a bootstrapping management compiler for fully homomorphic encryption that automatically inserts and optimizes bootstrapping operations. FORTE addresses two key challenges: placement for programs with static control flow and management for programs containing loops with dynamic iteration counts.

For programs with static control flow, FORTE introduces liveness-aware candidate selection, which analyzes live-out ciphertexts to identify insertion points that require the fewest simultaneous bootstrapping operations. FORTE further refines this candi-

date set through bypass-edge analysis, which excludes long-lived ciphertexts that do not require bootstrapping. Given the resulting candidate set, FORTE estimates placement costs under different scale-management plans and uses dynamic programming to select the lowest-latency plan. Evaluation on programs with static loop counts shows that FORTE achieves a geometric-mean speedup of 1.26× over manually implemented FHE programs by automatically finding lower-latency bootstrapping placements.

For programs containing loops, FORTE extends its management framework with loop-aware code generation and optimization. FORTE generates iteration-consistent loop code by matching the encryption status and levels of loop-carried ciphertexts across iterations through loop peeling and bootstrapping placement. To reduce the overhead introduced by iterative execution and level matching, FORTE applies three loop-aware optimizations: packing loop-carried variables into a single ciphertext, level-aware loop unrolling, and bootstrapping target-level tuning. For programs with dynamic loop counts, FORTE achieves 27% performance improvement over the state-of-the-art compiler that relies on full loop unrolling.

Keywords: Fully Homomorphic Encryption, Bootstrapping, Compiler Optimization, Privacy-preserving Machine Learning, RNS-CKKS

1. Introduction

The rapid growth of user data has driven significant advances in domains such as machine learning, healthcare, and financial services. However, the use of sensitive personal data in these applications raises substantial privacy concerns. Regulatory frameworks, including General Data Protection Regulation (GDPR) and Health Insurance Portability and Accountability Act (HIPAA), impose strict requirements on the protection of user data against unauthorized access. Consequently, enabling outsourced server-side computation without compromising individual privacy has become a critical challenge, motivating the development of reliable privacy-preserving computation techniques.

As a promising solution to this challenge, fully homomorphic encryption [1] (FHE) has emerged as a revolutionary cryptographic primitive in untrusted environments. FHE enables computations to be performed directly over encrypted data, and its decrypted outputs are identical to those computed on plaintext data.

1.1 Privacy Risk in Outsourced Computation

In third-party outsourced computation, a client sends a computation task, such as machine-learning inference, to an external server. Figure 1.1 illustrates a plaintext outsourced computation scenario. The server performs the computations while directly accessing the user data, which may expose private information to the server environment.

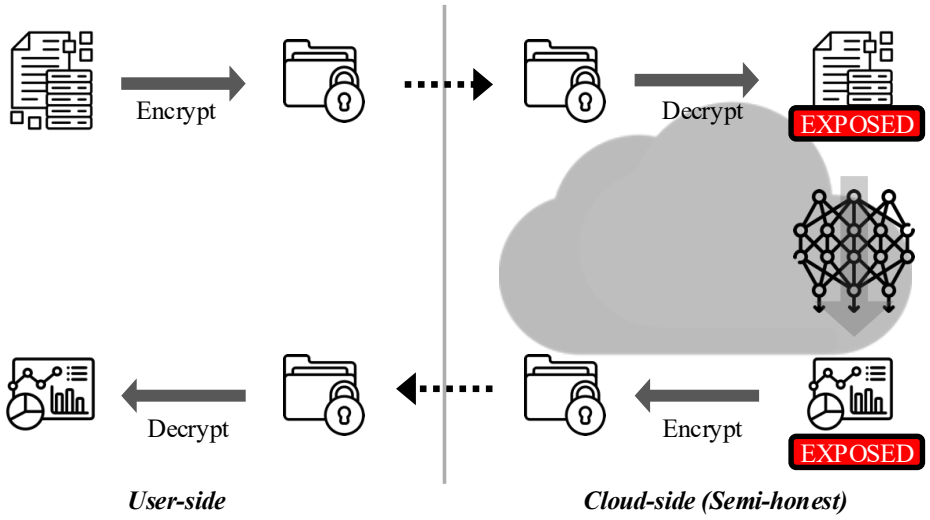


Figure 1.1: Privacy Issue in Server Computation

FHE generally targets outsourced computation under a *semi-honest* or honest-but-curious server model. Under the semi-honest model, the server follows the given computation and returns the result correctly, but the server may inspect accessible information during execution to infer the user's private data. Using FHE, the client encrypts the input data before sending the data to the server. The server then performs the requested computations on ciphertexts. Since homomorphic operations preserve encryption throughout computation, the intermediate values generated during server-side execution also remain encrypted. Therefore, the server performs the computation without directly observing the user's input data, intermediate values, or final output.

This dissertation follows the standard semi-honest threat model used in FHE-based outsourced computation. The client-side components are responsible for preparing the private inputs, encrypting the data, and decrypting the final result. The server-side components execute the compiler-generated homomorphic computation on encrypted data.

1.2 Cost of Encrypted Computation

FHE is commonly used to support privacy-preserving applications such as machine-learning inference, deep learning, and statistical analysis. Machine-learning inference and deep learning often have static execution patterns, where the computation structure is fixed by the trained model. In contrast, some statistical and machine-learning workloads may have dynamic execution patterns, where the amount of computation depends on runtime conditions. In both cases, practical FHE applications often require long sequences of arithmetic operations. FHE ciphertexts contain noise for security, and this noise is closely related to the Ring Learning with Errors (RLWE) problem [2]. While noise protects the underlying data, the amount of noise also affects the correctness of the decrypted result. Repeated operations on encrypted data, particularly multiplications, increase the amount of noise in ciphertexts.

To reduce this noise growth while computing over ciphertexts, prior work has proposed various optimizations for reducing multiplicative depth. Specifically, because FHE does not natively support non-linear functions, such functions are typically approximated by polynomials over additions and multiplications. To attain high-precision approximations, the degree of the polynomials needs to be increased, which in turn increases the multiplicative depth. Some prior works [3, 4] have focused on polynomial approximations for FHE to minimize the multiplicative depth.

Despite these depth-reduction optimizations, large-scale applications such as deep learning and machine learning still have high multiplicative depth, which can cause the accumulated noise to exceed the available noise budget. Once the noise budget is

exceeded, the decrypted result becomes corrupted. To prevent this corruption, FHE requires a noise-refreshing operation called *bootstrapping*, which is the most computationally expensive operation in FHE. Executing a single bootstrapping operation on a GPU takes about 100 times longer than a standard ciphertext-ciphertext multiplication. As a result, bootstrapping often dominates the execution time of FHE applications and becomes a critical performance bottleneck.

1.3 Contributions and Dissertation Organization

This dissertation makes the following contributions:

- **A Compiler for Automatic Bootstrapping Management:** This dissertation presents FORTE, a compiler that fully automates the placement and management of bootstrapping operations for FHE, relieving programmers of this error-prone burden.

- **Liveness-based Bootstrapping Candidate Analysis:** To minimize the total number of bootstrapping operations, FORTE introduces a liveness-based analysis that identifies a compact set of bootstrapping candidates.

- **Dynamic Programming-based Placement:** Building upon this candidate analysis, FORTE employs a dynamic programming algorithm that selects bootstrapping placements so as to minimize end-to-end execution latency.

- **Support for Dynamic Loop Structures:** To accommodate machine learning workloads with dynamic control flow, FORTE defines the conditions for matching encryption

status and level across iterations, thereby enabling loops whose trip counts are not known at compile time.

- **Advanced Compiler Optimizations:** FORTE incorporates a suite of optimizations, including loop-aware transformations (packing and loop unrolling) and bootstrapping target-level tuning, that further improve execution performance.

The remainder of this dissertation is organized as follows. Section 2 introduces the background of fully homomorphic encryption and the RNS-CKKS scheme. Section 3 introduces related work on FHE libraries, compilers, bootstrapping placement, and privacy-preserving machine learning. Section 4 presents the overall FORTE compiler design. Section 5 presents the bootstrapping candidate analysis and the latency-aware placement algorithm. Section 6 presents loop-enabled code generation for iterative FHE programs. Section 7 presents the compiler optimizations that reduce bootstrapping overhead and improve end-to-end performance. Section 8 evaluates FORTE across a range of real-world applications. Section 9 discusses limitations and broader implications, and Section 10 concludes the dissertation.

2. Background

This chapter describes the RNS-CKKS [5] fully homomorphic encryption scheme, including its encoding and encryption processes, arithmetic operations, and scale management operations in Section 2.1. Additionally, Section 2.2 introduces the operation constraints required for level management and the SCF dialect constraints required to represent structured control flow in dynamic execution environments.

2.1 RNS-CKKS Parameters and Operations

Among various FHE schemes, RNS-CKKS provides SIMD computation (vectorized parallel computation) and fixed-point arithmetic operations. These characteristics make RNS-CKKS especially suitable for machine learning workloads involving real or complex valued computations, where a bounded degree of approximation is acceptable. The following section focuses on the key parameters and operations required to describe computations in RNS-CKKS.

2.1.1 Preliminaries

The security, precision, and performance of the RNS-CKKS scheme depend on several core cryptographic and encoding parameters. Table 2.1 summarizes the fundamental parameters used throughout this dissertation.

Table 2.1: Summary of RNS-CKKS parameters and operations.

Notation	Description
N	Polynomial modulus degree
Q	Coefficient modulus ($Q = \prod q_i$)
q_i	Prime modulus in the RNS basis
S_f	Rescaling factor
L	Maximum level (initial level after `bootstrap`)
l	Current level of a ciphertext
m	Current scale of a ciphertext
S_w	Minimum scale waterline
`addcc`/`addcp`	Add two ciphertexts / a ciphertext and a plaintext
`mulcc`/`mulcp`	Multiply two ciphertexts / a ciphertext and a plaintext
`rotate`	Circular shift on the encrypted-value vector
`modswitch`	Consume one level without changing the scale
`rescale`	Consume one level while reducing the scale by S_f
`bootstrap`	Recover level to L, resetting scale to S_f

The polynomial modulus degree N determines the degree of the cyclotomic polynomial ring used in the scheme. A higher N increases the mathematical complexity of the underlying Ring Learning with Errors (RLWE) assumption [2], providing stronger security (e.g., 128-bit security) but at the cost of higher computational latency for FHE operations. The coefficients of these polynomials are bounded by a large integer known as the coefficient modulus Q.

Since Q is extremely large (e.g., $Q \approx 2^{1500}$), the polynomial coefficients are also too large to compute on standard hardware. To enable computation on real hardware, RNS-CKKS decomposes each polynomial into a set of smaller *residue polynomials*, where each residue polynomial has coefficients reduced modulo a small prime. These small primes $\{q_0, \ldots, q_{L-1}\}$ are chosen such that their product equals Q ($Q = \prod q_i$), forming the RNS basis. This decomposition is grounded in the Chinese Remainder Theorem (CRT): given

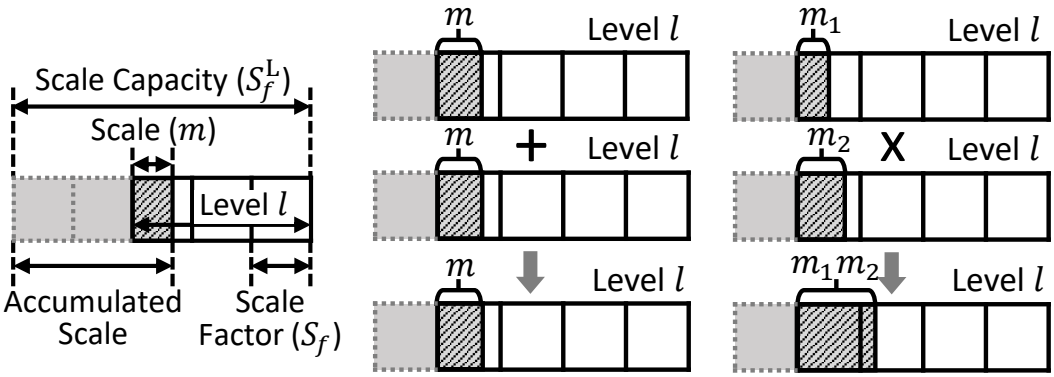
a set of pairwise coprime moduli $\{q_0, q_1, \dots, q_k\}$, any integer $x < Q$ can be uniquely reconstructed from its residues $(x \bmod q_0, x \bmod q_1, \dots, x \bmod q_k)$.

An encrypted or bootstrapped ciphertext starts at the maximum initial level L. As computation proceeds, multiplication consumes the prime moduli in the ciphertext. The number of remaining prime moduli is called *level* l of the ciphertext. In other words, repeated multiplications decrease this level, and bootstrapping recovers the levels back to the initial level L (or specified level).

2.1.2 Plaintext and Ciphertext

Prior to encoding, real- or complex-valued input data are packed into a vector of length $N/2$, which corresponds to the number of available slots. This packing mechanism enables multiple values to be represented within a single plaintext, supporting SIMD-style batched computation and improving computational throughput. The RNS-CKKS scheme manipulates two principal data representations: **plaintexts** and **ciphertexts**.

The encoder first maps the input vector to a polynomial in the ring through the inverse Fast Fourier Transform (iFFT). After that, the encoder multiplies the values in the vector by a scale m (e.g., 1.23 is encoded as 123 with scale 100). Homomorphic multiplications accumulate this scale; to prevent scale overflow beyond the maximum capacity ($Q \approx {S_f}^L$), the scheme applies a fixed rescaling factor S_f to divide the ciphertext and reduce the scale. (e.g., $1.23 \times 1.23 = 1.5129$ is encoded as 15129 with scale $10^4 = 10^2 \times 10^2$). To guarantee precision and control the cryptographic noise, the ciphertext scale m must remain above a predefined minimum threshold, the scale waterline S_w. The scaled coefficients are then reduced with respect to the RNS coefficient modulus



(a) Ciphertext definition. (b) Addition. (c) Multiplication.

Figure 2.1: RNS-CKKS ciphertext representation and arithmetic operations.

basis $\{q_0, q_1, \ldots, q_L\}$, yielding the corresponding residue representation. The resulting polynomial in the ring $\mathcal{R}_Q = \mathbb{Z}_Q[X]/(X^N + 1)$ is referred to as a plaintext. This single polynomial form is denoted as plaintext μ before encryption.

The encryptor encrypts the plaintext μ into a ciphertext $(c_0(X), c_1(X)) \in \mathcal{R}_Q^2$, where $(c_0, c_1) = (-A \cdot s + \mu + e,\ A)$, s is the secret key, A is a random polynomial, and e is cryptographic noise. The Ring Learning with Errors (R-LWE) assumption [2] guarantees the security of this construction: no efficient adversary can distinguish noisy ring elements from uniformly random ones. Under this assumption, the underlying data remains computationally hidden while still allowing homomorphic evaluation over the ciphertext.

2.1.3 Homomorphic Arithmetic Operations and Rotation

Figure 2.1a describes an RNS-CKKS ciphertext characterized by its scale and level, where the maximum scale capacity is ${S_f}^L$ and the initial level $L = 5$ is represented by the five boxes in the figure. The *accumulated scale* denotes the total scale growth incurred

through multiplications and must remain below the scale capacity ${S_f}^L$. Otherwise, scale overflow occurs, corrupting the result. The scale m, excluding the scale factors already consumed, should also remain above a predefined minimum scale, referred to as the *waterline* S_w, in order to control the noise introduced by RNS-CKKS operations. Based on these representation, RNS-CKKS supports basic arithmetic between vector operands as well as rotation operations, which together can be used to implement application-level algorithms.

Addition and Multiplication. In RNS-CKKS, vectorized arithmetic can be performed either between two ciphertexts or between a ciphertext and a plaintext. These operations can be performed either between two ciphertexts, using `addcc` and `mulcc`, or between a plaintext and a ciphertext, using `addcp` and `mulcp`. In general, arithmetic operations at higher levels incur greater cost than those at lower levels, and multiplication is more expensive than addition.

Figure 2.1b and Figure 2.1c illustrate the addition and multiplication operations, respectively. In the case of addition (Figure 2.1b), the operands are required to have the same scale and level, and the resulting scale and level remain the same as the operands. Multiplication (Figure 2.1c) requires only the levels of the operands to be the same. The resulting level remains consistent with the operand level, while the scale increases to the product of the operand scales: *i.e.*, $m = m_1 \cdot m_2$.

Rotation. To support vectorized computation, RNS-CKKS provides the `rotate` operation, which applies a cyclic shift to the slots of an encrypted vector by a specified offset.

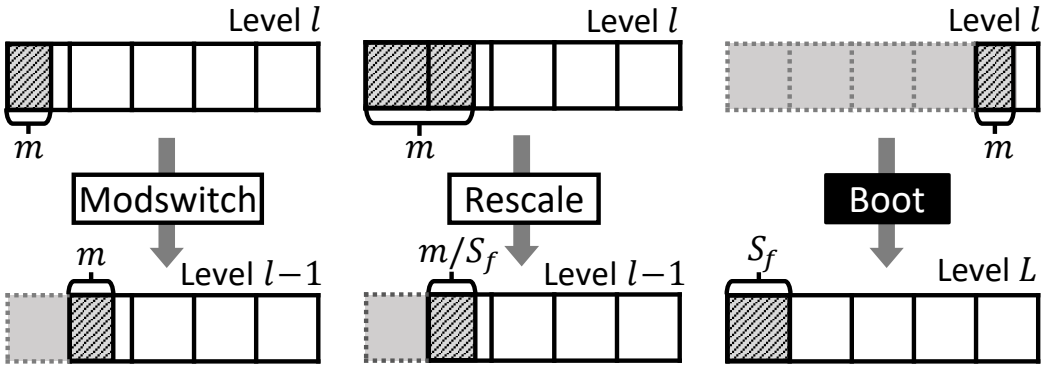
Table 2.2: Latency of FHE operations at different ciphertext levels (μs)

Operation	Level						
	4	6	8	10	12	14	16
Arithmetic operations and rotation							
`addcp`	24	32	40	47	54	64	70
`addcc`	33	44	54	65	74	85	94
`mulcp`	214	304	385	459	533	604	675
`mulcc`	1218	1513	1795	2306	2572	2927	3151
`rotate`	890	1080	1239	1699	1864	2035	2225
Scale management operations							
`modswitch`				11			
`rescale`	283	382	473	558	643	729	815
`bootstrap`	-	257787	276211	296115	313925	333571	354762

For example, rotating a four-slot ciphertext by one position transforms (c_0, c_1, c_2, c_3) into (c_1, c_2, c_3, c_0). The `rotate` operation is one of the more expensive homomorphic operations excluding bootstrapping, with a latency comparable to that of `mulcc`. Unlike multiplication or scale-management operations, `rotate` does not change the scale or level of the ciphertext.

2.1.4 Scale Management Operations

Scale management is necessary for both correctness and performance in RNS-CKKS programs. For correctness, binary operations impose strict constraints on operand scales and levels; violating these constraints can produce invalid computation results regardless of the application logic. For performance, homomorphic operations run faster at lower ciphertext levels, as shown in Table 2.2. Therefore, compilers insert dedicated scale management (*i.e.,* level management) operations to satisfy binary-operation con-



(a) Modswitch operation. (b) Rescale operation. (c) Bootstrap operation.

Figure 2.2: Overview of FHE operations.

straints and lower the levels of subsequent operations.

Modswitch. `modswitch` drops the last prime modulus q_l from Q, decreasing the ciphertext level from l to $l-1$ without changing the scale. By reducing the number of residue polynomials, it lowers the cost of subsequent arithmetic and rotation operations on the resulting ciphertext. Unlike most homomorphic operations, the latency of `modswitch` is largely independent of the ciphertext level because it only removes the last residue polynomial from the RNS representation. Compilers such as EVA [6] exploit this property by applying `modswitch` early to accelerate subsequent operations. Figure 2.2a illustrates the `modswitch` operation, where the level of the ciphertext decreases from l to $l-1$ while the scale m remains unchanged.

Rescale. The multiplications (`mulcc`, `mulcp`) increase the scale to the product of the operand scales, which can quickly exceed the maximum scale capacity defined by the coefficient modulus Q. To prevent scale exceeding the maximum capacity ${S_f}^L$, `rescale`

divides the coefficient modulus Q by the rescaling factor S_f, decreasing the scale from m to m/S_f and the level from l to $l-1$. Figure 2.2b illustrates the `rescale` operation, which divides the coefficient modulus Q by the scale factor S_f, decreasing the scale from m to m/S_f and the level from l to $l-1$. Unlike `modswitch`, `rescale` reduces the scale in addition to consuming the level, preventing the accumulated scale from exceeding the maximum scale capacity ${S_f}^L$.

Bootstrap. FHE programs with deep multiplicative chains can exhaust all available ciphertext levels despite rescaling, leading to scale overflow and corrupted results. To continue the computation, the ciphertext should be refreshed through *bootstrapping*. Bootstrapping enables further computation by recovering the ciphertext level and resetting the accumulated scale. After bootstrapping, the accumulated scale $l \times S_f + m$ is reset to the scale factor S_f, and the ciphertext level is restored up to the maximum level L. Figure 2.2c illustrates the `bootstrap` operation, which restores the ciphertext level to L and resets the accumulated scale to S_f. Despite its role in enabling further computation, `bootstrap` is the most expensive operation in RNS-CKKS. As shown in Table 2.2, a single `bootstrap` operation takes about 100 times longer than a `mulcc`.

2.2 Constraints of Level Management

Level management is a central correctness requirement in RNS-CKKS because each FHE operation changes the ciphertext level and scale. Section 2.2.1 explains the constraints that FHE operations impose on the levels and scales of their operands and results. Section 2.2.2 describes the loop-carried constraints that arise when using MLIR SCF loops

to represent iterative FHE programs, which require careful management of ciphertext properties across iterations.

2.2.1 FHE Operation Constraints

The preceding sections describe how individual RNS-CKKS operations change ciphertext levels and scales. This subsection summarizes the constraints that these operations must satisfy for an RNS-CKKS program to remain valid throughout execution.

Operand Constraints. Before executing an operation *op*, its operands must satisfy the following:

$$l > 0 \qquad \forall op \tag{2.1}$$

$$l_{c_1} = l_{c_2} \qquad \forall op \in \texttt{addition}, \texttt{multiply} \tag{2.2}$$

$$m_{c_1} = m_{c_2} \qquad \forall op \in \{\texttt{addcc}, \texttt{addcp}\} \tag{2.3}$$

$$l > 1 \qquad \forall op \in \{\texttt{rescale}, \texttt{modswitch}\} \tag{2.4}$$

$$m \geq S_f \cdot S_w \qquad \text{for } \texttt{rescale} \tag{2.5}$$

$$l \geq 3,\ m = S_f \qquad \text{for } \texttt{bootstrap} \tag{2.6}$$

Result Constraints. After executing an operation *op*, its result must satisfy the following:

$$m \geq S_w \qquad \forall op \tag{2.7}$$

$$l \cdot S_f + m \leq L \cdot S_f \qquad \forall op \tag{2.8}$$

$$l \leftarrow L,\ m \leftarrow S_f \qquad \text{for } \texttt{bootstrap} \tag{2.9}$$

Constraints (2.2)–(2.3) enforce that binary operations receive operands at the same level, with addition additionally requiring the same scale (Section 2.1.3). Constraint (2.6) requires the operand level to be at least 3 and the scale to equal S_f before `bootstrap` can be applied; this requirement comes from the backend implementation rather than from the RNS-CKKS scheme itself. On the result side, Constraints (2.1) and (2.7) guarantee that the level stays positive and the scale stays above the waterline S_w (Section 2.1.4). Constraint (2.8) ensures that the accumulated scale does not exceed the maximum scale capacity $L \times S_f$. Constraint (2.9) states that `bootstrap` recovers the level to L and resets the scale to S_f.

2.2.2 FHE Loop-Carried Constraints in MLIR SCF

In FHE programs, ciphertext states such as level and scale must often be tracked across loop iterations to verify that every operation remains valid. The structured control flow (SCF) dialect is useful for this purpose because it makes both the loop structure and the loop-carried values explicit in the IR, allowing these cross-iteration dependencies to be analyzed directly through the loop interface. The SCF dialect provides operations for

structured control-flow constructs such as `scf.for` and `scf.if`. In particular, `scf.for` represents a counted loop with a lower bound (`%lb`), an upper bound (`%ub`), and a step value (`%step`). The operation defines an induction variable for the loop body, and the loop body is represented as a single region terminated by `scf.yield`. When a loop carries values across iterations, `scf.for` uses `iter_args` to bind initial values to the region arguments, and `scf.yield` passes updated values to the next iteration or to the final loop result.

```
%r0, %r1 = scf.for %i = %lb to %ub step %step
    iter_args(%a_iter = %a0, %b_iter = %b0)
    -> (!fhe.cipher, !fhe.cipher) {
  ...
  scf.yield %a_next, %b_next : !fhe.cipher, !fhe.cipher
}
```

In this example, `%i` is the induction variable, `%a0` and `%b0` are the initial loop-carried values, and `%a_iter` and `%b_iter` are the corresponding values inside the current iteration. The values `%a_next` and `%b_next` yielded by the loop body become the loop-carried values for the next iteration. After the loop terminates, the final yielded values are returned as `%r0` and `%r1`.

More formally, let

$$\mathbf{x}^0 = (x_1^0, x_2^0, \dots, x_n^0)$$

denote the initial loop-carried values of an `scf.for` operation. At iteration k, let

$$\mathbf{x}^k = (x_1^k, x_2^k, \dots, x_n^k)$$

denote the region arguments for the loop-carried values, and let

$$\mathbf{y}^k = (y_1^k, y_2^k, \dots, y_n^k)$$

denote the values yielded by `scf.yield`. The SCF loop-carried semantics are then captured by

$$x_j^{k+1} = y_j^k \quad \text{for each loop-carried position } j. \tag{2.10}$$

MLIR requires the static type of each loop-carried position to remain unchanged across the initial value, the region argument, the yielded value, and the final loop result.

For FHE compilation, this static typing requirement alone is not sufficient. Values with the same MLIR type may still have different encryption types, ciphertext levels, or scales. These properties must remain compatible across iterations so that the compiler can execute the same FHE loop body repeatedly. For each value v, its encryption state is defined as

$$S(v) = (\tau(v), \ell(v), s(v)), \tag{2.11}$$

where $\tau(v)$ denotes the encryption type, $\ell(v)$ denotes the remaining ciphertext level, and $s(v)$ denotes the scale. For the loop-carried type matching problem, the most important components are the encryption type and the level. The following reduced state is therefore used:

$$C(v) = (\tau(v), \ell(v)). \tag{2.12}$$

To execute one identical FHE loop body across all iterations, each loop-carried posi-

tion must preserve this reduced state:

$$C(x_j^0) = C(x_j^k) = C(y_j^k) \quad \text{for all } j \text{ and } k. \tag{2.13}$$

Equivalently,

$$\tau(x_j^0) = \tau(x_j^k) = \tau(y_j^k), \tag{2.14}$$

$$\ell(x_j^0) = \ell(x_j^k) = \ell(y_j^k). \tag{2.15}$$

3. Related Work

3.1 FHE Schemes and Libraries

This section summarizes the data types supported by FHE schemes and the characteristics of FHE libraries. Different FHE schemes are suited to different classes of applications, and FHE libraries differ in their supported hardware backends and implementation-level optimizations. Table 3.1 shows the differences among BFV/BGV, CKKS, and TFHE in terms of plaintext representation, ciphertext representation, and operation granularity. BFV/BGV [7] support exact arithmetic over integers or finite rings, making them suitable for applications that require discrete computation without approximation. In contrast, CKKS [8] supports approximate arithmetic over packed real or complex values, and is therefore widely used for numerical workloads such as machine learning and signal processing. RNS-CKKS [5] further improves CKKS by using residue number system (RNS) techniques to make homomorphic operations more efficient for large-scale computations. Meanwhile, TFHE [9] operates at the Boolean-gate level and supports efficient bootstrapping for bit-level computation, making it suitable for applications that require encrypted Boolean circuits or programmable logic. Since this dissertation targets machine learning workloads, it focuses on RNS-CKKS and its support for approximate arithmetic over packed real-valued data.

Table 3.1: FHE schemes and data types, using R_q and R_t as polynomial rings over moduli q and t. $\mathbb{T}$ denotes the torus, and n is the TLWE dimension.

Scheme	**Plaintext Type**	**Ciphertext Type**	**Operation Unit**
BFV/BGV [7]	$m \in R_t$	$\in R_q^2$	Integer
CKKS [8]	$\mathbf{z} \in \mathbb{C}^{N/2}$	$\in R_q^2$	Real/complex vector
TFHE [9]	$m \in \{0, 1\}$	$= (\mathbf{a}, b) \in \mathbb{T}^n \times \mathbb{T}$	Boolean gate

To hide the complexity and increase the efficiency of FHE programs, dedicated libraries have been developed [10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21]. Table 3.2 compares these libraries by their supported schemes, target hardware, bootstrapping implementation, and open-source availability. These libraries expose APIs for the arithmetic and scale management operations described in Section 2.1.3 and Section 2.1.4. Each operation internally incorporates algorithmic optimizations [22, 23, 24, 25, 26]; for example, bootstrapping implementations apply sparse-key encapsulation[27] to reduce latency and improve the accuracy of modular approximation[4]. Recently, GPU-accelerated FHE libraries[21, 28] have been proposed to improve the performance of FHE operations by leveraging the massive parallelism of GPUs. Most of these libraries offer low-level APIs, which give developers the flexibility to optimize FHE programs aggressively but at the cost of significant effort.

3.2 FHE Compilers

To address the programming challenges, specialized compilers [6, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50] have been developed, pro-

Table 3.2: FHE libraries and their support for schemes, hardware platforms, and bootstrapping implementation.

Library	**Supported Schemes**	**Device**	**Bootstrap Support**	**Open Source**
SEAL [14]	BFV/BGV/CKKS	CPU		✓
HEAAN [11]	CKKS	CPU/GPU	✓	✓
CHEDDAR [28]	CKKS	GPU	✓	✓
HEonGPU [21]	BFV / CKKS / TFHE	GPU	✓	✓
Lattigo [10]	BFV / BGV / CKKS	CPU	✓	✓
OpenFHE [15]	BFV / BGV / CKKS / TFHE	CPU	✓	✓

viding higher-level abstractions and optimizations that simplify the development and optimization of FHE programs. Extensive surveys [51, 52, 53] have been conducted in this domain, organizing and detailing the diversities of the libraries and compilers. Table 3.3 summarizes representative CKKS compilers and bootstrapping-management approaches, focusing on how the compilers manage scales, whether they automate bootstrapping insertion, and what optimization each compiler primarily targets.

EVA [6] introduces a new programming language for FHE together with an optimizing compiler, which adopts a rule-based approach to scale management (waterline rescaling). This compiler introduces *waterline*, representing the minimum required scale, and automatically inserts the necessary scale-management operations. However, EVA relies on fixed-factor rescaling, which prevents it from further optimization chances.

To address EVA's limitation, Hecate [32] adopts proactive rescaling (PARS) and an exploration-based approach to identify an effective scale-management plan. Hecate introduces the `downscale` operation, and searches for an optimized plan through hill-climbing-based design-space exploration. ELASM [33] extends this exploration-based

Table 3.3: Comparison of FHE compilers and bootstrapping-management systems. A checkmark indicates automated support.

Compiler	**Scale Mgmt.**	**Bootstrap Mgmt.**	**Main Optimization**
EVA [6]	Waterline rescaling		Fixed-factor rescaling
Hecate [32]	Proactive Rescaling, Hill-climbing search		Downscale search
ELASM [33]	SNR Rescaling, MCMC Sampling Search		Error-latency search
Orion [54]		✓	Multiplexed packing
ReSBM [55]	Region Search-based	✓	Min-cut placement
Fhelipe [56]	Waterline Rescaling	✓	Automatic tensor packing
DaCapo [57]	Proactive Rescaling	✓	Latency-aware placement

approach by incorporating noise into the optimization process. Another contribution of ELASM is the observation that a fixed waterline in HE applications fails to reflect the errors introduced during scale management. Using an error-latency-aware scheme, ELASM achieves better latency and error trade-offs than prior work [6, 32]. Lee *et al.* [34] take a more analytical approach, deriving scale-management plans comparable to those of Hecate while requiring significantly less optimization time. However, the implementations of Hecate and ELASM are limited to static loops, as all loops are fully unrolled before optimization.

Several prior FHE compilers, including Porcupine [35], HECO [36], and Coyote [37], focus on improving the performance of FHE programs through efficient data layout. Among them, Porcupine automatically generates vectorized FHE kernels using comprehensive program synthesis. HECO provides an end-to-end compiler design that combines SIMD batching with circuit-level, hardware-specific, and scheme-specific optimizations, thereby enabling developers to achieve expert-level performance. Coyote, in turn,

improves the utilization of ciphertext vector formats by optimizing rotate operations and supports automatic vectorization for arbitrary applications while accounting for data movement. Despite these advances, Porcupine, HECO, and Coyote all assume static loops and handle them by unrolling during preprocessing, leaving dynamic loops unsupported. Related efforts such as CHET [31] automatically select data layouts and parameters for encryption while preserving the security and accuracy of tensor circuits. Also, EVA [6] and ALCHEMY [38] support data packing, but still require programmers to manually vectorize computations, which limits usability.

Overall, existing FHE compilers do not support loops with dynamic iteration counts. Instead, they support loops with static iterations by simply unrolling the entire loops and tracing the data flow to apply optimizations, which significantly degrades programmability. Furthermore, supporting only static iterations also presents difficulties in scaling to large programs. Whenever the number of iterations changes, the code should be recompiled, which can be a huge burden, and compile time tends to increase significantly as the number of iterations grows. Moreover, none of the compilers support large benchmarks that require bootstrapping.

3.3 Privacy-preserving Machine Learning

Table 3.4 compares representative privacy-preserving machine learning (PPML) systems in terms of their treatment of non-linear functions, noise-refresh mechanisms, and primary optimization objectives. Prior works have explored PPML using FHE and hybrid approaches. Gazelle [58] proposes a convolution algorithm for homomorphic encryption, and subsequent studies [59, 60, 61, 62, 63, 64] target CNN implementations.

Table 3.4: PPML approaches for handling non-linear functions, noise refresh for supporting large benchmarks, and main optimization of each work.

PPML	**Non-linear Func.**	**Noise Refresh**	**Main Optimization**
Gazelle [58]	Garbled Circuit		Linear-layer Optimization
Cheetah [59]	Garbled Circuit	Dec-Encrypt in Client	Automatic Parameter Tuning for Non-linear
Lee *et al.* [62]	Polynomial Approx.	Bootstrapping	End-to-end FHE ResNet
Lee *et al.* [63]	Polynomial Approx.	Bootstrapping	Multiplexed Packing for convolution
HyPHEN [64]	Polynomial Approx.	Bootstrapping	Packing to reduce memory

These systems combine multi-party computation with FHE to evaluate non-linear functions on the secure side. However, involving two parties creates an inherent trade-off between privacy guarantees and performance. While two-party protocols can reduce the cost of evaluating non-linear functions, they often require additional communication and expose intermediate values to the user for verification or evaluation. This exposure may leak information about the server-side model.

Several PPML implementations have demonstrated the feasibility of deep neural network inference over RNS-CKKS. Lee *et al.* [62] implemented the ResNet-20 with bootstrapping, though the large number of bootstrapping and convolution operations degrades performance. To reduce this overhead, Lee *et al.* [63] proposed an FHE-friendly deep CNN design that uses multiplexed packing and parallel convolution. Despite these optimizations, existing PPML implementations still depend heavily on hand-tuned code and lack automated scale/bootstrap management. This limitation forces FHE developers to manually revise bootstrapping placement whenever the waterline or model structure changes, increasing both development complexity and performance overhead.

3.4 Bootstrapping Placement

Bootstrapping enables unlimited computation on ciphertexts, but its high cost makes careful placement critical. Paindavoine and Vialla [65] prove that determining the minimal number of bootstraps and their locations is NP-complete, and propose heuristics based on mixed-integer linear programming. However, their approach considers only multiplicative depth without reflecting scale management and ignores the latency of other operations.

Recent studies after DaCapo have explored automatic bootstrap placement techniques for RNS-CKKS and related settings. DaCapo [57] proposes a method that identifies potential bootstrap locations at multiplication boundaries—points where ciphertext noise typically grows—and then refines this set by applying live-out ciphertext analysis. It finally determines an effective placement strategy using dynamic programming. Similarly, Fhelipe [56] segments the program's dataflow based on multiplicative depth and performs placement decisions at the operation level, also leveraging dynamic programming across these segments. ReSBM [55] introduces a hierarchical divide-and-conquer strategy, decomposing the dataflow graph into regions with consistent multiplicative depth. Within each region, it formulates the placement problem as a min-cut optimization, where edge weights capture the cost of bootstrapping. Orion [54], in contrast, operates at the network level by modeling neural networks as state-transition graphs. To address structural complexities such as residual connections, it isolates Single-Entry, Single-Exit regions as independent subproblems and subsequently integrates their solutions into the overall model.

4. FORTE Compiler Design Overview

4.1 Motivation

While existing RNS-CKKS compilers [6, 32, 33, 34, 36, 57] automate scale management by inserting `upscale`, `modswitch`, and `rescale` operations, they still leave two fundamental problems unresolved: limited support for automatic bootstrapping management and static-only loop support.

Limited support for automatic bootstrapping management Large RNS-CKKS applications require `bootstrap` operations to prevent scale overflow, yet no existing compiler automates their placement. Manual placement is difficult because a `bootstrap` operation must be inserted before the accumulated scale exceeds the maximum capacity ${S_f}^L$, and this bound changes as multiplications and `rescale` operations update ciphertext scales. The placement decision also has a direct performance impact: `bootstrap` is orders of magnitude slower than arithmetic operations (Table 2.2), yet minimizing only the number of bootstraps does not always minimize total latency. Because `bootstrap` resets the operand's scale and level, it changes the scale management and operating levels of subsequent operations. As a result, two placement plans with the same number of bootstraps can still produce different downstream latencies.

Static-Only Loop Support Although iterative algorithms are common in FHE workloads, state-of-the-art FHE compilers still lack support for loops with dynamic iteration counts. In particular, FHE-based machine learning often uses iterative methods, such as polynomial approximations for reciprocal, inverse square root, and exponential functions, as well as gradient descent for privacy-preserving training. In non-FHE execution, these algorithms can use data-dependent control flow, such as early termination after convergence. In FHE execution, however, the server cannot branch on encrypted values without leaking information about the data. Developers therefore fix the iteration count externally and rerun the encrypted computation with different loop counts to explore the accuracy-latency trade-off.

Existing FHE compilers do not provide scalable loop support. One reason is that an FHE loop body cannot always be reused unchanged across iterations. Each additional iteration increases the multiplicative depth, which changes the required scale/level management for subsequent iterations. Therefore, the existing compilers handle loops by statically unrolling them during preprocessing. This approach reduces programmability because developers must rewrite or regenerate the program whenever the iteration count changes. It also hurts scalability because compilation time and code size grow with the number of iterations.

4.2 FORTE Compiler Design Goal

This dissertation presents FORTE, an FHE compiler that automatically places bootstrapping operations in FHE programs with both static and dynamic loops and applies optimizations to improve end-to-end performance. FORTE takes a program written in a

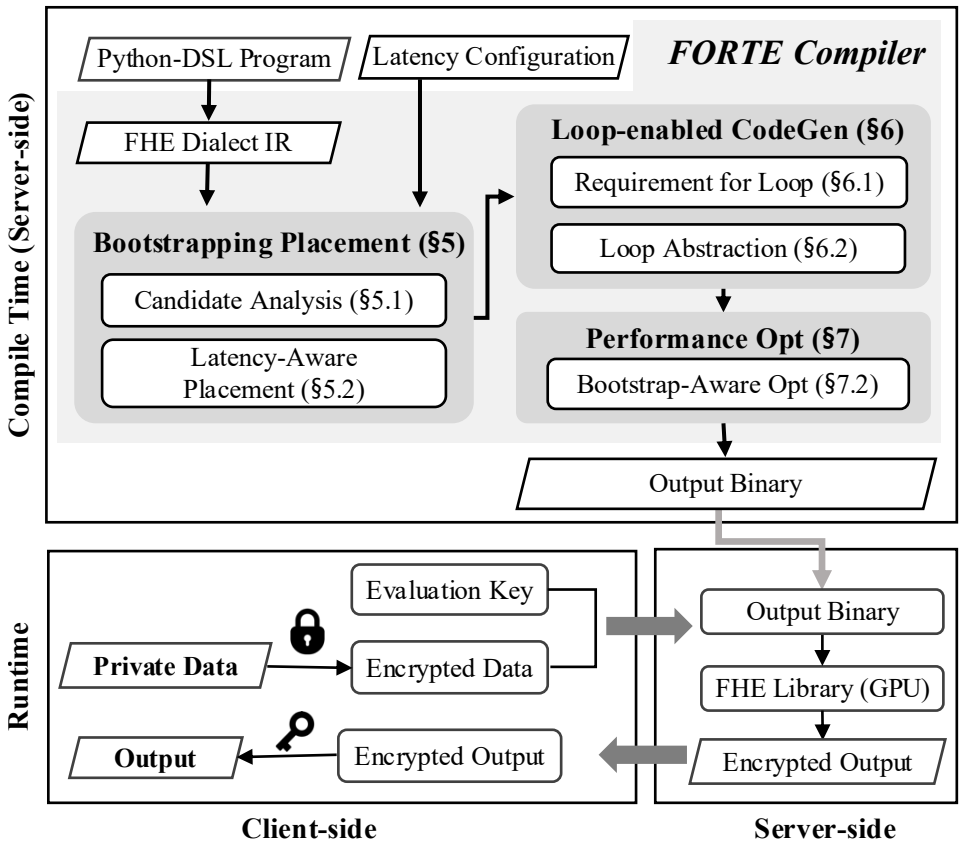


Figure 4.1: FORTE Compiler Overview

Python-based domain-specific language as input and traces the program into FHE Dialect IR, a graph-level representation consisting of arithmetic operations, rotations, and structured loop operations based on the MLIR Structured Control Flow (`scf`) dialect. Then, FORTE generates an executable binary linked with the target FHE library. The FORTE compiler is deployed on the server side, where it ahead-of-time compiles and optimizes the program written by an FHE developer. At runtime, the client transmits the user's encrypted data to the server. The compiled executable binary performs the

computations directly on the encrypted data using the underlying FHE library. The resulting output remains in an encrypted state; it is sent back to the client side, where the user decrypts the encrypted data locally to obtain the final computation result.

This design assumes that the input program explicitly specifies loop-carried variables and the number of valid elements in each input ciphertext. Because the input data remains encrypted throughout computation, FORTE does not support programs whose control-flow branches depend on input data values; branching conditions must be determined only by plaintext values.

Practicality. In existing FHE development workflows, programmers must manually identify program points where scale overflow may occur and insert `bootstrap` operations by hand—a tedious and error-prone process. This manual process demands significant engineering effort for each new application, making it impractical to extend support to diverse applications. FORTE eliminates this burden by automatically placing `bootstrap` operations at program points that prevent scale overflow, making FHE program development practical for non-expert users.

Scalability. Prior FHE compilers [32, 57] support only static computational graphs, requiring full loop unrolling before scale or level management. This approach increases both code size and compilation time proportionally to iteration count, and necessitates recompilation whenever the iteration count changes. FORTE extends the scope of automatic bootstrapping management to loop structures with both static and dynamic iteration counts, enabling the compilation of larger and more complex FHE programs.

Efficiency. Although automatic `bootstrap` placement ensures correctness, the overhead introduced by `bootstrap` operations can dominate end-to-end latency. To mitigate this, FORTE proposes three optimizations that reduce the number and cost of `bootstrap` operations, thereby producing efficient executables with lower latency.

To achieve these design goals, FORTE consists of three main modules: *Bootstrapping Placement*, *Loop-enabled Code Generation*, and *Performance Optimization*, as illustrated in Figure 4.1.

4.3 FORTE Bootstrapping Placement

The *Bootstrapping Placement* module takes the static computational graph as input and automatically schedules bootstrapping operations. Its primary role is to prevent scale overflow while simultaneously minimizing the overall execution latency.

Because `bootstrap` is the most computationally expensive RNS-CKKS operation, FORTE strategically places it to optimize performance using two main steps: candidate analysis and latency-aware placement. The candidate analysis step traverses the computational graph to identify promising program points for `bootstrap` insertion. By leveraging liveness analysis, FORTE targets points with minimal live-out ciphertexts, effectively reducing the number of required `bootstrap` operations. The latency-aware placement step then evaluates these candidates. Since the placement of a `bootstrap` resets the ciphertext levels and alters the operating levels of downstream operations, it can substantially affect overall execution time. The planner accounts for these cascading effects to select low-latency `bootstrap` locations.

4.4 Loop-enabled Code Generation

The role of this module is to enable loop structures in FHE Dialect IR, which improves scalability. If the input FHE Dialect IR takes the form of a static computational graph without loops, it bypasses this module and proceeds directly as input to the Bootstrapping Placement module. The final output of this module is a static computational graph representation, which allows the compiler to apply a unified bootstrapping placement algorithm in the subsequent stage regardless of the original loop structures. When the input FHE Dialect IR contains loop structures, this module performs two steps to preserve the loop form: *Type Alignment for Loop-carried Variables* and *Loop Abstraction*.

Type Alignment for Loop-carried Variables addresses the problem that variables with loop-carried dependencies may have different encryption statuses and levels at each iteration, causing the loop body to require different level management depending on the iteration count. Specifically, supporting unpredictable dynamic loop counts requires tailoring the level management to the varying incoming levels of ciphertexts at each iteration. Precomputing and handling all possible level states for numerous ciphertexts across dynamic loops is practically infeasible. To support loops, each iteration must execute an identical loop body; this step resolves the mismatch by aligning the types of loop-carried variables across iterations.

Loop Abstraction simplifies a program with control flow into a static computational graph representation. By abstracting loops and representing the input as a static computational graph, this step enables the application of a common bootstrapping placement technique regardless of whether the original program contains loops.

4.5 FORTE Performance Optimization

The *Performance Optimization* module further reduces end-to-end latency by optimizing the executable graph after bootstrapping placement. Because bootstrapping overhead can still dominate execution time after placement, this module applies three bootstrapping-aware optimizations. First, FORTE packs loop-carried ciphertexts into a single ciphertext to reduce the number of `bootstrap` operations. Second, the compiler unrolls loops to reduce unused levels and lower the number of `bootstrap` operations per iteration. Third, the compiler sets the target level of each `bootstrap` operation to the level required by subsequent computation, thereby reducing bootstrapping latency.

5. FORTE Bootstrapping Placement

This chapter presents the bootstrapping placement algorithms in FORTE. Section 5.1 describes the liveness analysis for bootstrapping reduction and introduces heuristic methods for identifying valid bootstrapping candidate program points. Section 5.2 describes the latency-aware bootstrapping placement algorithm, which selects a bootstrapping placement plan from the candidate set based on latency estimation. Section 5.2.2 presents the bootstrapping placement algorithm that integrates the insights introduced earlier.

Figure 5.1 presents a representative deep learning program with three convolution layers, labeled `Conv1`–`Conv3`, along with two activation functions (`Act`) and one addition operation (`Add`). The right side of the figure shows the corresponding ciphertext dataflow, where white boxes denote ciphertexts and arrows represent RNS-CKKS arithmetic and rotation operations. Each convolution layer comprises three channel groups, where each group performs one `rotate` and two `mulcp` operations, followed by a `CAdd` operation to combine the results. The activation function (`Act`) is implemented using a single `mulcc` operation. The input ciphertext x carries a scale of 2^{40}, with a waterline of 2^{40}, a rescaling factor of 2^{60}, and a maximum scale capacity of 2^{300}, corresponding to an initial level of 5.

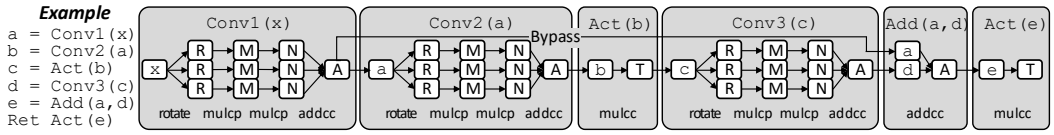


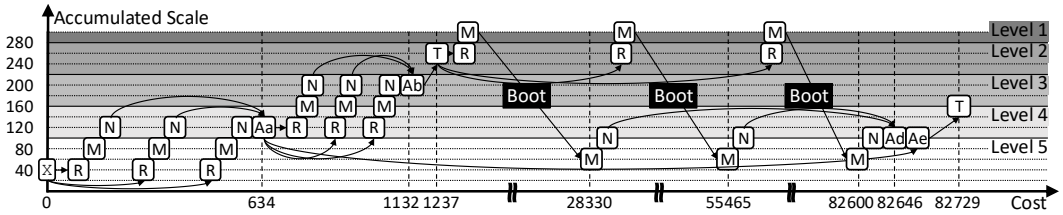
Figure 5.1: Example program and its dataflow. The example consists of three convolution layers and two activation functions.

5.1 Candidate Analysis

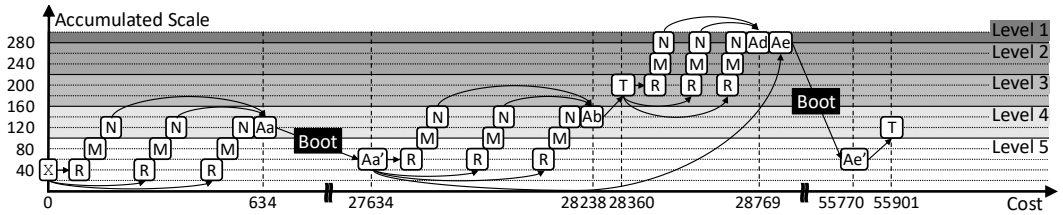
This section describes how FORTE selects candidate ciphertexts for bootstrapping. The section first presents the key observations and insights behind effective candidate selection, and then introduces the algorithm that identifies candidate ciphertexts based on these insights.

5.1.1 Observation and Insight

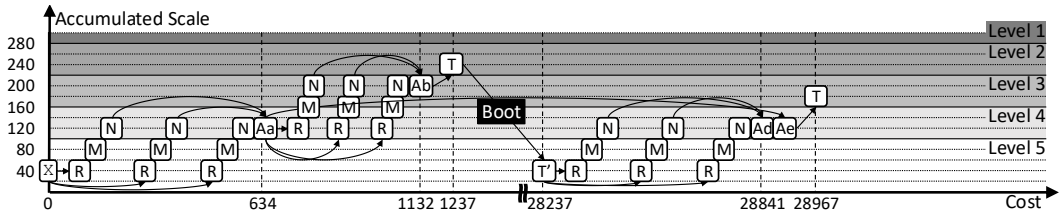
Observation 1: A straightforward accumulated-scale analysis can introduce an excessive number of `bootstrap` operations, leading to significant performance overhead. Figure 5.2a shows the changes in accumulated scale (left y-axis) and ciphertext level (right y-axis) throughout the example program, with scale-management operations omitted for clarity. The x-axis represents execution latency, computed from the profiled operation costs shown in Table 2.2. As depicted in Figure 5.2a, the naïve bootstrapping strategy inserts a `bootstrap` operation immediately before any RNS-CKKS operation whose accumulated scale would otherwise exceed the maximum scale capacity. During `Conv3`, the accumulated scale exceeds this capacity at each of the three second `mulcp` operations. To avoid scale overflow, the naïve strategy therefore inserts three `bootstrap` operations, thereby resetting the scales and levels of the three intermediate



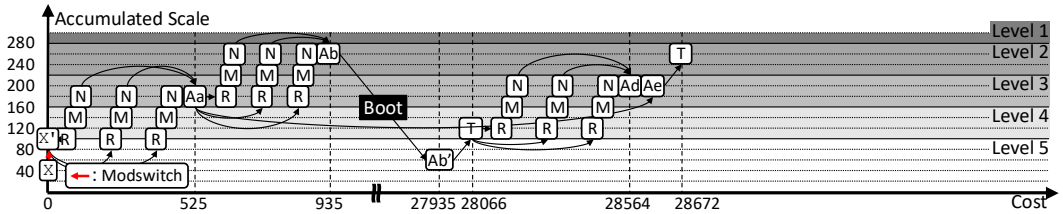
(a) Naïve placement inserts bootstrapping before the scale exceeds the capacity.



(b) Liveness-aware placement bootstraps at the latest minimum live-out point.



(c) Bypassed-liveness-aware placement excludes bypass edges from live-outs.



(d) Cost-aware placement selects the best-performance points among candidates.

Figure 5.2: Motivating example execution with different bootstrapping placement policies. All examples assume a 2^{40} waterline, a 2^{60} scaling factor, and a 2^{300} scale capacity, corresponding to maximum level 5.

ciphertexts M.

As shown in Table 2.2, the `bootstrap` operation imposes a significantly higher latency than any other RNS-CKKS operation. Minimizing the number of `bootstrap` operations therefore yields a substantial reduction in overall program latency. To address this inefficiency, FORTE applies two key insights—liveness analysis and bypass-edge analysis—to reduce the number of `bootstrap` operations and identify more favorable insertion points, as demonstrated in Figure 5.2b and Figure 5.2c.

Insight 1A: Ciphertext liveness analysis reduces the number of required `bootstrap` operations. The naïve approach inserts three `bootstrap` operations because the ciphertext dataflow diverges at `Conv3`, producing three intermediate ciphertexts (M) that are subsequently consumed and must each be bootstrapped for correctness. In compiler terms, these three ciphertexts constitute three *live-out* variables in the dataflow graph, each requiring a `bootstrap` operation. To minimize the number of `bootstrap` operations, the compiler should target program points with fewer live-out ciphertexts—specifically, program points at which the dataflow converges rather than diverges.

Building on this insight, FORTE employs *liveness*-aware bootstrapping management. At each program point, FORTE computes the number of live-out ciphertexts and identifies points with a smaller number of live-out ciphertexts as candidates for `bootstrap` insertion. For example, in Figure 5.2b, the first `bootstrap` operation is placed at an earlier point where only a single live-out ciphertext ($A\alpha$) remains—after the dataflow from `Conv1(x)` has converged and before it diverges toward `Conv2(a)`—even though the accumulated scale is still well below the maximum capacity at that point.

Insight 1B: Bootstrapping is typically unnecessary for a long-lived ciphertext while it bypasses RNS-CKKS operations. A bypass edge represents a ciphertext dependency whose source and destination are separated by a long sequence of intervening operations in a dataflow graph (*e.g.,* Figure 5.1). For instance, the ciphertext A is produced at the end of `Conv1(x)`, bypasses `Conv2(a)`, `Act(b)`, and `Conv3(c)` without participating in any of these operations, and is finally consumed by `Add(a,d)`. The dataflow edge from A in `Conv1(x)` to a in `Add(a,d)` represents a long-lived bypass ciphertext that is unlikely to require a `bootstrap` operation during the bypass period.

Including bypass ciphertexts in the liveness analysis without distinction inflates the live-out count at certain program points, obscuring optimal bootstrapping candidates and reducing optimization opportunities. Because bypass ciphertexts do not consume scale capacity during the bypass period, excluding bypass ciphertexts from the live-out count when selecting bootstrapping candidates is beneficial.

To overcome the limitations of pure liveness analysis, FORTE excludes long-lived bypass ciphertexts from the live-out count, enabling more aggressive bootstrapping reduction. As a concrete example, the live-out count at the program point corresponding to ciphertext T in `Act(b)` is two: one short-lived edge from A in `Conv1(x)` to b in `Act(b)`, and one bypass edge from A in `Conv1(x)` to a in `Add(a,d)`. A pure liveness-based approach would not identify this program point as an optimal bootstrapping candidate, producing the suboptimal placement shown in Figure 5.2b. After FORTE excludes the bypass edge, the adjusted live-out count at this program point becomes one. FORTE then identifies the superior plan shown in Figure 5.2c, which requires only one `bootstrap` operation instead of two.

5.1.2 Bootstrapping Candidate Selection

This section presents FORTE's bootstrapping candidate selector, which identifies appropriate candidate program points for `bootstrap` placement. The candidate selector evaluates the live-out ciphertexts at each program point through liveness analysis, and computes the number of required `bootstrap` operations by excluding bypass edges (Section 5.1.2). The candidate selector then finalizes the bootstrapping candidates by grouping program points according to the required number of `bootstrap` operations (Section 5.2.2).

Liveness and Bypass Edge Analysis The combined liveness and bypass-edge analysis determines which ciphertexts require bootstrapping when FORTE considers inserting a `bootstrap` operation at a given program point. Liveness analysis identifies the live-out ciphertexts that will be consumed by subsequent operations ($LiveVariable$ in Algorithm 5.1) and records the last operation where each ciphertext is used ($LastUse$ in Algorithm 5.1). Bypass-edge analysis then identifies ciphertexts that remain live but unused across long computations—that is, computations whose accumulated scale exceeds the threshold S_{th}—and excludes such bypass ciphertexts from the bootstrapping requirement.

Algorithm 5.1 presents the algorithm for generating bootstrapping candidate sets. The algorithm begins with bypass-edge analysis (Lines 2–14). For each program point (`target`) in the function, it derives the set of ciphertexts requiring `bootstrap` by removing bypass edges from the live-out set (Line 5) and then groups `target` into a candidate set according to the resulting number of required `bootstrap` operations (Line 6).

Algorithm 5.1: Bootstrap Candidate Selection

Input: *Func*: FHE function without bootstrapping
Output: *Set*: Set of bootstrapping candidates

```
1  Function CandidateSelector (Func) :
2      // Bypass Edge Analysis
3      BypassEdges, CandidatesSet ←{}
4      for each target ∈ Func do
5          BootTarget ← LiveVariable(target) − BypassEdges
6          CandidatesSet[|BootTarget|] += target
7          ScaleManaged ← ScaleManagement(Func, target)
8          for each op ∈ ScaleManaged after target do
9              if AccumulatedScale(op) > S_th and
10                 LastUse(target) after op then
11                 BypassEdges += {target}
12             if op.AccumulatedScale > S_f^L then
13                 target.Coverage ← op
14                 break
15     // Candidate Filtering
16     SetNum ← 1
17     Candidates ← CandidatesSet[SetNum]
18     valid ← False
19     while not valid do
20         valid ← True
21         ScaleManaged ← ScaleManagement(Func, Candidates)
22         for each op ∈ ScaleManaged do
23             if AccumulatedScale(op) > S_f^L then
24                 SetNum += 1
25                 if SetNum > |CandidateSet| then
26                     return fail
27                 Candidates += CandidateSet[SetNum]
28                 valid ← False
29     return Candidates
30 end
```

The algorithm next determines whether the output ciphertext of the operation at `target` forms a bypass edge and computes the `bootstrap` coverage of `target`. Assuming that a `bootstrap` operation is inserted at `target`, the algorithm simulates scale management over the function using Hecate's proactive rescaling scheme [32] (Line 7). An output ciphertext is classifed as a bypass ciphertext if it survives an operation whose accumulated scale exceeds S_{th} (Lines 9–11). The algorithm then evaluates the accumulated scale across all operations in *ScaleManaged* and marks each operation able to proceed without an additional `bootstrap` (Lines 12–14).

Algorithm 5.1 also illustrates how the combined analysis operates on the example program. Liveness analysis first counts the live variables at each program point; for example, three live variables (three Rs) appear after `Rot` in `Conv1`. Because the analysis counts live-out variables rather than individual uses, only one live variable remains after `Conv1`. Then, the algorithm finds and excludes bypass edges from the corresponding live-out sets. In this example (Figure 5.2b), Aa in `Conv1` lives until `Add`, but the accumulated scale of Ad in `Conv3` reaches 280 before `Add`, exceeding the threshold S_{th} (*e.g.,* 180). Accordingly, Algorithm 5.1 marks Aa as a bypass edge and excludes Aa from the live-variable sets associated with the operations between its definition and use. Consequently, the live-variable count after Ab decreases from 2 ($\{Aa, Ab\}$) to 1 ($\{Ab\}$).

S_{th} is a user-chosen heuristic parameter whose value directly influences `bootstrap` placement. With a larger S_{th}, Algorithm 5.1 becomes more conservative in recognizing bypass edges, which can retain redundant `bootstrap` operations for long-lived ciphertexts left unused. Conversely, a lower S_{th} causes Algorithm 5.1 to classify shorter-lived edges as bypass edges, potentially causing FORTE to incorrectly exclude ciphertexts that

require bootstrapping from the candidate set. Selecting an appropriate S_{th} value is therefore critical for achieving optimal bootstrapping placement. In the evaluation, S_{th} is set to 0.5, corresponding to half of the maximum scale capacity.

5.2 Latency-Aware Bootstrapping Placement

This section presents the latency-aware bootstrapping planner, which finds a low-latency plan satisfying level constraints from the candidates selected in Section 5.1.

5.2.1 Observation and Insight

Observation 2: Different bootstrapping placements may produce widely different end-to-end latency, because a `bootstrap` operation enables subsequent RNS-CKKS operations to execute at lower levels. Applying `bootstrap` refreshes a ciphertext by restoring its scale and level, with each `bootstrap` placement splitting the program into sub-programs around the refresh point. A sub-program with fewer ciphertext multiplications has a smaller multiplicative depth, allowing arithmetic and rotation operations to execute at lower levels and reducing overall latency. As shown in Table 2.2, RNS-CKKS operations execute faster at lower levels. A counter-intuitive consequence of this observation is that a greater number of `bootstrap` operations does not necessarily produce higher latency. Inserting more `bootstrap` operations partitions the program into more sub-programs, each of which contains fewer operations that can execute at lower levels. The latency reduction from lower-level operation execution may exceed the overhead introduced by the additional `bootstrap` operations.

Insight 2: Cost-aware bootstrapping placement is necessary to minimize end-to-end latency. Drawing from the above observation, FORTE employs the liveness analysis described above to enumerate distinct bootstrapping candidate placements, and selects the candidate with the minimum static latency estimate. Although both Figure 5.2c and Figure 5.2d placements insert only one `bootstrap` operation, the cost-aware placement chooses the position before `mulcc` in `Act(b)`. This placement enables early `modswitch`, shown by the red arrow, and lowers the execution level of the arithmetic and rotation operations in the first partition, which includes `Conv1(x)` and `Conv2(a)`, to execute at level 4. In contrast, Figure 5.2c keeps a substantial portion of the same operations at level 5, increasing the total latency.

5.2.2 FORTE Bootstrapping Placement

Candidate Selector The candidate selector identifies program points at which bootstrapping can be inserted. The selector performs three steps: ciphertext liveness analysis, bypass-edge analysis, and candidate filtering. First, *liveness analysis* step records ciphertexts that remain needed beyond each program point. Second, *bypass edge analysis* detects long-lived ciphertext dependencies left unused; for instance, the dependency from A in `Conv1(x)` to a in `Add(a,d)` in Figure 5.1 is excluded from the live-out count. The combined liveness and bypass edge analysis determines the number of valid live-out ciphertexts that require bootstrapping at each program point. Finally, *candidate filtering* step finalizes the set of candidate program points. $SetNum$ denotes the index of a candidate set and corresponds to the number of live-out ciphertexts in CandidateSet (i.e., program points with the same live-out count share the same $SetNum$). If the candidate

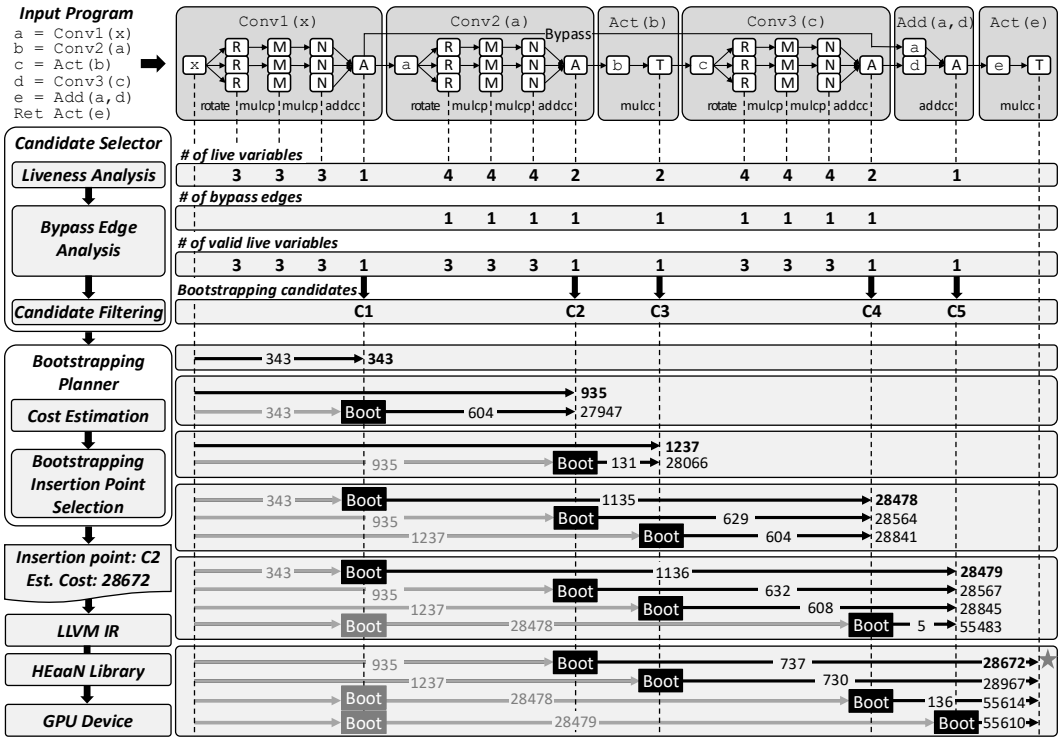


Figure 5.3: Bootstrapping planner for the example in figure 5.1. The candidate selector finds feasible insertion points from live-outs and bypass edges, while the planner estimates costs for each candidate and selects the minimum-latency plan. Bold numbers denote minimum computation costs; the star marks the selected plan.

set is insufficient to generate a correct FHE program due to scale overflow, the candidate filtering step incrementally increases the $SetNum$ until the scale overflow is resolved.

The FORTE candidate selector inserts `bootstrap` operations synchronously for all live-out ciphertexts at each selected candidate program point, excluding bypass ciphertexts. If a compiler bootstraps only a subset of the live-out ciphertexts at a given program point, the refreshed accumulated scale of the bootstrapped ciphertexts can be wasted due to the level-matching constraints of RNS-CKKS arithmetic operations. Specifically,

when two operands of an binary operation (*e.g.,* add, mul) carry different levels, the scale management scheme applies `modswitch` to reduce both operands to the lower level. If only a subset of live-out ciphertexts is bootstrapped, the refreshed accumulated scale of the bootstrapped ciphertexts is degraded to match the stale level of the unbootstrapped ciphertexts for subsequent operations. FORTE avoids this bootstrapping waste by bootstrapping all non-bypass live-out ciphertexts simultaneously at each selected candidate point.

Candidate Filtering The candidate filtering step in FORTE constructs the final candidate set by retaining bootstrapping targets with the smallest required number of bootstrappings that still admit a valid scale-management plan. If scale management fails for a candidate set—that is, if the accumulated scale of any ciphertext exceeds the scale capacity—the set must be expanded. Accordingly, candidate filtering searches for the minimum live-out $SetNum$ under which scale management remains valid. Starting from a $SetNum$ of 1, the procedure increases the $SetNum$ incrementally until it obtains a valid program.

Lines 15–28 describe the candidate-filtering algorithm. It begins by initializing the candidate set with program points that require only one `bootstrap` operation (Line 17). Scale management is then executed for the current candidate set (Line 21) to verify that no ciphertext exceeds the scale capacity. If this condition is violated, the $SetNum$ is increased and the candidate set is expanded accordingly (Lines 22–28). Once a valid candidate set is identified, the algorithm returns the finalized set of `bootstrap` candidates (Line 29).

Figure 5.3 illustrates the candidate filtering procedure applied to the example program. The procedure first considers the candidate set $\{C1, C2, C3, C4, C5\}$, where each point corresponds to exactly one valid live variable drawn from $\{Aa, Ab, T, Ad, Ae\}$. FORTE inserts `bootstrap` at these candidate points and then performs scale management. If the accumulated scale of every operation remains within the scale capacity, the candidate set is accepted. Otherwise, the live-out SetNum is increased to enlarge the candidate set.

Bootstrapping Planner Given the insertion positions generated by the candidate selector, the planner compares possible `bootstrap` placements based on the profiled cost of the corresponding RNS-CKKS computation and chooses the minimum-latency plan. The bootstrapping planner operates in two steps: cost estimation and bootstrapping insertion-point selection. First, The cost estimation step uses profiled RNS-CKKS operation costs to estimate the latency at each candidate position. Because the cost of an RNS-CKKS operation depends on its operating level, this stage implicitly performs scale management from the preceding bootstrapping point to the current candidate point in order to determine the level at which each operation executes. Then, the insertion-point selection step chooses the plan with the minimum estimated latency and finalizes the bootstrapping placement.

Algorithm 5.2 presents the algorithm for determining the bootstrapping placement plan. For each *to* candidate, the algorithm computes the minimum estimated cost from the beginning of the program to the corresponding program point. To do so, it iterates over each preceding *from* candidate, performs scale management from *from* to *to*, esti-

Algorithm 5.2: Bootstrapping Planner Algorithm

```
   Input: Set: Candidates
   Output: Set: Bootstrapping Targets
 1 Function BootstrappingPlanner (Func, Candidates):
 2     Candidates.push (Func.ReturnOp)
 3     foreach to ∈ Candidates do
 4         foreach from ∈ Candidates do
 5             if to before from.Coverage then
 6                 ScaleManaged ← ScaleManagement (Func,
                                    BestPlan[from] + to)
 7                 cost ← EstimateCost (ScaleManaged, from, to)
 8                 cost ← cost + MinCost[from]
 9                 if cost < MinCost[to] then
10                     MinCost[to] ← cost
11                     BestPlan[to] ← BestPlan[from]
12                     BestPlan[to] += to
13         end
14     end
15     return BestPlan[ReturnOp]
16 end
```

mates the latency of that subprogram, and combines it with the minimum cost already computed at *from*. The resulting value is then compared with the current minimum estimated cost at *to* to retain the lower one. Although omitted from the pseudocode, the result of scale management is memoized. A `bootstrap` placement separates scale management across the two adjacent subprograms. The following subprogram starts with the scale and level produced by `bootstrap`, independent of the preceding subprogram. Therefore, the cost for *to* is computed as the best cost at *from* plus the estimated cost of the current segment. After the search reaches *ReturnOp*, the corresponding plan is returned as the final `bootstrap` placement.

Figure 5.3 illustrates the operation of the bootstrapping planner on the example pro-

gram. The planner first performs scale management and estimates the cost of each partial program. For instance, the estimated cost of the partial program up to $C1$ is 343. To determine the minimum-cost plan up to the next candidate $C2$, the planner considers two cases. If no `bootstrap` is inserted at $C1$, the estimated cost up to $C2$ is 935, computed in the same manner as for $C1$. If a `bootstrap` is inserted at $C1$, the planner reuses the previously computed cost $MinCost[C1]$ (343) and adds the latency of `bootstrap` operation (27000) and the latency of subprogram from $C1$ to $C2$ (604), yielding a total cost of 27947. The planner compares these two estimates and selects the lower-cost plan, namely the one without `bootstrap`. The plan selected for $C2$ is then reused when evaluating later candidates such as $C3$, $C4$, $C5$, and Ret.

6. Loop-enabled Code Generation

This chapter presents the requirements for representing input programs with loops in the FHE IR and describes the code transformations used to satisfy these requirements (Section 6.1). It also describes how FORTE abstracts control flow across the program (Section 6.2) into static computational graphs that can be processed by the bootstrapping placement module.

6.1 Requirements for Loop Support in FHE

Supporting loops in FHE requires the compiler to apply an identical level management plan across all iterations. That is, placing scale management operations (`upscale`, `rescale`, `modswitch`, `bootstrap`) at the same program points must not violate the correctness of the program.

For example, suppose the levels of incoming ciphertexts vary per iteration. The compiler needs to generate different level management blocks covering all possible combinations of input ciphertext levels and dynamically route the execution to the correct block at runtime. Because the ciphertext level dictates subsequent computations within the loop, handling all possible combinations yields an exponential explosion of cases, which imposes compile-time and code-size overheads that make the approach impractical for general applications.

To support loops practically, FORTE ensures that the loop body maintains a unified level management plan, even if the levels of incoming ciphertexts change dynamically at runtime. To satisfy FHE operation constraints with the same loop body across iterations, the compiler must consider two properties: *encryption status and level*. The encryption status consists of plaintext and ciphertext, and the level is represented as an integer. The following sections discuss the type mismatch problems in Section 6.1.3 and Section 6.1.4. The FORTE compiler supports loops with static or dynamic iteration counts. However, the compiler requires the target loops to exclude ciphertext-dependent control flows. Resolving branch conditions based on ciphertext comparisons requires the evaluation of the corresponding plaintext values. Consequently, handling ciphertext-dependent control flows remains an open challenge across existing CKKS compilers [32, 33, 34, 36, 57].

6.1.1 Frontend Loop Interface and SCF Lowering

FORTE exposes loops through a frontend construct, `hc.loop`, that provides the information needed to preserve loop structure during compilation. The loop construct specifies the lower bound, upper bound, and step, and it also explicitly lists the variables that have loop-carried dependencies. In addition, the frontend provides the number of valid elements in each ciphertext through `num_elements`, which is later used by loop-aware packing and unpacking optimizations.

```
with hc.loop(0, epochs, 1,
             inputarr=[W, b],
```

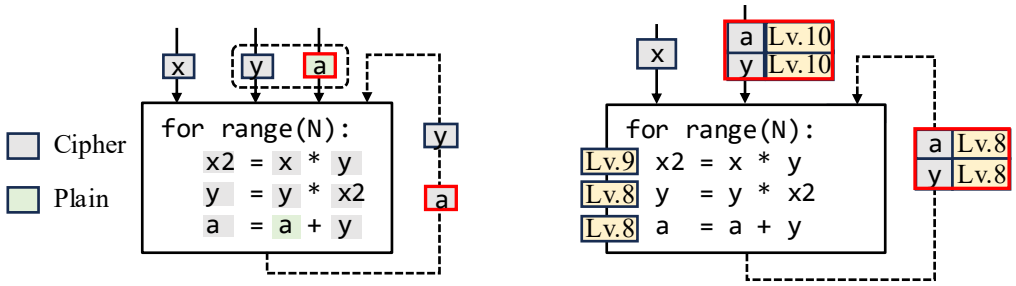
```
            num_elements=length) as i:
        W_next = ...
        b_next = ...
```

In this example, `0`, `epochs`, and `1` denote the lower bound, upper bound, and step, respectively. The variable `i` is the induction variable. The list `inputarr=[W, b]` identifies `W` and `b` as loop-carried variables whose values are updated across iterations. The `num_elements` argument records the logical number of valid elements in the ciphertexts, which may be smaller than the physical slot capacity of the underlying RNS-CKKS ciphertext.

The frontend loop is lowered to the MLIR SCF dialect in the middle end. The lower bound, upper bound, and step become the corresponding operands of `scf.for`. The loop-carried values become `iter_args`, and the updated values at the end of the loop body are returned by `scf.yield`.

```
%W_out, %b_out = scf.for %i = %c0 to %epochs step %c1
    iter_args(%W_iter = %W_init, %b_iter = %b_init)
    -> (!fhe.cipher, !fhe.cipher) {
  %W_next = ...
  %b_next = ...
  scf.yield %W_next, %b_next : !fhe.cipher, !fhe.cipher
}
```

This lowering makes loop-carried dependencies explicit in the compiler IR. For each loop-carried position, the initial value, the region argument, the yielded value, and the final result must agree in type. In FHE compilation, this agreement further requires the encryption status and the ciphertext level to be consistent across iterations, as formal-



(a) Encryption Status Mismatch (b) Level Mismatch

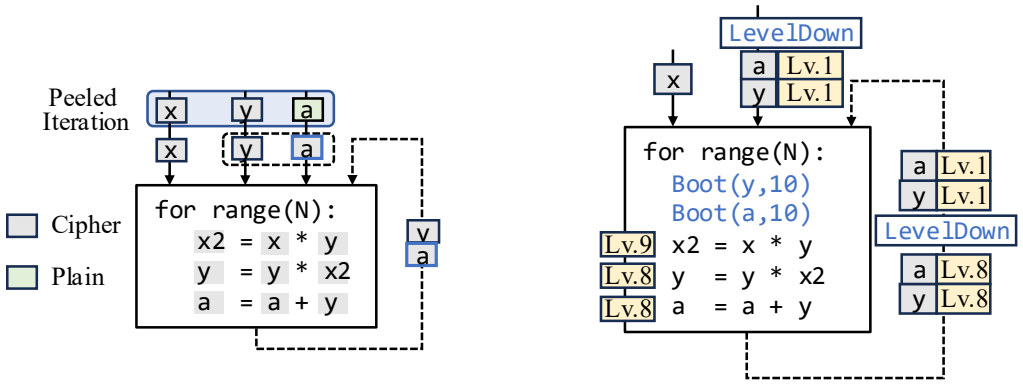
Figure 6.1: Type mismatches in loop-carried ciphertexts. Dashed arrows represent loop-carried dependencies. The initial levels of x and y are set to 10.

ized in Section 2.2.2. The next subsection describes the type mismatches that arise from this requirement.

6.1.2 Type Matching for Loop-carried Variables

Problem 1: The encryption status may mismatch on loop-carried variables. In RNS-CKKS, arithmetic operations involving `plain` and `cipher` operands (*e.g.*, `mulcp`, `addcp`) always produce a `cipher` result. This behavior can introduce an encryption status mismatch on loop-carried variables. Specifically, if a loop-carried variable initially holds a `plain` type but is updated to `cipher` through participation in a mixed-type arithmetic operation, the variable retains the `cipher` type in all subsequent iterations.

Figure 6.1 illustrates a loop-carried encryption status mismatch on the variable `a`. In the first iteration, `a` enters the loop as `plain` (depicted without a lock icon). After the first iteration, `a` transitions to `cipher` as a result of addition with the `cipher` operand `y` (depicted with a lock icon). Consequently, an encryption status mismatch arises between the initial `plain a` and the updated `cipher a` produced at the end of the first iteration.



(a) Peeling the first iteration

(b) Bootstrapping and LevelDrop on loop-carried variables

Figure 6.2: Requirements for matching encryption status and level.

Problem 2: The level type may mismatch on loop-carried variables. Successive multiplications within a loop often require level management operations such as `rescale` and `modswitch`, both of which reduce the level of ciphertext operands. These level reductions can produce a level mismatch among loop-carried variables across iterations. Figure 6.1: Challenge A-2 demonstrates this level mismatch problem. In the example, `y` and `a` both enter the loop at level 8. After two multiplications, the level of `y` is reduced to 6 via `rescale`, and the level of `a` is reduced to 6 via `modswitch` to satisfy the level alignment requirement for addition with `y`. As a result, both `y` and `a` enter the subsequent iteration at level 6.

6.1.3 Encryption-Status Alignment

Solution 1: Peeling the first iteration of a loop outside the loop. The encryption status mismatch arises exclusively during the initial iteration. Once loop-carried variables transition from `plain` to `cipher`, the resulting ciphertexts cannot revert to `plain`

Algorithm 6.1: Loop-enabled Code Generation

```
   Input: forOp: Structured For operation (Type-mismatched)
1  Function MatchTypeForLoop (forOp):
2      // Peel the first loop if type mismatch
3      if any_of(forOp.getInitArgs()) is plain then
4          peelFirstIteration(forOp)
5      // Reduce the level of loop inputs and yielded results
6      for input ∈ forOp.getInitArgs() do
7          modswitch (input)
8      end
9      for output ∈ forOp.getBody().getTerminator() do
10         modswitch (output)
11     end
12     // Bootstrap the loop-carried variables
13     for arg ∈ forOp.getBody().getArguments() do
14         bootstrapped ← bootstrap (arg)
15         arg.replaceUsesWith(bootstrapped)
16     end
17 end
```

without explicit decryption. Therefore, loop peeling—executing the first iteration outside the loop—resolves the encryption status mismatch.

Figure 6.2 illustrates the transformed program after applying loop peeling. Peeling the first iteration reduces the total iteration count from `K` to `K-1`. The variable `a` transitions from `plain` to `cipher` during the peeled iteration and remains `cipher` in all subsequent iterations. Lines 3-4 in Algorithm 6.1 presents the loop peeling procedure for resolving encryption status mismatches among loop-carried variables. FORTE triggers loop peeling when any loop body input carries a `plain` type. Because arithmetic operations such as `mulcp`, `addcp`, `mulcc`, and `addcc` always produce a `cipher` result, no intermediate value reverts to `plain` during program execution.

If a loop-carried variable enters the loop as `plain` and the corresponding output also

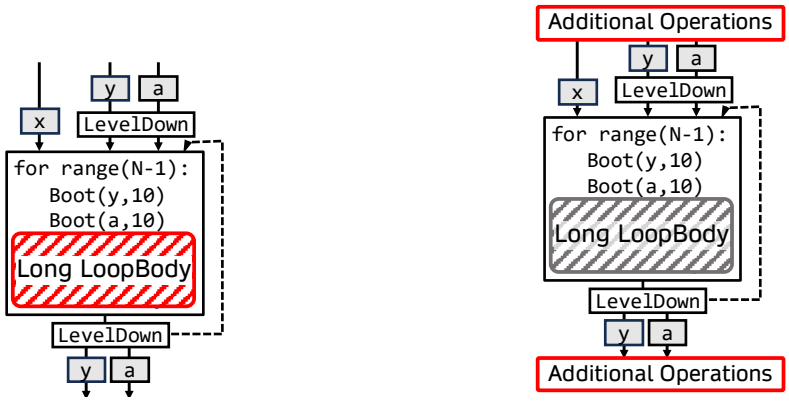
remains `plain`, no FHE operation within the loop body modifies the variable. This condition implies either the absence of an actual loop-carried dependency or the removal of the relevant operations by dead code elimination. Consequently, if a loop-carried variable produces a `plain` output in the first iteration, the encryption status of the variable remains unchanged across all subsequent iterations. Therefore, an encryption status mismatch can arise only during the first iteration. Peeling the first iteration guarantees that the encryption status of loop-carried variables remains consistent between the loop input and the loop output.

6.1.4 Level Alignment

Solution 2: Bootstrapping all loop-carried variables at the beginning of each iteration. The `bootstrap` operation reinitializes the level of a ciphertext regardless of the number of levels consumed by prior operations. Therefore, inserting `bootstrap` at the start of each loop iteration resolves the level mismatch among loop-carried variables.

Figure 6.2 illustrates the level-matching process applied to the inputs and outputs of the loop body. Before entering the loop body, FORTE applies `modswitch` to reduce the levels of all live-in and loop-carried variables—including `y` and `a`—to the minimum level. FORTE then applies `bootstrap` to all loop inputs at the entry of the loop body, restoring the ciphertext levels to the maximum value of 10.

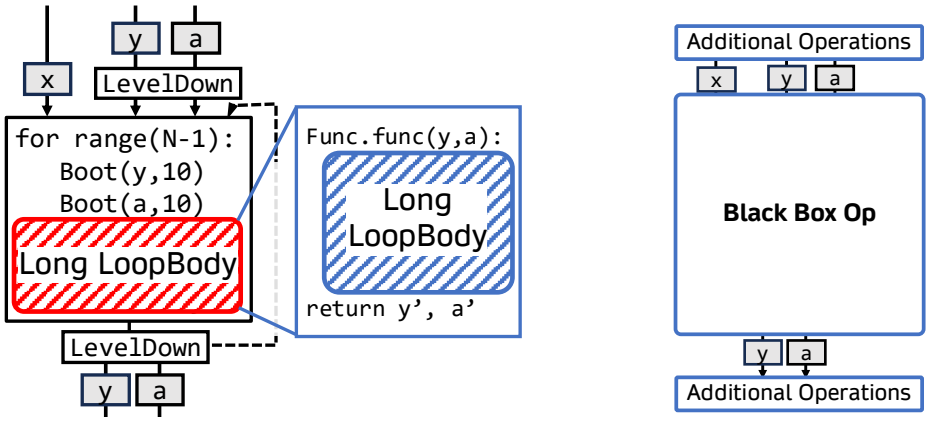
Upon completing the type-matching process, the loop is transformed into a *type-matched loop* that executes correctly across all iterations. However, a naive type-matched loop incurs redundant bootstrapping overhead. Bootstrap-based loop restructuring is therefore necessary to achieve efficient loop execution.



(a) Long Multiplication Chain Inside Loop Body (b) Outside of Loop Structure

Figure 6.3: Locations requiring bootstrapping management in FHE programs with control-flow structures.

Lines 6-16 in Line 6.1 describes the level-matching procedure for loop-carried variables. The outputs of the loop body serve as the inputs to the subsequent iteration through loop-carried variables. To align the levels of the loop outputs with the loop inputs, FORTE inserts `modswitch` operations before the loop body for the inputs and at the end of the loop body for the outputs (Lines 6-11). These insertions reduce the levels of both loop inputs and loop outputs to the minimum, producing a level-matched loop. Once the levels of the loop inputs and outputs are aligned, FORTE applies `bootstrap` to all loop-carried variables at the entry point of the loop body to restore ciphertext levels to the maximum (Lines 13-16). Bootstrapping loop-carried variables at the loop body entry ensures sufficient ciphertext levels to support long multiplication chains within the loop body.



(a) Extract Loop Body

(b) Abstract Loop as a Black-box Operation

Figure 6.4: Loop abstraction for static computational graphs.

6.2 Loop Abstraction

Using the method described in the previous section, FORTE can ensure that all ciphertexts entering the loop have the same level and the example code can execute. However, in a general application, a loop constitutes only a portion of the entire program. Therefore, bootstrapping management must be applied not only within the loop body but across the entire program structure that contains the loop. When the loop body contains a long multiplication chain, additional bootstrapping management is required within the loop body (Figure 6.3a). Furthermore, since the loop is embedded within a larger program, a long multiplication chain outside the loop may also require additional bootstrapping management outside the loop structure (Figure 6.3b). Additionally, programs may consist of a static computational graph that contains no loop structure. A unified method for applying bootstrapping management across these diverse program

structures is therefore required.

When the loop body is considered independently from the surrounding loop structure, it can be represented as a straight-line sequence of operations, equivalent to a fully unrolled loop body.

This observation yields the key insight that the bootstrapping management method designed for static computational graphs can be applied identically to the loop body. Extending this principle, abstracting control flows and representing diverse program structures as a unified static computational graph enables a single bootstrapping management method to cover all program structures. The FORTE compiler adopts this approach by abstracting control flows and representing the entire program as a static computational graph.

Figure 6.4a illustrates the process of extracting the loop body and representing the extracted body as a static computational graph. The inputs to the loop body are y and a at level 10, and the outputs are y and a at an arbitrary level. Since y and a are loop-carried ciphertexts, the level-matching procedure described in the preceding section always reduces the levels of y and a to the minimum after the loop body executes. Figure 6.4b illustrates the abstraction of the entire loop structure as a single black-box operation. The initial inputs to the loop and the result values produced after the loop correspond to the inputs and outputs of the black-box operation, respectively. Because the level-matching procedure drops the levels of all loop-carried variables to the minimum regardless of the input ciphertext levels, the output ciphertexts y and a always carry the minimum level of 1. The number of levels consumed by the black-box operation equals the initial level of y and a minus one. Treating the loop as a level-consuming operation, the

program depicted in Figure 6.4b is fully representable as a static computational graph.

This principle extends naturally to nested loops, which can also be represented as a static computational graph without loop structures. By representing the inner loop as a black-box operation, the outer loop body—in which the inner loop appears as a control-flow-free operation—can be expressed as a static computational graph. Applying this abstraction recursively enables the FORTE compiler to extend the same bootstrapping management optimizations to arbitrarily nested loops. The optimization proceeds from the innermost loop outward to the outermost loop.

7. FORTE Optimizations for Performance Improvement

This chapter describes the optimizations implemented in FORTE to improve the performance of the generated FHE code. In Section 7.1, this work first identifies the excessive bootstrapping overhead introduced by loop support, where bootstrapping operations may be repeatedly invoked at the beginning of each loop-body execution. To mitigate this overhead, FORTE suggests three optimizations : bootstrapping-aware packing (Section 7.2.1), level-aware unrolling (Section 7.2.2 and bootstrapping target level tuning (Section 7.2).

7.1 Bootstrap-Induced Performance Overhead

To maintain an identical loop body across all iterations, the compiler must align the levels of loop-carried variables by inserting `bootstrap` operations at each iteration. Figure 7.1 shows the executable loop structure after type matching, after loop peeling and bootstrapping are applied to resolve encryption status and level mismatches, respectively. Because `bootstrap` is the most expensive FHE operation, these requirements often degrade overall performance through excessive bootstrapping.

Observation 1: Bootstrapping all loop-carried ciphertexts introduces significant performance overhead. Bootstrapping loop-carried ciphertexts is necessary to re-

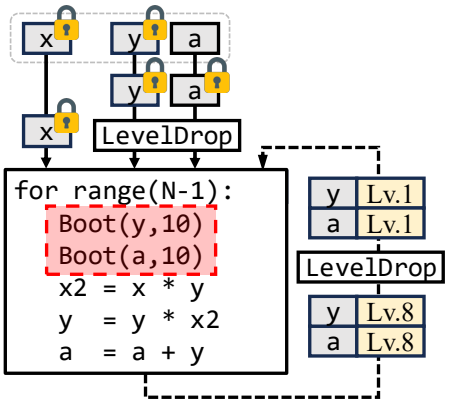
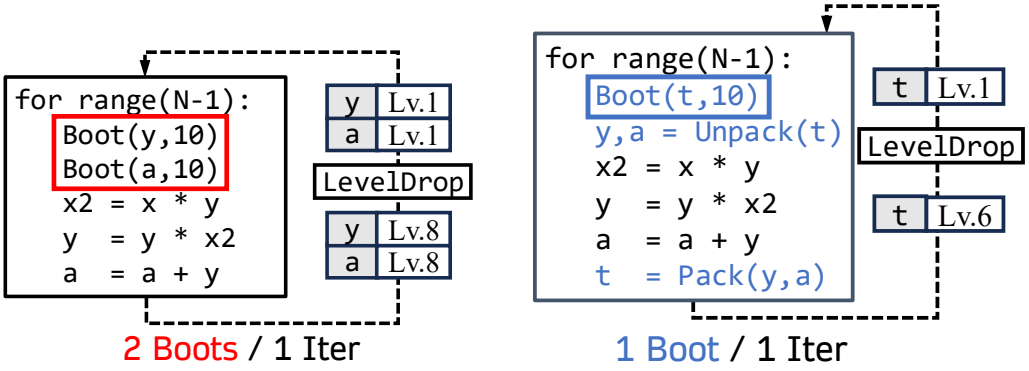


Figure 7.1: Executable Loop Structure in FHE after type matching



(a) Large overhead from bootstrapping all loop-carried ciphertexts

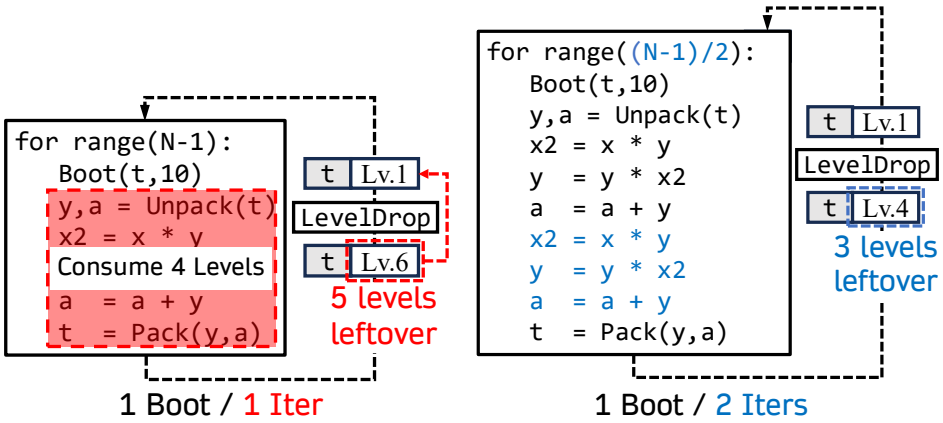
(b) Packing Loop-carried Variables

Figure 7.2: Observation (Figure 7.2a) and Insight (Figure 7.2b) from bootstrapping all loop-carried ciphertexts

solve the level type mismatch problem when supporting loops with dynamic iteration counts, for which unrolling is not feasible. However, bootstrapping all loop-carried ciphertexts simultaneously introduces significant performance overhead. In the scenario shown in Figure 7.2a, the loop carries two loop-carried variables, `y` and `a`, requiring two `bootstrap` operations per iteration. In more complex programs such as multivariate regression, which may involve nine loop-carried variables, bootstrapping all loop-carried ciphertexts per iteration incurs substantial overhead.

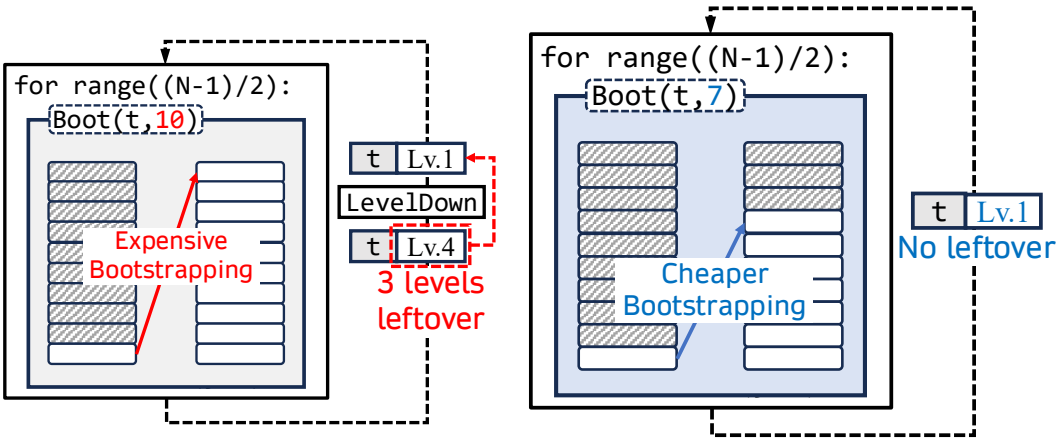
Observation 2: A loop may not fully utilize the levels recovered through bootstrapping. The levels recovered by a `bootstrap` operation may remain partially unused when the loop body is too short. For example, in the scenario shown in Figure 7.3a, the loop body consumes only 4 levels per iteration, leaving 5 levels unused. The loop then discards the unused levels via `modswitch` before advancing to the next iteration. A more efficient loop structure would enable better utilization of these excess levels.

Observation 3: Restoring ciphertext levels to the maximum via `bootstrap` may incur unnecessary overhead. As shown in Table 2.2, the latency of `bootstrap` increases with higher target levels. When the loop body does not fully utilize all recovered levels, bootstrapping loop-carried ciphertexts to the maximum level introduces unnecessary overhead. However, DaCapo [57], the state-of-the-art bootstrapping placement compiler, generates `bootstrap` operations that unconditionally restore ciphertext levels to a predetermined maximum target level. In Figure 7.4a, the `bootstrap` operation restores the loop-carried variable `t` to the maximum level 10. However, the short com-



(a) Not fully utilize the recovered level (b) Level-Aware loop body Unrolling

Figure 7.3: Observation (Figure 7.3a) and Insight (Figure 7.3b) from under-utilizing recovered level



(a) Incur Unnecessary Overhead in Bootstrapping (b) Bootstrapping Target Level Tuning

Figure 7.4: Observation (Figure 7.4a) and Insight (Figure 7.4b) from maximum target level bootstrapping

putation within the loop body consumes only 6 levels, leaving the remaining levels unused. The loop then applies `modswitch` to discard the unused levels before advancing to the next iteration, resulting in wasted computational resources.

7.2 Bootstrapping-Aware Optimizations

7.2.1 Packing of Loop-carried Ciphertexts

Insight 1: Packing loop-carried ciphertexts into a single ciphertext reduces the number of bootstrapping operations. To reduce the number of bootstrapping operations, FORTE packs multiple loop-carried ciphertexts into a single ciphertext and applies a single `bootstrap` to the packed ciphertext, rather than bootstrapping each loop-carried ciphertext individually. Because RNS-CKKS encrypts vectors as a single ciphertext, multiple ciphertexts can be packed into one ciphertext as long as the total dimension does not exceed the slot count ($\frac{N}{2}$), where N is the polynomial modulus degree. For example, in Figure 7.2a, the loop carries two loop-carried ciphertexts, `y` and `a`, requiring two `bootstrap` operations per iteration. As illustrated in Figure 7.2b, FORTE packs `y` and `a` into a single ciphertext `t`, applies `bootstrap` to `t`, and unpacks the bootstrapped `t` back into the original ciphertexts `y` and `a`. The `pack` optimization reduces the number of `bootstrap` operations from two per iteration to one per iteration. Although `pack` and `unpack` introduce additional overhead, the overhead of these operations is negligible compared to the latency of `bootstrap`.

The packing method reduces the performance overhead of bootstrapping all loop-carried ciphertexts. The `pack` operation requires the number of elements (num_e) in the

value vector, which the programmer specifies as a parameter. When the value vector size num_e does not exceed 2^{N-1} (the number of elements in a ciphertext vector), FORTE repeats the vector elements until the vector size reaches 2^{N-1}, and then encrypts the enlarged vector. The resulting ciphertext therefore contains redundant data occupying the remaining slots.

To pack ciphertexts, FORTE extracts the valid (non-redundant) data from each ciphertext using `mulcp` with a binary plaintext mask consisting of zeros and ones.

For m loop-carried variables, FORTE generates m zero-filled plaintexts. To extract the valid portion of the i-th ciphertext ($0 \leq i < m$), FORTE replaces the zeros in the i-th plaintext with ones at positions from the $(i \times num_e)$-th element to the $((i+1) \times num_e - 1)$-th element. FORTE then multiplies the i-th ciphertext with the i-th plaintext mask using `mulcp` to produce a masked ciphertext. The m masked ciphertexts are combined into a single packed ciphertext using `addcc`.

FORTE inserts `unpack` immediately after bootstrapping the packed ciphertext. The `unpack` operation uses `mulcp` and `addcc`, similarly to `pack`, but additionally requires `rotate`. To unpack, FORTE applies the same binary plaintext masks used in `pack`. Multiplying the packed ciphertext with each mask extracts the data of the i-th ciphertext from the packed ciphertext. Because the extracted ciphertext contains zeros in the unmasked slots, the ciphertext must be restored to the original all-valid-data form. FORTE achieves this restoration through iterative `rotate` and `addcc`: the ciphertext is cyclically shifted by `rotate`, and the shifted ciphertext is added to the prior ciphertext by `addcc` to fill the zero-valued slots. Repeating this rotation-and-addition procedure restores the extracted ciphertext to the original form. Although the packing method introduces ad-

ditional overhead—particularly for programs with many loop-carried ciphertexts—the overhead is justified by the latency reduction achieved through fewer `bootstrap` operations.

7.2.2 Level-Aware Loop Unrolling

Insight 2: Level-aware loop unrolling fully utilizes the levels recovered by bootstrapping. Loop unrolling reduces the total iteration count, thereby reducing the number of `bootstrap` operations. When the computation within a single loop iteration does not fully consume the ciphertext levels recovered by `bootstrap`, FORTE unrolls the loop to utilize the excess levels. In Figure 7.3a, the loop body—including the `pack` and `unpack` operations—consumes only four of ten available levels per iteration. FORTE unrolls the loop by a factor of two, consuming eight levels per unrolled iteration as illustrated in Figure 7.3b. The unrolling reduces the number of `bootstrap` operations to one per two iterations, thereby improving overall performance.

FORTE applies the level-aware loop unrolling optimization to reduce wasted levels in the loop body and to decrease the number of `bootstrap` operations per iteration. The first step computes the maximum multiplicative depth of the loop body. In the example program shown in Figure 7.3b, FORTE computes the multiplicative depth by traversing the definition-use chain starting from the loop-carried variables `y` and `a`. The multiplicative depth of `x2`—which is produced by a multiplication of `x` and `y`—is 1. The depth of `y` becomes 2 because `x2` is the operand of a subsequent multiplication. The depth of `a` becomes 2 due to the addition of `a` with `y`, which carries a depth of 2. In this example, the maximum multiplicative depth of the loop body is therefore 2. When loop-carried

ciphertexts carry different depths, FORTE selects the maximum depth ($depth_{max}$).

The second step determines the unrolling factor. The level limit $depth_{limit}$ denotes the maximum number of levels that the loop body may consume without requiring additional `bootstrap` operations. In the common case, $depth_{limit}$ equals the maximum ciphertext level after bootstrapping, L. When the loop body includes `pack` and `unpack` operations, the `mulcp` operations introduced by packing consume two additional levels, reducing $depth_{limit}$ to $L - 2$. Once $depth_{limit}$ is established, FORTE computes the unrolling factor as $\lfloor \text{depth}_{\text{limit}} / \text{depth}_{\text{max}} \rfloor$. FORTE does not apply loop unrolling if the unrolling factor is 0 or 1. After the unrolling factor is determined, FORTE replicates the loop body according to the unrolling factor and reduces the total iteration count accordingly. This approach maximizes level utilization within the loop body.

7.2.3 Bootstrapping Target-Level Tuning

Insight 3: Tuning the bootstrapping target level to the required level reduces overhead. As shown in Table 2.2, the latency of `bootstrap` decreases as the target level is reduced. Restoring only the exact level required by the subsequent computation therefore reduces bootstrapping overhead. FORTE determines the required level by traversing the dataflow chain of each ciphertext and sets the bootstrapping target level accordingly. For instance, in Figure 7.4a, the loop body requires 7 levels; FORTE therefore lowers the bootstrapping target level from 10 to 7, as shown in Figure 7.4b. As shown in Table 2.2, reducing the target level from 10 to 7 yields a latency reduction of 45,335 microseconds, which is comparable to the latency of approximately 60 `mulcc` operations.

Algorithm 7.1: Bootstrapping Target-Level Tuning

Input: *Func*: Function after scale management
Output: *Func*: Function with tuned bootstrapping target levels

Function *TuneBootstrapTargetLevel (Func):*
  // Step 1: move shared level drops earlier
  for $op \in Func$ *in reverse topological order* **do**
    $U \leftarrow \text{GETUSERS}(op)$
    if $\textit{IsBootstrap}(op) \ \vee \ \exists u \in U : \neg \textit{IsModSwitch}(u)$ **then**
      continue
    $d \leftarrow \min_{u \in U} \text{DROPFACTOR}(u)$
    $m\_op \leftarrow \text{CREATEMODDROP}(x, d)$ where $x = \text{OPERAND}(op)$
    $\text{OPERAND}(op) \leftarrow \text{RESULT}(m\_op)$
    $\text{LEVEL}(\text{RESULT}(m\_op)) \leftarrow \text{LEVEL}(x) - d$
    for $u \in U$ **do**
      $\text{DROPFACTOR}(u) \leftarrow \text{DROPFACTOR}(u) - d$
      if $\textit{DropFactor}(u) = 0$ **then**
        remove u
  // Step 2: reflect post-bootstrap drops in the target level
  for `bootstrap` *operation* $b \in Func$ **do**
    $U \leftarrow \text{GETUSERS}(b)$
    if $\exists u \in U : \neg \textit{IsModSwitch}(u)$ **then**
      continue
    $d \leftarrow \min_{u \in U} \text{DROPFACTOR}(u)$
    $b.targetLevel \leftarrow b.targetLevel - d$
    for $u \in U$ **do**
      $\text{DROPFACTOR}(u) \leftarrow \text{DROPFACTOR}(u) - d$
      if $\textit{DropFactor}(u) = 0$ **then**
        remove u
  return *Func*
end

To determine the target level for each `bootstrap` operation, FORTE first applies early `modswitch` placement through a backward analysis, as shown in Algorithm 7.1. Existing scale management approaches [6, 32, 33, 34] hoist `modswitch` operations as early as possible, enabling the entire program to operate at lower ciphertext levels. Operating at lower levels is beneficial because lower-level homomorphic operations execute faster. This algorithm is applied after scale and bootstrapping management, where explicit modswitch operations have already been inserted to align the levels of operands.

The first step of Algorithm 7.1 hoists common `modswitch` operations backward. It follows Early-Modswitch proposed in [6], with only an additional condition for handling bootstrapping operations. For each operation, FORTE skips bootstrapping operations and operations whose users are not all modswitch operations (Line 6). If all users are modswitch operations, the algorithm computes the minimum drop factor among them (Line 7). This value is the common level drop shared by all users, and therefore it can be safely moved before the current operation without changing the required level alignment. The algorithm hoists the shared level drop by inserting a new modswitch on the operand x and replacing the operand of the current operation with its result (Lines 8–10). Afterward, it subtracts the hoisted amount from the original modswitch users and removes the users that are fully covered by the hoisted drop (Lines 12–14).

After early `modswitch` placement, the presence of a `modswitch` immediately following a `bootstrap` indicates that the levels recovered by `bootstrap` are not fully consumed by subsequent operations. The second step therefore traverses bootstrapping operations and first checks whether all users of a bootstrapping result are `modswitch`

operations (Lines 17–19). If so, FORTE computes the minimum drop factor d among them and reflects this common level drop in the bootstrapping target level (Lines 20–21). As a result, `bootstrap` recovers only the levels needed by its users, reducing the latency of the `bootstrap` operation while preserving the level expected by subsequent operations.

8. Evaluation

This chapter evaluates the FORTE compiler on static loop applications (Section 8.1) and dynamic loop applications (Section 8.2).

8.1 Static Loop Count Applications

This section evaluates six benchmarks with static loop structures. Section 8.1.1 describes the experimental setup. Section 8.1.2 demonstrates the overall performance improvement of FORTE over manually placed bootstrapping. Section 8.1.3 evaluates the effectiveness of FORTE through bootstrapping count comparison. Section 8.1.4 analyzes compilation time to assess the practicality of FORTE. Section 8.1.5 reports the accuracy of the latency estimation used by FORTE. Section 8.1.6 presents a waterline sensitivity study across all benchmark applications.

8.1.1 Evaluation Setup

This dissertation evaluates six deep neural network models under two activation functions: ResNet-20/44 [66], AlexNet [67], VGG16 [68], SqueezeNet [69], and MobileNet [70]. All FHE primitives are implemented using the APIs provided by HEaaN [11], an RNS-CKKS library.

For the non-linear layers, we use polynomial approximations following the approach

Table 8.1: Classification accuracy on multiple CIFAR-10 images using RNS-CKKS models generated by FORTE.

Model	ResNet-20		ResNet-44		AlexNet		VGG16		SqueezeNet		MobileNet	
	ReLU	SiLU	ReLU	SiLU	ReLU	SiLU	ReLU	SiLU	ReLU	SiLU	ReLU	SiLU
PyTorch	90.6%	92.6%	91.2%	92.7%	89.0%	86.6%	93.8%	93.0%	89.4%	88.0%	91.4%	91.4%
FORTE	90.7%	92.6%	91.3%	92.7%	89.0%	86.6%	93.8%	92.9%	89.2%	87.9%	91.2%	91.4%

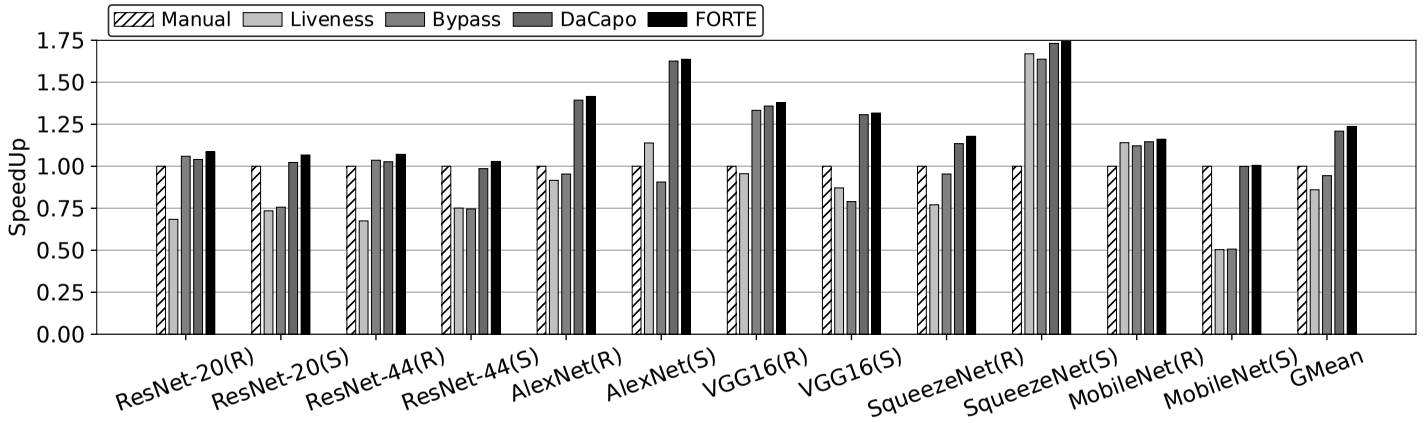


Figure 8.1: Speedup of each approach relative to Manual at waterline 2^{40}. The activation functions are denoted as (R) for ReLU and (S) for SiLU.

of [3]. Specifically, SiLU is represented as a 96-degree polynomial obtained with the Remez algorithm, resulting in a multiplicative depth of 7. ReLU is instead evaluated using a minimax composite approximation with degrees $15, 15, 27$, whose multiplicative depth is 13 [3]. For linear layers such as convolution, FORTE applies the multiplexed parallel convolution method from [63]. This technique enables sparse outputs from multiple channels to be encoded in a compact form. This work evaluates 12 benchmarks, constructed from two activation–pooling combinations: ReLU paired with max pooling (R) and SiLU paired with average pooling (S). All workloads are implemented using the RNS-CKKS scheme.

The correctness of FORTE-generated programs is validated by comparing their classification accuracy against that of PyTorch on 1,000 CIFAR-10 images at a waterline of 2^{40}. The resulting accuracy is reported in Table 8.1. ResNet-20(R), SqueezeNet(R), SqueezeNet(S), and MobileNet(R) show only a 0.1% accuracy difference. Although approximate activation functions introduce a maximum error of 13 bits, the resulting impact on classification accuracy remains negligible.

In the absence of prior bootstrapping compilers, the evaluation adopts manually placed bootstrapping (**Manual**) as the baseline. Following the state-of-the-art ResNet-20/44 design [63], which places `bootstrap` operations before activation layers only when the remaining level budget is insufficient, bootstrapping is manually inserted for each benchmark. This manual process, including insertion, verification, and optimization, requires approximately 4 hours per benchmark for each waterline configuration. Because an incorrect `bootstrap` placement can cause FHE execution errors such as ciphertext scale overflow, the process requires careful and time-consuming analysis and

validation.

In the manual baseline, `bootstrap` operations are inserted between layers. In addition, because the sign function used in ReLU and max pooling requires bootstrapping, an extra `bootstrap` operation is inserted at the multiplication of the second 15th-degree polynomial term. The manual baseline uses 2^{40} as the scale-management waterline.

8.1.2 Latency Comparison

Figure 8.1 reports the speedup in single-image CIFAR-10 inference time achieved by Liveness, Bypass, DaCapo, and FORTE relative to Manual. Liveness reduces the number of `bootstrap` operations, achieving performance comparable to Manual on VGG16(R/S), and improving performance on AlexNet(S) and SqueezeNet(S). Liveness improves performance by finding `bootstrap` positions that manual placement can miss, including positions inside activation functions. Bypass expands the feasible candidate set and exposes additional beneficial `bootstrap` positions. DaCapo applies dynamic programming over the selected candidate set to determine bootstrapping placement, using a cost model that accounts for the latency of surrounding FHE operations. FORTE follows this latency-aware placement approach and further optimizes the generated bootstrapping operations through target-level tuning.

On ResNet-20/44(R) and VGG16(R), Bypass achieves improved latency by identifying appropriate bootstrapping positions through liveness analysis. However, VGG16(S) and AlexNet(S) incur higher total latency, as shown in Figure 8.2, because non-`bootstrap` operations become more expensive. FORTE achieves a geometric mean speedup of 1.26× over Manual. FORTE outperforms Manual because FORTE accounts for bootstrapping

Table 8.2: Number of bootstrapping operations for each benchmark under manual placement and FORTE.

Model	ResNet-20		ResNet-44		AlexNet		VGG16		SqueezeNet		MobileNet	
	ReLU	SiLU	ReLU	SiLU	ReLU	SiLU	ReLU	SiLU	ReLU	SiLU	ReLU	SiLU
Manual	37	19	85	43	66	12	54	20	58	18	53	27
FORTE	37	19	85	43	67	12	53	20	57	19	61	27

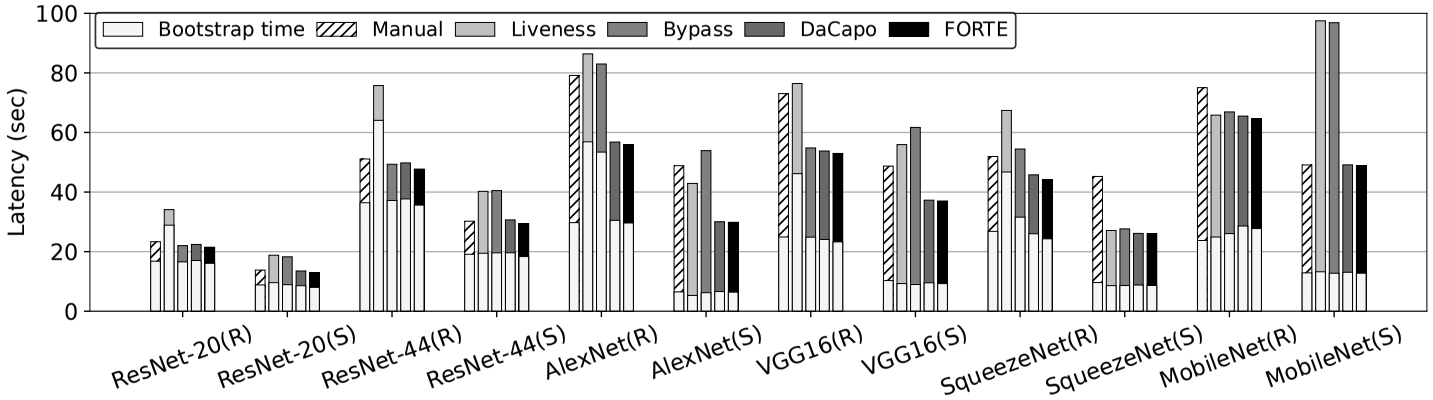


Figure 8.2: Single-image inference latency. The white portion of each bar represents the latency overhead caused by bootstrapping.

placement opportunities that manual analysis may overlook. FORTE analyzes the detailed RNS-CKKS operations that constitute deep learning model APIs. Although a developer can inspect the provided library and insert `bootstrap` manually, the optimal bootstrapping position varies with each function call due to scale management variability. FORTE accounts for scale management variations induced by each `bootstrap` placement, enabling FORTE to identify lower-latency bootstrapping plans than Manual.

8.1.3 Bootstrapping Count

Table 8.2 shows that the number of `bootstrap` operations is similar between Manual and FORTE. Nevertheless, FORTE achieves lower latency than Manual on AlexNet, VGG16, and SqueezeNet, as shown in Figure 8.2, because FORTE places `bootstrap` operations at positions that reduce arithmetic operation latency. Notably, in the MobileNet benchmark, FORTE inserts a greater number of `bootstrap` operations than Manual yet still achieves lower total latency. This result demonstrates that arithmetic operation latency influences end-to-end latency, even though each individual arithmetic operation has latency several orders of magnitude lower than a `bootstrap` operation. In MobileNet(R), the high channel count leads to many `rotate` and multiplication operations. As the accumulated cost of these arithmetic operations becomes larger than the cost of `bootstrap`, inserting additional `bootstrap` operations can sometimes reduce the overall latency by lowering the ciphertext levels used in subsequent intermediate computations. This observation shows that reducing the number of `bootstrap` operations alone does not necessarily minimize end-to-end latency.

Table 8.3: Compile time and bootstrapping management time by FORTE on the RNS-CKKS scheme (sec).

Model	ResNet-20		ResNet-44		AlexNet	
	ReLU	SiLU	ReLU	SiLU	ReLU	SiLU
# Ops	8784	9144	19412	20204	53533	50535
# Candidates	138	100	306	220	662	95
Compile Time (s)	33.0	31.0	116.5	109.2	1163.8	454.0
Bootstrap Mgmt. Time (s)	15.8	14.4	79.4	72.8	1042.3	336.5
Model	VGG16		SqueezeNet		MobileNet	
	ReLU	SiLU	ReLU	SiLU	ReLU	SiLU
# Ops	45798	44766	21404	19264	44919	45411
# Candidates	166	71	251	52	191	136
Compile Time (s)	332.7	290.2	131.2	84.8	302.3	302.1
Bootstrap Mgmt. Time (s)	230.1	188.1	89.1	44.1	222.8	218.0

8.1.4 Compile Time

Table 8.3 summarizes, for each benchmark, the total compilation time and bootstrapping-management time, together with the number of operations and `bootstrap` candidates. Benchmarks with ReLU produce a larger number of `bootstrap` candidates, since the composite polynomial used to approximate ReLU creates multiple program points with a single live-out variable. For all benchmarks except AlexNet, bootstrapping management completes within 5 minutes, which is acceptable for practical use. The remainder of this section examines the scalability of compilation time in greater detail. Within bootstrapping management, the two dominant costs arise from bypass-edge detection and bootstrapping-plan selection. The bypass-edge detection algorithm computes coverage by applying scale management to each operation, resulting in time complexity

$O(num\_op^2)$, where num_op is the number of operations. By comparison, Algorithm 5.2 runs in $O(num\_cand \times d \times num\_op)$ time, where num_cand denotes the number of `bootstrap` candidates and d denotes the maximum number of candidates covered by a given candidate.

The effect of operation count on bypass-edge detection can be seen by comparing ResNet-20(S) and AlexNet(S): the ratio $(50535/9144)^2 = 30.54$ is close to the observed compile-time ratio of $336.5/14.4 = 23.37$. The scale-management unit [32] further improves scalability by reducing the effective number of operations. The effect of candidate count on plan selection can be observed by comparing AlexNet(R) and AlexNet(S). In AlexNet(R), the intermediate feature maps cannot be packed into a single ciphertext, requiring a higher candidate-filtering SetNum than the other benchmarks and substantially increasing the candidate count. Developing a context-sensitive filtering SetNum may further improve the scalability of the bootstrapping-plan selection algorithm.

8.1.5 Performance Estimation

Figure 8.3 shows the accuracy of the latency model used in Algorithm 5.2, comparing estimated and measured latency across 108 configurations from 12 benchmarks and 9 waterlines. The regression model $y = 1.034x$ relates estimated to actual latency, and the R^2 value of 0.9986 indicates that the estimated latency closely tracks the actual latency. The small deviations in the regression coefficient and R^2 are attributable to memory transfer latency introduced by Unified Virtual Memory (UVM). The evaluation GPU provides 24 GB of physical RAM, and some benchmarks exceed this capacity, necessitating UVM to satisfy memory requirements and thereby introducing additional

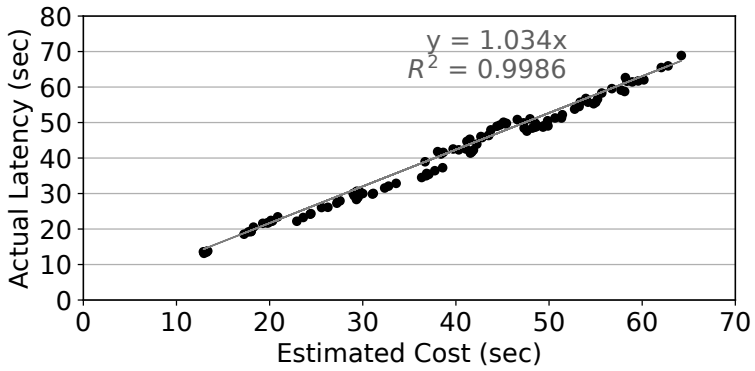
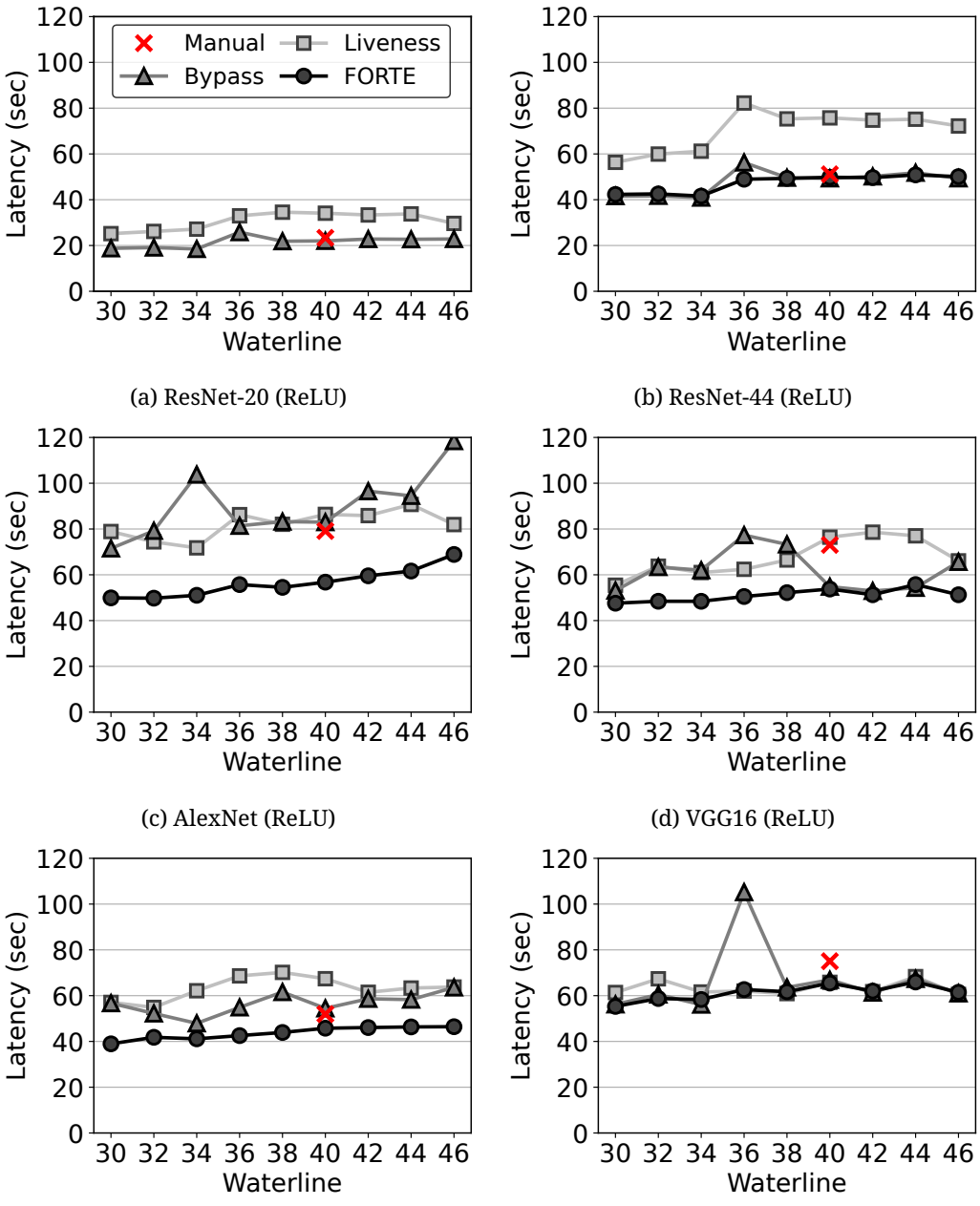


Figure 8.3: Latency estimation accuracy. The x-axis represents the estimates produced from profiled cost, and the y-axis reports the corresponding measured latency.

execution latency beyond the estimated time. The selected bootstrapping plan remains unchanged because the memory-starvation penalty is consistent across all plans for the same benchmark.

8.1.6 Waterline Sensitivity

Figures 8.4 and 8.5 show the inference latency of deep learning benchmarks across varying waterlines. The activation function approximation introduces a maximum error of 13 bits. The evaluation uses waterlines above 2^{30} to keep RNS-CKKS numerical error comparable to activation-function approximation error. The latency of Manual corresponds to a single point at a waterline 2^{40}. A lower waterline leads scale management to slow the accumulation of scale, thereby changing the optimal `bootstrap` positions. The greedy bootstrapping placement approaches (Liveness and Bypass) identify reasonable positions for ResNet-20 and SqueezeNet, but show substantial latency variance for AlexNet and MobileNet, since the `bootstrap` positions significantly affects the latency



(a) ResNet-20 (ReLU)

(b) ResNet-44 (ReLU)

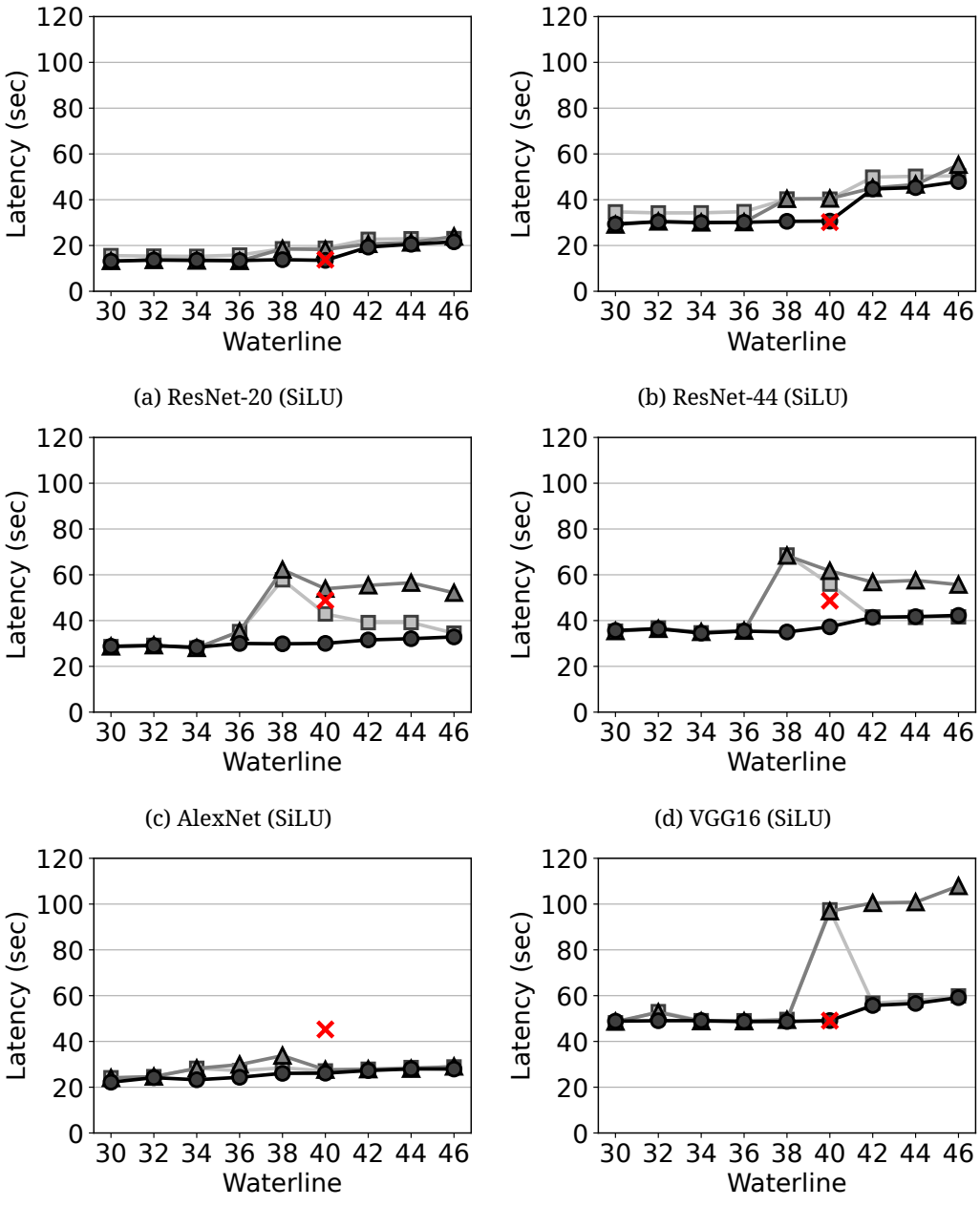
(c) AlexNet (ReLU)

(d) VGG16 (ReLU)

(e) SqueezeNet (ReLU)

(f) MobileNet (ReLU)

Figure 8.4: Inference latency of ReLU-based benchmarks with respect to varying waterlines (lower is better). Manual results are represented by a single point at waterline 2^{40}.



(a) ResNet-20 (SiLU)

(b) ResNet-44 (SiLU)

(c) AlexNet (SiLU)

(d) VGG16 (SiLU)

(e) SqueezeNet (SiLU)

(f) MobileNet (SiLU)

Figure 8.5: Inference latency of SiLU-based benchmarks with respect to varying waterlines (lower is better). Manual results are represented by a single point at waterline 2^{40}.

of arithmetic operations.

By contrast, FORTE exhibits lower latency variation across waterlines and shows a consistent decreasing trend in total latency as the waterline decreases. As shown in figure 8.4f, a small variance in latency is observed across waterlines for MobileNet(R). This variance originates from the proactive rescaling strategy of Hecate [32], which does not consistently minimize arithmetic operation latency and may produce suboptimal scale management plans. Hecate performs space exploration to identify optimal rescaling points but requires substantially longer compilation time for large benchmarks. Further refinement of the scale management scheme may reduce this residual variance.

8.2 Dynamic Loop Count Applications

8.2.1 Evaluation Setup

FORTE is implemented on top of the MLIR [71] framework, with hardware interfaces provided by the HEaaN library [11] for GPU execution. All experiments are conducted on an Intel Core i7-12700 CPU and an NVIDIA GeForce RTX A6000 GPU with 48 GB of memory. The evaluation compares programs generated by five bootstrapping management compilers:

- **FORTE w/o loop support** applies the state-of-the-art automatic bootstrapping management after fully unrolling the loops.
- **Type-matched** uses the loop structure that only matches types w/o any optimization.
- **Packing** applies only packing optimization to the Type-matched loop.
- **Packing+Unrolling** applies additional unrolling optimization to Packing.

Table 8.4: Characteristics of the benchmarks used in FORTE. Max and Min RMSE represent each benchmark's maximum and minimum root mean square error (RMSE) value, calculated by comparing the encrypted and non-encrypted results.

Benchmark	Loop Depth	# of Loop-carried Vars.	Approx. Functions	RMSE	
				Max	Min
Linear	1	2	-	4.14E-6	4.01E-6
Polynomial	1	3	-	5.59E-6	4.91E-6
Multivariate	1	9	-	3.43E-5	1.88E-5
Logistic	1	1	sigmoid	1.52E-4	9.37E-6
K-means	1	2	sign	9.13E-3	5.40E-3
SVM	1	3	sign	4.41E-3	1.59E-4
PCA	2	1, 1	sqrt	1.16E-4	2.16E-6

- **FORTE (Packing+Unrolling+Level Tuning)** applies all optimizations including packing, unrolling, and level tuning to Type-matched loop.

FORTE is evaluated on the seven benchmarks represented in Table 8.4, all of which share the common characteristic of iteratively computing values in loops.

- **Linear Regression (Linear)** is a statistical method that models the linear relationship between variables by fitting a linear equation to observed data.
- **Polynomial Regression (Polynomial)** models a polynomial relationship between the dependent and independent variables, allowing for complex patterns.
- **Multivariate Regression (Multivariate)** captures relationships among sets of independent variables and their response values.
- **Logistic Regression (Logistic)** is a statistical model for predicting the probability of a binary outcome with one or more predictor variables.
- **K-means** is an unsupervised clustering algorithm that divides a dataset into K groups by minimizing the within-cluster sum of squares.

- **Support Vector Machine (SVM)** finds the optimal hyperplane that maximizes the margin between different classes for classification and regression tasks.
- **Principal Component Analysis (PCA)** is a dimensionality reduction technique that simplifies a dataset into a set of orthogonal components.

The seven benchmarks consist of six flat loops (depth 1) and one nested loop (depth 2). The regression benchmarks (Linear, Polynomial, Multivariate, Logistic) exhibit an RMSE of up to 10^{-4}, while the remaining benchmarks (K-means, SVM, PCA) exhibit larger errors of up to 10^{-3}, due to the approximation of non-linear functions. Non-linear functions are implemented based on the algorithm in [3]. The sigmoid and sign function use the same minimax composite approximation applied to ReLU in the Section 8.1, with only the approximation range adjusted. The square root (`sqrt`) function iteratively approximates the square root of the input. The `sqrt` function therefore introduces an inner loop within the outer loop of PCA.

The Linear, Polynomial, and Multivariate regression benchmarks use 4096 randomly generated inputs for each independent variable. For the SVM and K-means benchmarks, randomly generated clusters are used as inputs. PCA uses the Iris dataset [72], and logistic regression uses the Breast Cancer dataset [73].

The deep learning applications such as ResNet typically involve loops with fixed iteration counts, which FORTE handles by full unrolling; such applications do not effectively demonstrate FORTE's support for dynamic iteration counts. The performance gains achieved on these applications are comparable to those of fully-unrolled optimization code, and the evaluation excludes them accordingly. The polynomial modulus de-

Table 8.5: Bootstrapping count of benchmarks with varying compilers. The number of bootstrapping operations is based on 40 iterations.

Benchmark	FORTE w/o Loop Support	Type-matched	Packing	Packing+ Unrolling	FORTE
Linear	18	78	39	20	20
Polynomial	39	117	39	20	20
Multivariate	81	351	39	20	20
Logistic	40	79	79	79	79
K-means	160	200	200	200	200
SVM	273	240	160	160	160

gree is $N = 2^{17}$, enabling each ciphertext to contain $N/2 = 65536$ elements. The coefficient modulus is composed of prime factors with a uniform scaling factor R_f of 51 bits, yielding a total modulus of 2^{1479}.

8.2.2 Latency Comparison

This section analyzes the end-to-end latency of programs generated by the five compilers. Figure 8.6 shows the latency of code generated by each compiler. FORTE achieves lower latency than fully-unrolled code in all benchmarks except K-means, where fully-unrolled optimization exhibits a performance advantage of up to 12.4% as shown in Figure 8.6e. Fully-unrolled Opt achieves lower latency on K-means because K-means does not execute the same code at every iteration, allowing full unrolling to adapt bootstrapping placement to the exact level consumption of each iteration. Although full unrolling provides optimization flexibility in specific cases such as K-means, Fully-unrolled Opt produces suboptimal performance on most benchmarks due to the scalability challenges of identifying bootstrapping positions across a fully unrolled program.

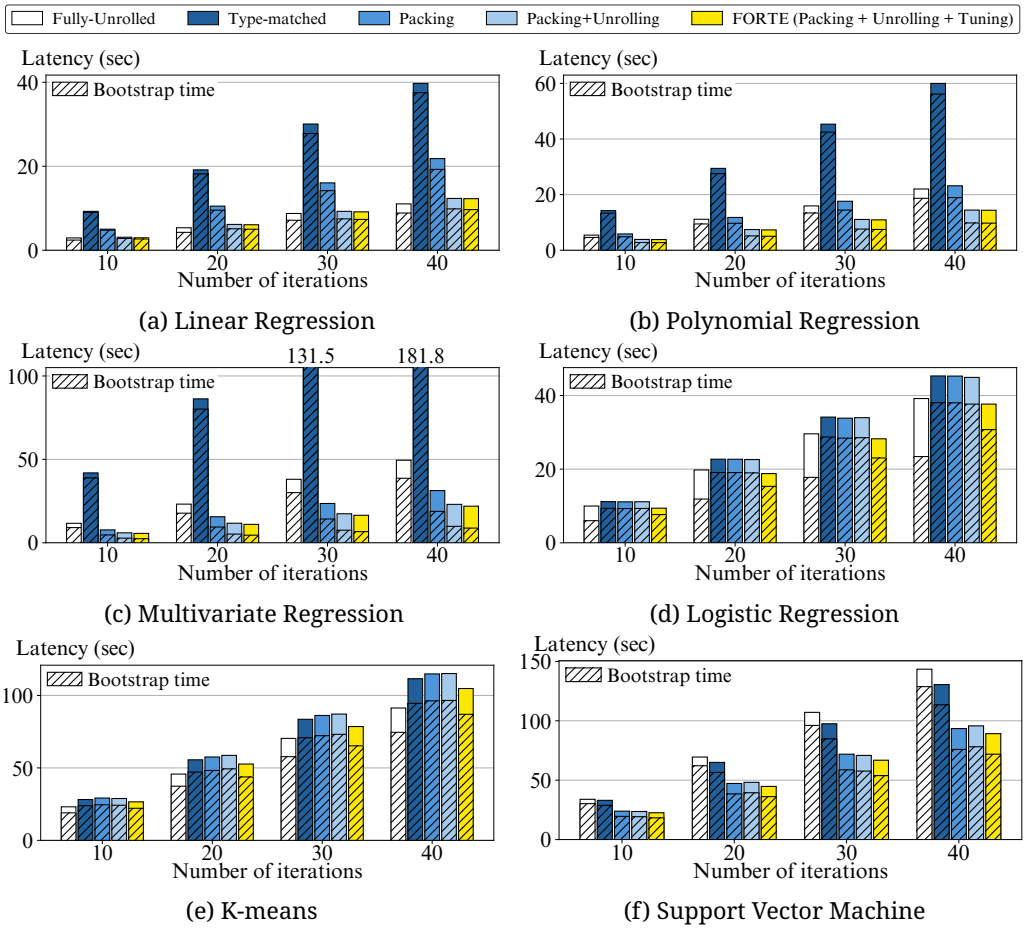


Figure 8.6: End-to-end latency comparison of six flat-loop benchmarks with five compilers. The diagonal pattern indicates the bootstrapping time.

On average, FORTE achieves a performance improvement of 27% over Fully-unrolled Code [57]. Fully-unrolled Code operates on fully unrolled code, which exposes an excessive number of bootstrapping candidate points. To manage this, Fully-unrolled Code filters candidates by the number of bootstrapping operations required at each point (equivalent to the number of live-in variables at that point) and selects insertion points from the filtered set. This heuristic-based filtering may exclude superior solutions, leading to suboptimal performance. Figure 8.6c shows the most significant improvement, with FORTE achieving a speedup of up to 2.18× on the Multivariate benchmark.

Table 8.5 compares the effect of each of the three optimizations. First, packing is effective when the loop carries two or more ciphertexts. Comparing the bootstrapping counts of Type-matched and Packing confirms a reduction in bootstrapping operations for benchmarks with more than two loop-carried variables. In K-means, packing has no effect because the $depth_{limit}$ within the loop body is insufficient; the loop body requires more levels than available, resulting in one additional `bootstrap` operation per iteration with a small overhead. Target level tuning mitigates this additional bootstrapping overhead. Second, level-aware loop unrolling is effective for regression benchmarks that contain short loop bodies. Comparing the bootstrapping counts of Packing and Packing+Unrolling confirms that unrolling reduces the bootstrapping count for Linear, Polynomial, and Multivariate. Third, target level tuning is effective for benchmarks with long multiplicative depths that require multiple `bootstrap` operations within the loop body. Comparing the bootstrapping counts of Packing+Unrolling and FORTE reveals no difference, but Figure 8.6 shows a reduction in bootstrapping latency attributable to target level tuning. Benchmarks with long multiplicative depths (Logistic, K-means, SVM)

Table 8.6: Compile time (s) of each benchmark. The numbers below FORTE (fully unrolled) represent the number of iterations used in the benchmark.

Benchmark	**FORTE w/o loop support [57]**				**FORTE**	**Improvement** (for iter 40)
	10	20	30	40		
Linear	0.13	0.34	0.60	0.94	0.041	23.09x
Polynomial	0.21	0.49	0.91	1.40	0.067	20.79x
Multivariate	13.6	24.6	37.6	54.1	0.301	179.6x
Logistic	5.51	19.9	45.7	80.1	0.360	222.7x
K-means	9.80	38.0	85.3	152	0.270	562.6x
SVM	95.0	314	675	1172	0.151	7743x

include multiple bootstrapping operations within the loop body; when a `bootstrap` operation occurs near the end of the loop body, recovering levels beyond those consumed by subsequent operations wastes computational resources. Target level tuning alone yields up to a 19% performance improvement on the Logistic benchmark.

8.2.3 Compile Time

This section evaluates the compilation time of FORTE and Fully-unrolled Code [57] across six flat-loop benchmarks with varying iteration counts. Table 8.6 presents the compilation time for each benchmark in seconds. Fully-unrolled Code fully unrolls loops prior to analysis, and the compilation time increases substantially as the iteration count grows, reaching a factor of $15.51\times$ from 10 to 40 iterations on the K-means benchmark. Because Fully-unrolled Code does not support dynamic iteration counts, Fully-unrolled Code processes an increasingly large unrolled program as the iteration count rises, resulting in a substantial increase in compilation time. The quadratic growth in K-means compilation time is attributable to the exploration-based analysis employed by Fully-

Table 8.7: Code size (KB) of each benchmark. The numbers below FORTE (fully unrolled) represent the number of iterations used in the benchmark.

Benchmark	**FORTE w/o loop support [57]**				**FORTE**	**Improvement** (for iter 40)
	10	20	30	40		
Linear	8.37	16.4	24.5	32.3	5.946	5.438x
Polynomial	13.2	25.6	37.8	50.4	9.290	5.429x
Multivariate	36.2	70.9	106	140	32.08	4.376x
Logistic	67.8	131	195	259	17.20	15.04x
K-means	122	242	362	483	15.49	31.17x
SVM	118	235	352	470	16.09	29.19x

unrolled Code. By contrast, FORTE is loop-aware and supports dynamic iteration counts, maintaining constant compilation time regardless of the number of iterations. FORTE achieves a geometric mean compilation time reduction of 209.12× over Fully-unrolled Code. In the most demanding case, FORTE reduces compilation time by 7743× relative to Fully-unrolled Code, demonstrating the efficiency and scalability of the loop-aware compilation approach.

8.2.4 Code Size Reduction

This section evaluates the code size produced by FORTE and Fully-unrolled Code [57] for six flat-loop benchmarks with varying iteration counts. Table 8.7 summarizes the code size of each benchmark in kilobytes (KB), including constant data. Because Fully-unrolled Code does not support dynamic iteration counts, the generated code size increases up to 3.98× as the iteration count grows from 10 to 40, as observed in the SVM benchmark. The code size scales linearly with the iteration count, imposing substantial memory and storage overhead for programs with large iteration counts.

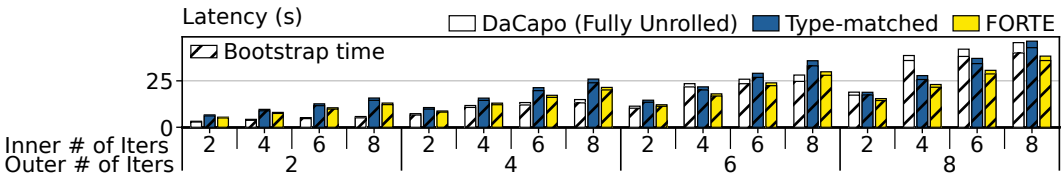


Figure 8.7: Latency of Principal Component Analysis (PCA) with varying inner and outer iteration counts.

By contrast, FORTE supports dynamic iteration counts and generates code of constant size regardless of the number of iterations. FORTE achieves a geometric mean code size reduction of 11.0× over Fully-unrolled Code at 40 iterations, demonstrating the scalability advantage of loop-aware compilation for FHE applications.

8.2.5 Case Study: PCA Nested-Loop

This section presents a case study on the PCA benchmark, which contains a nested loop, to demonstrate that FORTE handles complex loop structures. PCA consists of a depth-2 nested loop with one loop-carried variable in each of the inner and outer loops. Each loop body has a long multiplicative depth, so level-aware loop unrolling does not apply. Consequently, FORTE applies only the target level tuning optimization to the PCA benchmark.

Figure 8.7 compares the latency of the compiled programs across varying inner and outer iteration counts. Type-matched and FORTE generate identical loop bodies for all configurations, so the latency of both is proportional to the iteration count. Fully-unrolled Code [57] does not produce consistent results across configurations. Although Fully-unrolled Code fully unrolls the loop for each configuration, thereby exposing additional optimization opportunities, Fully-unrolled Code fails to identify an efficient boot-

Table 8.8: Bootstrapping count for the PCA benchmark. Column headers indicate the number of outer and inner loop iterations for Fully-unrolled Code [57] (fully unrolled), Type-matched, and FORTE.

Outer # of Iters	2		4		6		8	
Inner # of Iters	2	8	2	8	2	8	2	8
FORTE w/o loop support	5	10	13	27	21	52	36	85
Type-matched	12	30	20	50	28	70	36	90
FORTE	12	30	20	50	28	70	36	90

strapping plan for large iteration counts, such as (outer, inner) = (8, 4). FORTE achieves iteration-proportional latency for configurations (8, 2) and (8, 4), whereas Fully-unrolled Code incurs substantial overhead at (8, 4). Because full unrolling exposes an excessive number of bootstrapping candidates, Fully-unrolled Code filters the candidate set and selects the most efficient plan from the filtered list; this filtering step eliminates superior solutions in cases such as (8, 4). By contrast, FORTE optimizes the loop body as a repeated code segment rather than the full unrolled trace, allowing the optimization to scale to large iteration counts.

Table 8.8 reports the bootstrapping count for each combination of inner and outer iteration counts. When both the inner and outer loops iterate 8 times, FORTE reduces the code size by $13.66\times$ and reduces the compilation time by $146.75\times$ compared to Fully-unrolled Code [57].

8.3 Case Study: Comparison with Orion

This section compares FORTE with Orion [54], a recent FHE compiler that optimizes bootstrapping placement for deep learning applications. Orion [54] formulates bootstrapping placement as a shortest-path problem over a layer-level directed acyclic graph.

Table 8.9: Latency and bootstrapping comparison between Orion [54] and FORTE.

Benchmark	Total Latency (s)		Boot. Time (s)		# of Bootstrap		Avg Boot. (ms)	
	Orion	FORTE	Orion	FORTE	Orion	FORTE	Orion	FORTE
AlexNet	8.2402	7.5146	2.8425	1.7672	9	6	315.83	294.54
ResNet	8.6138	8.5144	6.2479	5.7955	19	18	328.84	321.97
VGG16	13.0781	11.5331	4.7680	3.9058	15	13	317.87	300.45

Instead of considering arbitrary program points, Orion restricts bootstrapping candidates to layer boundaries and does not insert bootstrapping operations inside linear layers or polynomial evaluations. Each node represents executing a layer at a feasible ciphertext level, while edges model level transitions and include bootstrapping costs when a ciphertext refresh is required. The minimum-cost path therefore jointly determines the execution level of each layer and the locations of bootstrapping operations. For networks with residual connections, Orion decomposes the graph into single-entry single-exit regions and solves the corresponding subgraphs before integrating them into the global placement problem.

In comparison, FORTE optimizes bootstrapping placement at the FHE primitive operation level (*i.e.,* add, multiply, rotate, bootstrapping), allowing bootstrapping operations to be placed at arbitrary program points, including within layers. As a result, FORTE can identify more fine-grained bootstrapping points and better account for the latency impact of individual FHE operations. To evaluate this trade-off, this section compares FORTE against Orion under the same execution environment. The comparison is conducted on an NVIDIA RTX A6000 GPU with 48 GB of memory, using HEonGPU as the execution backend. The evaluation uses AlexNet, ResNet, and VGG16, which are

the benchmarks reported by Orion. Table 8.9 compares the total latency, showing that FORTE achieves speedups of 1.097× on AlexNet, 1.012× on ResNet, and up to 1.134× on VGG16 over Orion.

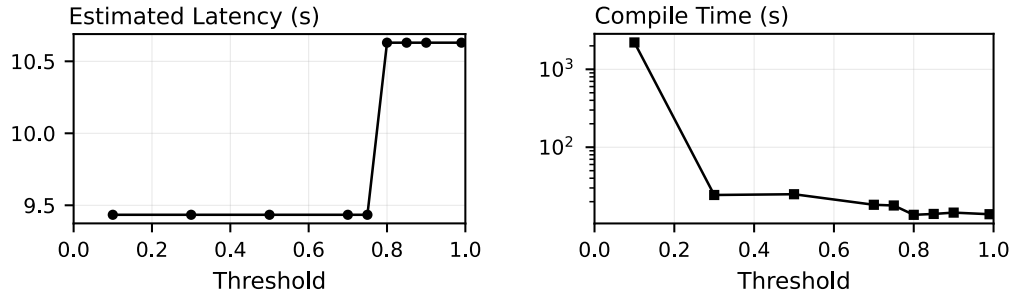
Table 8.9 also shows the source of this improvement by comparing the bootstrapping behavior of the two systems. For all three applications, FORTE requires fewer bootstrapping operations than Orion. This reduction comes from FORTE's primitive-level placement, which can insert `bootstrap` operations inside layers rather than only at layer boundaries, allowing FORTE to better utilize the available ciphertext levels before each refresh. The table also shows that FORTE reduces the average latency of individual `bootstrap` operations, which reflects the effect of target-level tuning. To further evaluate the effectiveness of target-level tuning, Table 8.9 compares the average latency of a single `bootstrap` operation between Orion and FORTE. The average bootstrapping latency is reduced by 6.74% on AlexNet and 2.09% on ResNet compared to Orion. This result shows that FORTE avoids the overhead incurred by Orion's fixed maximum-level recovery, by recovering only the levels required by the subsequent computation.

9. Discussion

9.1 Optimality of Bootstrapping Placement

In general, bootstrapping placement is known to be an NP-hard optimization problem. To reduce this large search space, FORTE avoids exhaustively considering all possible placements in the full program. Instead, FORTE first identifies a set of candidate insertion points and then optimizes the placement plan within that candidate set. The candidate selection step combines liveness analysis with bypass-edge analysis. Liveness analysis identifies program points where fewer live-out ciphertexts need to be refreshed, while bypass-edge analysis avoids counting long-lived ciphertexts that remain live but do not consume the scale budget during the bypassed region. This bypass-edge analysis is heuristic and depends on the user-defined threshold S_{th}. In bypass-edge analysis, FORTE records the operation identifier *opid* of a program point whose accumulated scale satisfies $AS(opid) \geq S_{\text{th}} \cdot S_{\text{max}}$, where S_{max} is the maximum scale budget and $S_{\text{th}} \in [0, 1]$ is the bypass threshold. The bypass threshold S_{th} controls the size of the candidate set and the aggressiveness of bypass-edge detection. A smaller S_{th} may expose lower-latency plans by adding more candidates, but it can increase compilation time. A larger S_{th} reduces the search space but may miss useful placement opportunities.

Figure 9.1 presents a sensitivity study of the bypass threshold S_{th} on the ResNet



(a) Estimated latency across bypass thresholds. (b) Compile time across bypass thresholds.

Figure 9.1: Sensitivity study of FORTE to the bypass threshold S_{th} on ResNet.

benchmark. The latency reported in Figure 9.1a is the estimated latency used during compile-time planning; as shown in Section 8.1.5, this estimate is highly proportional to the actual execution time and therefore serves as a reliable proxy for plan quality. From the estimated-latency trend, when $S_{\text{th}} \geq 0.75$, FORTE relies mostly on liveness-based candidates and excludes fewer bypass edges, which can cause the planner to miss better placement opportunities and choose slower plans than those obtained with smaller thresholds. By contrast, once S_{th} decreases below approximately 0.75, the estimated latency becomes nearly unchanged, indicating that the selected plan has effectively saturated and that further expansion of bypass candidates does not materially improve plan quality.

From the compilation-time perspective, however, the trend is markedly different. As S_{th} approaches 0, the compilation time increases sharply because more candidate points and more candidate plans must be explored, substantially enlarging the search space. Notably, these smaller thresholds still lead to essentially the same plan as the one obtained at $S_{\text{th}} = 0.5$. Based on this tradeoff, FORTE heuristically adopts $S_{\text{th}} = 0.5$, which

enables the compiler to find a strong bootstrapping plan while keeping compile time practical. Based on the above observations, this paragraph now discusses the optimality of FORTE. FORTE does not guarantee global optimality of the final bootstrapping plan. The first reason is that the candidate selector itself is heuristic. If the globally optimal insertion point is excluded during liveness- and bypass-based candidate pruning, then the globally optimal plan cannot be considered by the subsequent planner. The bypass threshold S_{th} directly affects this behavior by trading off search-space size against the risk of excluding useful program points.

Even if the analysis is restricted to the selected candidate set, however, the bootstrapping planner still does not solve the full optimal planning problem over that set. First, the planner performs dynamic programming under the assumption that every `bootstrap` restores the ciphertext to the maximum target level. In practice, however, FORTE later applies target-level tuning and may choose a lower target level for a given `bootstrap`. If target levels were optimized jointly with placement during planning, a different plan could become preferable. Second, the dynamic programming state used by FORTE stores only the minimum latency up to a candidate point, rather than the richer state needed to characterize the subsequent effect of that choice. For example, the plan with the lowest latency at candidate point $i-1$ may leave bypass edges at levels that make the subsequent segment to i expensive, whereas a slightly slower partial plan at $i-1$ may preserve more favorable bypass-edge levels and therefore produce a lower total latency after extending to i. Because the planner does not retain multiple competing states with different bypass-edge level configurations, it can discard partial plans that are suboptimal locally but superior globally.

For these reasons, the current FORTE bootstrapping planner should be understood as a compile-time-efficient heuristic planner rather than an exact optimizer. The design intentionally prioritizes scalable and practical plan search over exhaustive optimality, enabling the compiler to find good bootstrapping plans with substantially lower compilation overhead.

9.2 Limitation: Static Shape Assumption

FORTE assumes that input shapes are known and bounded at compile time, so that the compiler can statically construct a fixed operation graph and analyze its level and scale evolution. This design limits FORTE to workloads whose multiplicative-depth structure is invariant across executions. For example, in neural-network workloads, changes in input shape, channel count, or feature-map size may alter intermediate control flow, change the number of intermediate ciphertexts, or add and remove homomorphic operations at runtime. These changes can introduce or eliminate multiplicative-depth increments, making a statically chosen bootstrapping placement invalid or suboptimal for some executions. Supporting these workloads would require shape-dependent depth analysis, runtime-dependent ciphertext allocation, and either recomputing bootstrapping placements or generating multiple shape-specialized plans. It would also require a rule-based runtime bootstrapping management mechanism that can adapt placement decisions to execution-specific depth and level consumption.

9.3 Scalability and Practicality

A central limitation of prior RNS-CKKS compilers is the requirement to fully unroll loops before applying scale management, which causes compilation time and code size to grow proportionally with the iteration count and necessitates recompilation whenever the iteration count changes. FORTE eliminates this limitation by supporting loops with both static and dynamic iteration counts without full unrolling. A single FORTE-compiled binary handles varying iteration counts at runtime, removing the need to maintain multiple iteration-specific builds and simplifying deployment workflows for loop-heavy workloads such as iterative regression, clustering, and nested-loop routines. Through the packing, level-aware unrolling, and target-level tuning optimizations, FORTE achieves an average performance improvement of 27% over DaCapo [57], which relies on full unrolling, demonstrating that the loop abstraction approach incurs no performance regression relative to unrolling-based compilation.

Beyond scalability, FORTE substantially reduces the engineering burden of FHE program development. Manual bootstrapping placement requires approximately four hours of analysis, insertion, and verification per benchmark per waterline configuration, as an incorrect placement causes ciphertext scale overflow and requires repeated debugging. FORTE automates this process and compiles each benchmark in 30 seconds to 16 minutes, enabling FHE application developers to iterate rapidly without deep expertise in scale management. FORTE is made available as open-source software to further lower the barrier to adoption for researchers and programmers.

9.4 Long-Term Impact

The techniques introduced in this dissertation carry implications that extend beyond the immediate scope of the FORTE compiler. Many privacy-sensitive industries (*e.g.,* healthcare, finance, and cloud service providers) currently hesitate to adopt outsourced computation due to the risk of exposing confidential data to third-party servers. FORTE advances the feasibility of deploying FHE-based privacy-preserving computation in such settings by making the compilation process practical for a broader class of iterative programs. In particular, the increasing prevalence of large-scale machine learning workloads, including large language models, creates a pressing need for FHE compilers that scale beyond static operator graphs. While current GPU memory constraints limit the direct application of FORTE to the largest models, the automatic bootstrapping management framework established in this dissertation provides a foundation for future compiler infrastructure targeting emerging FHE hardware accelerators with larger memory capacities. As FHE hardware continues to mature and FHE libraries expand their algorithmic coverage, the compiler-level techniques proposed in this dissertation will play an increasingly central role in bridging the gap between high-level program specifications and efficient, secure execution on encrypted data, ultimately enabling privacy-preserving computation to become a practical component of production data pipelines.

10. Conclusion

This dissertation presents FORTE, the FHE compiler that automatically manages bootstrappings for programs containing both static and dynamic loop structures. The key motivation behind FORTE is that bootstrapping, which refreshes noise to support long FHE computations, is highly expensive, and its placement has a critical impact on end-to-end latency. Manual bootstrapping placement demands significant engineering effort, is error-prone, and must be repeated for each new application or parameter configuration. Prior automatic bootstrapping compilers address only static computational graphs, requiring full loop unrolling before any scale management and thereby rendering these compilers impractical for iterative workloads with dynamic iteration counts. FORTE resolves both the correctness and efficiency challenges of bootstrapping management in a unified compiler framework.

To support FHE programs containing loops with dynamic iteration counts, FORTE establishes formal requirements for loop-carried variable type consistency and introduces two code transformations to satisfy these requirements. FORTE resolves encryption status mismatches on loop-carried variables by applying loop peeling, and level mismatches on loop-carried variables by applying `modswitch` and `bootstrap` to align ciphertext levels across iterations. Following type alignment, FORTE abstracts the loop structure as a black-box operation within a static computational graph, enabling the

subsequent bootstrapping placement module to process loop-containing programs through a unified algorithm identical to that applied to loop-free programs.

FORTE employs a bootstrapping candidate analysis that combines ciphertext liveness analysis with bypass-edge analysis to identify favorable program points for bootstrapping placement. Building on the candidate analysis, FORTE uses a latency-aware bootstrapping placement algorithm that selects the placement plan with the minimum end-to-end latency from the candidate set by using dynamic programming. Consequently, minimizing the number of bootstrapping operations does not always minimize end-to-end latency.

To further reduce the overhead of `bootstrap` operations in loop-containing programs, FORTE introduces three additional optimizations. Packing loop-carried ciphertexts reduces the number of bootstrapping operations per iteration by consolidating multiple loop-carried ciphertexts into a single ciphertext and applying a single bootstrapping, rather than bootstrapping each loop-carried ciphertext individually. The level-aware loop unrolling optimization eliminates wasted levels by unrolling the loop body to fully consume the levels recovered by each bootstrapping, avoiding the levels that would otherwise be discarded by `modswitch` before the next iteration. The bootstrapping target-level tuning reduces the latency of each individual `bootstrap` operation by setting the target level to the exact level consumed by subsequent operations, rather than unconditionally restoring ciphertext levels to the maximum.

FORTE achieves a geometric mean speedup of 1.26× over manually placed bootstrapping across all 12 benchmarks, while the classification accuracy of FORTE-generated programs matches the PyTorch baseline within 0.1% for all evaluated models. For ma-

chine learning programs with dynamic loop counts, FORTE is evaluated on seven benchmarks spanning linear regression, polynomial regression, multivariate regression, logistic regression, K-means clustering, support vector machine, and principal component analysis. FORTE achieves an average performance improvement of 27% over fully-unrolled code, the state-of-the-art bootstrapping placement compiler that requires full loop unrolling, with a maximum improvement of 2.18× on the multivariate regression benchmark.

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] V. Lyubashevsky, C. Peikert, and O. Regev, “On ideal lattices and learning with errors over rings,” in *Advances in Cryptology: 29th Annual International Conference on the Theory and Applications of Cryptographic Techniques*, H. Gilbert, Ed., Berlin, Heidelberg: Springer, 2010.

[3] E. Lee, J.-W. Lee, J.-S. No, and Y.-S. Kim, “Minimax approximation of sign function by composite polynomial for homomorphic comparison,” *IEEE Transactions on Dependable and Secure Computing*, vol. 19, no. 6, pp. 3711–3727, 2021.

[4] R. Agrawal, J. H. Ahn, F. Bergamaschi, R. Cammarota, J. H. Cheon, F. D. M. de Souza, H. Gong, M. Kang, D. Kim, J. Kim, H. de Lassus, J. H. Park, M. Steiner, and W. Wang, “High-precision rns-ckks on fixed but smaller word-size architectures: Theory and application,” in *Proceedings of the 11th Workshop on Encrypted Computing & Applied Homomorphic Cryptography*, ser. Workshop on Encrypted Computing and Applied Homomorphic Cryptography 2023, Association for Computing Machinery, 2023.

[5] 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*, C. Cid and M. J. Jacobson Jr., Eds., Springer International Publishing, 2018.

[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*, London, UK: ACM, 2020, ISBN: 978-1-4503-7613-6.

[7] J. Fan and F. Vercauteren, *Somewhat practical fully homomorphic encryption*, Cryptology ePrint Archive, Report 2012/144, `https://eprint.iacr.org/2012/144`, 2012.

[8] J. H. Cheon, A. Kim, M. Kim, and Y. Song, “Homomorphic encryption for arithmetic of approximate numbers,” in *Advances in Cryptology: 23rd International Conference on the Theory and Applications of Cryptology and Information Security*, Springer, 2017, pp. 409–437.

[9] I. Chillotti, N. Gama, M. Georgieva, and M. Izabachène, “Tfhe: Fast fully homomorphic encryption over the torus,” *Journal of Cryptology*, no. 1, pp. 34–91, 2020, `https://doi.org/10.1007/s00145-019-09319-x`.

[10] *Lattigo v4*, Online: https://github.com/tuneinsight/lattigo, EPFL-LDS, Tune Insight SA, Aug. 2022.

[11] *HEAAN Open-Source HE Library*, https://github.com/snucrypto/HEAAN, 2020.

[12] *PALISADE Lattice Cryptography Library*, https://palisade-crypto.org/, Oct. 2020.

[13] W. Dai and B. Sunar, "Cuhe: A homomorphic encryption accelerator library," in *Cryptography and Information Security in the Balkans*, E. Pasalic and L. R. Knudsen, Eds., Cham: Springer International Publishing, 2016, pp. 169–186, ISBN: 978-3-319-29172-7.

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

[15] 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.

[16] *HElib Open-Source HE Library*, https://github.com/homenc/HElib, 2020.

[17] *FullRNS-HEAAN*, https://github.com/KyoohyungHan/FullRNS-HEAAN, 2018.

[18] I. Chillotti, N. Gama, M. Georgieva, and M. Izabachène, *TFHE: Fast fully homomorphic encryption library*, https://tfhe.github.io/tfhe/, Aug. 2016.

[19] L. Ducas and D. Micciancio, "Fhew: Bootstrapping homomorphic encryption in less than a second," in *Annual international conference on the theory and applications of cryptographic techniques*, Springer, 2015.

[20] Zama, *TFHE-rs: A Pure Rust Implementation of the TFHE Scheme for Boolean and Integer Arithmetics Over Encrypted Data*, https://github.com/zama-ai/tfhe-rs, 2022.

[21] A. Ş. Özcan and E. Savaş, *HEonGPU: A GPU-based fully homomorphic encryption library 1.0*, Cryptology ePrint Archive, Paper 2024/1543, 2024.

[22] K. Han and D. Ki, "Better bootstrapping for approximate homomorphic encryption," in *Cryptographers' Track at the RSA Conference*, Springer, 2020, pp. 364–390.

[23] W. Jung, S. Kim, J. H. Ahn, J. H. Cheon, and Y. Lee, "Over 100x faster bootstrapping in fully homomorphic encryption through memory-centric optimization with GPUs," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pp. 114–148, 2021.

[24] J.-W. Lee, E. Lee, Y.-S. Kim, and J.-S. No, “Rotation key reduction for client-server systems of deep neural network on fully homomorphic encryption,” in *International Conference on the Theory and Application of Cryptology and Information Security*, Springer, 2023, pp. 36–68.

[25] J. Park, D. Kim, J. Kim, S. Kim, W. Jung, J. H. Cheon, and J. H. Ahn, *Toward practical privacy-preserving convolutional neural networks exploiting fully homomorphic encryption*, arXiv preprint arXiv:2310.16530, 2023. arXiv: 2310.16530.

[26] Y. Lee, J. Seo, Y. Nam, J. Chae, and J. H. Cheon, “Heaan-stat: A privacy-preserving statistical analysis toolkit for large-scale numerical, ordinal, and categorical data,” *IEEE Transactions on Dependable and Secure Computing*, 2024.

[27] J.-P. Bossuat, C. Mouchet, J. Troncoso-Pastoriza, and J.-P. Hubaux, “Efficient bootstrapping for approximate homomorphic encryption with non-sparse keys,” in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, Springer, 2021, pp. 587–617.

[28] W. Choi, J. Kim, and J. H. Ahn, “Cheddar: A swift fully homomorphic encryption library designed for gpu architectures,” in *Proceedings of the 31st ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, 2026, pp. 35–49.

[29] F. Boemer, Y. Lao, R. Cammarota, and C. Wierzynski, “Ngraph-he: A graph compiler for deep learning on homomorphically encrypted data,” in *Proceedings of the 16th ACM International Conference on Computing Frontiers*, 2019, pp. 3–13.

[30] F. Boemer, A. Costache, R. Cammarota, and C. Wierzynski, “Ngraph-he2: A high-throughput framework for neural network inference on encrypted data,” in *Proceedings of the 7th ACM Workshop on Encrypted Computing & Applied Homomorphic Cryptography*, 2019, pp. 45–56.

[31] 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*, (Phoenix, AZ, USA), New York, NY, USA: ACM, 2019, ISBN: 978-1-4503-6712-7.

[32] 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*, 2022.

[33] 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*, 2023, pp. 4697–4714.

[34] Y. Lee, S. Cheon, D. Kim, D. Lee, and H. Kim, “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.

[35] M. Cowan, D. Dangwal, A. Alaghi, C. Trippel, V. T. Lee, and B. Reagen, “Porcupine: A synthesizing compiler for vectorized homomorphic encryption,” in *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, 2021, pp. 375–389.

[36] A. Viand, P. Jattke, M. Haller, and A. Hithnawi, “HECO: Fully homomorphic encryption compiler,” in *32nd USENIX Security Symposium*, 2023.

[37] 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.

[38] E. Crockett, C. Peikert, and C. Sharp, “Alchemy: A language and compiler for homomorphic encryption made easy,” in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018, pp. 1020–1037.

[39] S. Bian, Z. Zhao, Z. Zhang, R. Mao, K. Suenaga, Y. Jin, Z. Guan, and J. Liu, "Heir: A unified representation for cross-scheme compilation of fully homomorphic computation," in *31st Annual Network and Distributed System Security Symposium*, 2024.

[40] S. Park, W. Song, S. Nam, H. Kim, J. Shin, and J. Lee, "HEaaN.MLIR: An optimizing compiler for fast ring-based homomorphic encryption," *Proceedings of the ACM on Programming Languages*, vol. 7, no. ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 196–220, 2023.

[41] *Cingulata*, `https://github.com/CEA-LIST/Cingulata`, 2020.

[42] E. Chielle, O. Mazonka, H. Gamil, N. G. Tsoutsos, and M. Maniatakos, *E3: A framework for compiling c++ programs with encrypted operands*, Cryptology ePrint Archive, Report 2018/1013, `https://ia.cr/2018/1013`, 2018.

[43] A. Viand and H. Shafagh, "Marble: Making fully homomorphic encryption accessible to all," in *Proceedings of the 6th Workshop on Encrypted Computing & Applied Homomorphic Cryptography*, ser. Workshop on Encrypted Computing and Applied Homomorphic Cryptography 2018, Association for Computing Machinery, 2018.

[44] S. Gorantala, R. Springer, S. Purser-Haskell, W. Lam, R. Wilson, A. Ali, E. P. Astor, I. Zukerman, S. Ruth, C. Dibak, et al., "A general purpose transpiler for fully homomorphic encryption," *arXiv preprint arXiv:2106.07893*, 2021.

[45] S. Chowdhary, W. Dai, K. Laine, and O. Saarikivi, "Eva improved: Compiler and extension library for ckks," in *Proceedings of the 9th on Workshop on Encrypted Computing & Applied Homomorphic Cryptography*, ser. Workshop on Encrypted Computing and Applied Homomorphic Cryptography 2021, New York, NY, USA: Association for Computing Machinery, 2021.

[46] Zama, *Concrete: TFHE Compiler that converts python programs into FHE equivalent*, `https://github.com/zama-ai/concrete`, 2022.

[47] T. van Elsloo, G. Patrini, and H. Ivey-Law, *Sealion: A framework for neural network inference on encrypted data*, arXiv preprint arXiv:1904.12840, 2019. arXiv: 1904.12840.

[48] D. W. Archer, J. M. Calderón Trilla, J. Dagit, A. Malozemoff, Y. Polyakov, K. Rohloff, and G. Ryan, "Ramparts: A programmer-friendly system for building homomorphic encryption applications," in *Proceedings of the 7th ACM Workshop on Encrypted Computing & Applied Homomorphic Cryptography*, ser. Workshop on Encrypted Computing and Applied Homomorphic Cryptography 2019, Association for Computing Machinery, 2019.

[49] D. Kim, Y. Lee, S. Cheon, H. Choi, J. Lee, H. Youm, D. Lee, and H. Kim, "Privacy set: Privacy-authority-aware compiler for homomorphic encryption on edge-cloud system," *IEEE Internet of Things Journal*, vol. 11, no. 21, pp. 35 167–35 184, 2024.

[50] S. Cheon, Y. Lee, H. Youm, D. Kim, S. Yun, K. Jeong, D. Lee, and H. Kim, "Halo: Loop-aware bootstrapping management for fully homomorphic encryption," in *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, ser. ASPLOS '25, Rotterdam, Netherlands: Association for Computing Machinery, 2025, pp. 572–585, ISBN: 9798400706981.

[51] A. Viand, P. Jattke, and A. Hithnawi, "Sok: Fully homomorphic encryption compilers," in *2021 IEEE Symposium on Security and Privacy*, IEEE, 2021, pp. 1092–1108.

[52] C. Marcolla, V. Sucasas, M. Manzano, R. Bassoli, F. H. P. Fitzek, and N. Aaraj, "Survey on fully homomorphic encryption, theory, and applications," *Proceedings of the IEEE*, 2022.

[53] C. Gouert, D. Mouris, and N. Tsoutsos, "Sok: New insights into fully homomorphic encryption libraries via standardized benchmarks," *Proceedings on privacy enhancing technologies*, 2023.

[54] A. Ebel, K. Garimella, and B. Reagen, "Orion: A fully homomorphic encryption framework for deep learning," in *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, 2025.

[55] Y. Liu, J. Lai, L. Li, T. Sui, L. Xiao, P. Yuan, X. Zhang, Q. Zhu, W. Chen, and J. Xue, "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*, 2025.

[56] 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. ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 126–150, 2024.

[57] 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*, 2024, pp. 6993–7010.

[58] C. Juvekar, V. Vaikuntanathan, and A. Chandrakasan, “Gazelle: A low latency framework for secure neural network inference,” in *27th USENIX Security Symposium*, 2018, pp. 1651–1669.

[59] Z. Huang, W.-j. Lu, C. Hong, and J. Ding, “Cheetah: Lean and fast secure two-party deep neural network inference,” in *31st USENIX Security Symposium*, 2022, pp. 809–826.

[60] D. Rathee, M. Rathee, N. Kumar, N. Chandran, D. Gupta, A. Rastogi, and R. Sharma, “Cryptflow2: Practical 2-party secure inference,” in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, 2020, pp. 325–342.

[61] P. Mishra, R. Lehmkuhl, A. Srinivasan, W. Zheng, and R. A. Popa, “Delphi: A cryptographic inference system for neural networks,” in *Proceedings of the 2020 Workshop on Privacy-Preserving Machine Learning in Practice*, 2020, pp. 27–30.

[62] J.-W. Lee, H. Kang, Y. Lee, W. Choi, J. Eom, M. Deryabin, E. Lee, J. Lee, D. Yoo, Y.-S. Kim, et al., “Privacy-preserving machine learning with fully homomorphic encryption for deep neural network,” *IEEE Access*, 2022.

[63] E. Lee, J.-W. Lee, J. Lee, Y.-S. Kim, Y. Kim, J.-S. No, and W. Choi, “Low-complexity deep convolutional neural networks on fully homomorphic encryption using

multiplexed parallel convolutions," in *International Conference on Machine Learning*, PMLR, 2022, pp. 12 403–12 422.

[64] D. Kim, J. Park, J. Kim, S. Kim, and J. H. Ahn, "Hyphen: A hybrid packing method and optimizations for homomorphic encryption-based neural networks," *arXiv preprint arXiv:2302.02407*, 2023.

[65] M. Paindavoine and B. Vialla, "Minimizing the number of bootstrappings in fully homomorphic encryption," in *Selected Areas in Cryptography: 22nd International Conference*, Springer, 2016, pp. 25–43.

[66] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.

[67] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," *Communications of the ACM*, vol. 60, no. 6, pp. 84–90, 2017.

[68] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.

[69] F. N. Iandola, S. Han, M. W. Moskewicz, K. Ashraf, W. J. Dally, and K. Keutzer, "SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <0.5 mb model size," *arXiv preprint arXiv:1602.07360*, 2016.

[70] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, "MobileNets: Efficient convolutional neural networks for mobile vision applications," *arXiv preprint arXiv:1704.04861*, 2017.

[71] 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*, IEEE, 2021, pp. 2–14.

[72] R. A. Fisher, "The use of multiple measurements in taxonomic problems," *Annals of eugenics*, 1936.

[73] W. Wolberg, O. Mangasarian, N. Street, and W. Street, *Breast Cancer Wisconsin (Diagnostic)*, UCI Machine Learning Repository, DOI: https://doi.org/10.24432/C5DW2B, 1995.

Abstract in Korean

효율적인 완전동형암호 연산을 위한 반복문 지원 부트스트래핑 관리

완전동형암호(Fully Homomorphic Encryption, FHE)는 복호화하지 않은 암호문 상태에서 직접 연산을 수행할 수 있게 해 주는 기술로, 클라우드와 같은 비신뢰 환경에서 프라이버시를 보장하는 기계학습을 실현할 수 있는 유망한 방법이다. 여러 FHE 방식 가운데 RNS-CKKS는 고정소수점 연산과 SIMD 병렬성을 지원하므로 기계학습 응용에 특히 적합하다. 하지만 RNS-CKKS에서는 반복적인 곱셈이 암호문의 유한한 레벨을 지속적으로 소모한다. 이 한계를 넘어 계산을 계속 수행하려면 부트스트래핑을 통해 소모된 레벨을 복원해야 하는데, 부트스트래핑은 RNS-CKKS에서 가장 비용이 큰 연산이다. 더욱이 부트스트래핑을 어디에 배치하느냐에 따라 부트스트래핑의 횟수뿐 아니라 이후의 스케일 관리 방식과 FHE 연산 비용까지 달라진다. 따라서 부트스트래핑 배치 최적화는 프로그램 전체의 성능을 함께 고려해야 하는 문제이며, 이를 수작업으로 수행하는 것은 어렵고 비효율적이다.

본 학위논문은 완전동형암호를 위한 반복문 구조 기반 부트스트래핑 관리 컴파일러 FORTE를 제안한다. FORTE 는 부트스트래핑 연산을 자동으로 삽입하고 최적화하며, 정적 제어 흐름을 갖는 프로그램과 동적 반복 횟수를 포함하는 루프 프로그램이라는 두 가지 핵심 문제를 다룬다.

정적 제어 흐름을 갖는 프로그램에 대해서는, FORTE 는 live-out 암호문을 분석하여 동시에 부트스트래핑해야 하는 암호문의 수가 적은 지점을 우선적으로 찾는 liveness-aware 후보 선택 기법을 제안한다. 여기에 더해 bypass-edge 분석을 적용하여 오랫동안 살아 있으나 실제로는 부트스트래핑이 필요하지 않은 암호문을 후보 집합에서 제외한다. 이렇게 정제된 후보들

에 대해 FORTE 는 서로 다른 스케일 관리 계획 아래에서 비용을 추정하고, 동적 계획법을 이용해 전체 지연 시간이 가장 작은 부트스트래핑 배치 계획을 선택한다. 평가 결과, 정적 루프 반복 횟수를 갖는 프로그램에서 FORTE 는 수작업으로 구현한 FHE 프로그램 대비 평균적으로 1.26× 의 성능 향상을 보였다.

루프를 포함하는 프로그램에 대해서는, FORTE 는 루프 인지적 코드 생성 및 최적화 기법을 통해 부트스트래핑 관리 프레임워크를 확장한다. FORTE 는 loop peeling과 부트스트래핑 배치를 통해 반복 사이에서 loop-carried 암호문의 암호화 상태와 레벨을 일치시켜, 모든 반복에서 일관되게 실행될 수 있는 루프 코드를 생성한다. 또한 반복 실행과 레벨 정합 과정에서 발생하는 오버헤드를 줄이기 위해 loop-carried 변수를 하나의 암호문으로 패킹하는 기법, 레벨 인지적 루프 언롤링, 그리고 부트스트래핑 목표 레벨 조정의 세 가지 최적화를 적용한다. 평가 결과, 동적 루프 반복 횟수를 갖는 프로그램에서 FORTE 는 전체 루프 언롤링에 의존하는 최신 컴파일러 대비 27% 의 성능 향상을 달성하였다.

핵심되는 말: 완전 동형암호, 부트스트래핑, 스케일 관리, 컴파일러 최적화, MLIR, RNS-CKKS