Faster CoeffToSlot and SlotToCoeff for Sparsely Packed Ciphertexts with Application to CKKS Bootstrapping

Xiaopeng Zheng

College of Mathematics and Computer Science, Shantou University, Shantou 515821, China xiaopengzheng@stu.edu.cn

Abstract. CKKS bootstrapping is a central tool for restoring the available modulus budget of approximate ciphertexts, thereby enabling homomorphic computations beyond a fixed leveled circuit. A key component is the pair of linear transformations CoeffToSlot and SlotToCoeff, which move data to the slot representation for homomorphic modular reduction and then back to the coefficient representation. In the sparse packing setting of Cheon et al. (EUROCRYPT 2018), the useful data occupy a short effective slot vector that is repeated across the full slot space. Existing methods for this setting mainly use the smaller effective dimension, whereas our approach exploits the repetition pattern itself to obtain simpler and cheaper transformations.

We use the repeated slot pattern to improve the efficiency of both CoeffToSlot and SlotToCoeff. Each transform keeps multiplicative depth 1 and uses fewer homomorphic operators. Let N be the ring dimension, let the packed vector have length $n/2$, and write $r = N/n$ for the repetition factor. For each transform, when $n \leq r/2$, the cost is one plaintext-ciphertext multiplication and $O(\log n)$ rotations. When $n > r/2$, the cost is $2n/r$ plaintext-ciphertext multiplications and $O(\sqrt{2n/r} + \log r)$ rotations. We also analyze the auxiliary slots produced by the new CoeffToSlot layout and prove that they satisfy the same sub-Gaussian range bound as the desired coefficient slots. Consequently, under sparse-secret encapsulation, our bounds give $K = 24$ for $p_{\text{fail}} < 2^{-128}$, compared with $K = 16$ for the standard layout.

We implement the proposed transforms in OpenFHE using its default multithreaded configuration and evaluate them as part of the CKKS bootstrapping pipeline with sparse-secret encapsulation. For $N = 2^{16}$ and the tested effective sparse dimensions $64 \leq n/2 \leq 1024$, our transforms are 2.08× to 8.87× faster than OpenFHE's depth 1 sparse linear transforms, yielding a 1.22× to 6.07× speedup in end-to-end bootstrapping. For $64 \leq n/2 \leq 512$, our algorithm is slightly faster in end-to-end bootstrapping than OpenFHE's depth 2 baseline (whose time is 1.03× to 1.25× that of our algorithm) while using two fewer levels, and is comparable to the depth 3 baseline (whose time is 0.86× to 1.07× that of our algorithm) while using four fewer levels. The timings include the resulting EvalMod overhead associated with the larger K required by our layout. Overall, the results show that slot repetition can reduce the practical cost of CKKS bootstrapping for sparsely packed ciphertexts.

Keywords: Homomorphic encryption · CKKS bootstrapping · Sparse packing · CoeffToSlot and SlotToCoeff

1 Introduction

Fully homomorphic encryption (FHE) enables computation on encrypted data without decryption and is a basic tool for privacy-preserving computation, with applications including private information retrieval and encrypted machine learning inference [38,17,24,12]. Since Gentry's breakthrough in 2009 [16], several efficient leveled schemes have been developed, including BGV, B/FV, CKKS [7,15,12]. Among them, CKKS is especially attractive for approximate arithmetic over real or complex numbers, but its ciphertext modulus is gradually consumed during homomorphic evaluation. To support arbitrary depth computation, Cheon et al. introduced the first bootstrapping algorithm for CKKS in 2018 [10].

The standard CKKS bootstrapping pipeline consists of modulus raising, CoeffToSlot(C2S), homomorphic modular reduction, and SlotToCoeff(S2C). Since 2018, this pipeline has been improved along several largely orthogonal directions. Chen et al. reduced the amortized bootstrapping cost per slot by about two orders of magnitude through level-collapsed FFT-like linear transforms and improved polynomial evaluation [8]. Han and Ki adapted the procedure to full-RNS CKKS [21]. Bossuat et al. made bootstrapping practical with non-sparse keys at 128-bit security [4]. Lee et al., Jutla and Manohar, and Lee et al. pushed the high-precision regime through better modular-reduction approximations and improved evaluation strategies [30,23,31]. Kim et al. and Sung et al. revisited the level cost of the standard pipeline through *EvalRound* and *EvalRound+* [27,34]. More recent work has further broadened the design space to data-specific and structurally redesigned bootstrapping, including bootstrapping for bits and small integers, shallow or highly parallel algorithms, partial CoeffToSlot, and specialized C2S/S2C optimizations [1,2,3,11,14,13,26,25,37,33]. Together, this line of work has moved beyond feasibility and now actively re-optimizes individual subroutines as well as the overall architecture.

Among these stages, CoeffToSlot and SlotToCoeff deserve separate attention. CKKS ciphertexts are RLWE ciphertexts over a polynomial ring, while SIMD computation is naturally expressed in the canonical embedding as a vector of complex slots. CoeffToSlot is the homomorphic decoding step that moves encrypted coefficients to slots so that the modular-reduction stage can be evaluated in parallel. SlotToCoeff applies the inverse transform and packs the refreshed slot values back into the polynomial form required for subsequent homomorphic arithmetic. These two steps are therefore not peripheral implementation details. They are the interfaces that connect the ring representation of CKKS ciphertexts with the slotwise view of approximate computation.

They are also expensive. In practice, both procedures are realized by dense or structured DFT-like linear transforms, so they require many rotations, many plaintext-ciphertext multiplications, and a substantial part of the available level budget. This cost has motivated a long line of work on homomorphic DFT and

CKKS-specific linear transforms. Cheon et al. and Han et al. showed that homomorphic DFT admits recursive Cooley–Tukey-type decompositions and that sparse diagonal structure can be exploited to accelerate linear transforms in bootstrapping [9,20]. Chen et al. [8] then translated this viewpoint into level-collapsed FFT-like transforms for CKKS bootstrapping. In their FFT-like realization, each layer uses only two rotations and three plaintext multiplications, and the resulting algorithm offers a spectrum of tradeoffs between operation count and level consumption. In particular, under a two-level budget, both CoeffToSlot and SlotToCoeff require $O(\ell^{1/4})$ rotations and $O(\sqrt{\ell})$ plaintext multiplications for an ℓ-dimensional transform. EvalRound later identified these linear transforms as a structural bottleneck, observing that about half of the levels in conventional CKKS bootstrapping are spent in CoeffToSlot and SlotToCoeff [27]. This line has continued in recent work that directly targets modulus consumption or redesigns these transforms, including *EvalRound+*, *OverModRaise*, faster CoeffToSlot, *SCORE*, and *PaCo* [34,26,37,33,14]. Bossuat et al. also improved the practical efficiency of the linear-transform stage through techniques such as double hoisting [4].

Sparse encoding in this work refers to the Cheon-style repeated-slot packing rather than a layout with many zero or invalid slots. In the original CKKS bootstrapping paper, Cheon et al. considered plaintexts in the subring $\mathbb{Z}[Y]/(Y^n + 1) \subset \mathbb{Z}[X]/(X^N + 1)$, with $Y = X^{N/n}$, and showed that the decoded slot vector is obtained by repeating an $n/2$-dimensional block N/n times [10]. This formulation reduces the cost of linear transforms in the sparse regime. The same notion also appears in encrypted machine learning. Lee et al. employ the sparse packing method used in CKKS bootstrapping to pack one CIFAR-10 channel using only 2^{10} sparse slots, since sparsely packed bootstrapping is faster and convolution requires fewer rotations [29]. Lee et al. later make this notion explicit under the name repetitive-slot (RS) packing, where a vector v is encoded as $(v|v|\cdots|v)$ before encryption and RS bootstrapping is used as a basic ingredient in compact CNN inference [28]. Throughout this paper, we use the term sparse encoding only in this repeated-slot sense.

Let $R_N = \mathbb{Z}[X]/(X^N + 1)$ and $R_n = \mathbb{Z}[Y]/(Y^n + 1)$, where $n \mid N$ and $r = N/n$. Via the embedding $Y \mapsto X^r$, we view R_n as a subring of R_N. A plaintext is sparsely packed if it belongs to this embedded subring. Under the canonical embedding, an effective slot vector $\boldsymbol{z} \in \mathbb{C}^{n/2}$ is encoded as

$$\mathbf{1}_r \otimes \boldsymbol{z} = (\underbrace{z_0, \ldots, z_{n/2-1}, z_0, \ldots, z_{n/2-1}, \ \cdots \ , z_0, \ldots, z_{n/2-1}}_{r \text{ times}})^{\mathrm{T}} \in \mathbb{C}^{N/2}.$$

where $\mathbf{1}_r = (1, \ldots, 1)^T$, and $\mathbf{1}_r \otimes \boldsymbol{z}$ denotes the vector obtained by concatenating r copies of $\boldsymbol{z}$. Equivalently, the corresponding plaintext polynomial is a polynomial in X^r, so only every r-th coefficient may be nonzero. A sparsely packed CKKS ciphertext is therefore not only lower-dimensional. It also has a repeated-slot or slot-periodic structure.

This repeated-slot structure suggests a further optimization opportunity. The original sparse formulation mainly exploits the smaller effective dimension, while

the associated CoeffToSlot and SlotToCoeff procedures still follow the generic DFT-like viewpoint. In other words, the benefit mainly comes from dimension reduction, not yet from a transform design built explicitly around the repeated-slot pattern itself. Subsequent HDFT and FFT-like decompositions reveal much richer structure in these linear transforms [9,20,8]. However, reducing the number of transform operations does not automatically lead to a better bootstrapping circuit, because these decompositions may also introduce extra multiplicative stages. Level-collapsing mitigates this tradeoff, but it does not remove it. In CKKS, extra levels are costly because they usually require a longer modulus chain and larger evaluation-key material. Since CoeffToSlot and SlotToCoeff already account for a large fraction of the level consumption [27], it is natural to ask whether the slot-periodic structure of sparse ciphertexts can be exploited more directly.

Can we fully exploit the slot-periodic structure of sparsely packed vectors to design more efficient CoeffToSlot and SlotToCoeff algorithms without increasing the multiplicative level?

We answer this question affirmatively in the targeted sparse regimes by exploiting slot repetition to reduce the operation counts of both transforms while preserving multiplicative depth one.

1.1 Main result

For clarity, we present the main result at the plaintext-circuit level using the notation of Cheon et al. [10]. Our implementation, however, realizes the same linear maps in a full-RNS CKKS bootstrapping framework and follows the standard scale-management and matrix-scaling optimizations of Bossuat et al. [4].

In this paper, we mainly follow the notation in [10]. First, we briefly recall how CoeffToSlot and SlotToCoeff are performed in the sparsely packed setting. Let $R = \mathbb{Z}[X]/(X^N + 1)$, where N is a power of two. For an integer $n \mid N$, set $r = N/n$ and $Y = X^r$. Suppose $r > 1$. Then the subring generated by Y is naturally identified with $\mathbb{Z}[Y]/(Y^n+1) \subset R$. Let $\zeta = \exp(-\pi i/N)$ be a primitive $2N$-th root of unity, and let $\xi = \exp(-\pi i/n) = \zeta^{N/n}$ be a primitive $2n$-th root of unity. For $0 \le j < n/2$, write $\xi_j = \xi^{5^j}$. If $m(Y) \in Z[Y]/(Y^n + 1)$, then its canonical embedding in the full slot space $\mathbb{C}^{N/2}$ is obtained by repeating the vector $\boldsymbol{z} = (m(\xi_0), m(\xi_1), \ldots, m(\xi_{n/2-1}))^T \in \mathbb{C}^{n/2}$ exactly r times.

Write $m(Y) = a_0 + a_1 Y + \cdots + a_{n-1} Y^{n-1}$ and $\boldsymbol{a} = (a_0, a_1, \ldots, a_{n-1})^T$. Define

$$U = \begin{pmatrix} 1 & \xi_0 & \xi_0^2 & \cdots & \xi_0^{n-1} \\ 1 & \xi_1 & \xi_1^2 & \cdots & \xi_1^{n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \xi_{n/2-1} & \xi_{n/2-1}^2 & \cdots & \xi_{n/2-1}^{n-1} \end{pmatrix} \in \mathbb{C}^{n/2 \times n}.$$

Then $\boldsymbol{z} = U\boldsymbol{a}$. Since the coefficients a_i are real, we also have $\overline{\boldsymbol{z}} = \overline{U}\boldsymbol{a}$, and hence

$$\begin{pmatrix} \boldsymbol{z} \\ \overline{\boldsymbol{z}} \end{pmatrix} = \begin{pmatrix} U \\ \overline{U} \end{pmatrix} \boldsymbol{a} = V\boldsymbol{a}.$$

The matrix V is the full Fourier-type evaluation matrix for $Y^n + 1$, so $V^{-1} = \frac{1}{n}V^*$. Therefore,

$$\boldsymbol{a} = \frac{1}{n}\left(\overline{U}^T\ U^T\right)\begin{pmatrix}\boldsymbol{z}\\ \overline{\boldsymbol{z}}\end{pmatrix} = \frac{1}{n}\left(\overline{U}^T\boldsymbol{z} + U^T\overline{\boldsymbol{z}}\right).$$

Equivalently, if $\boldsymbol{b} = \frac{1}{n}\overline{U}^T\boldsymbol{z}$, then $\boldsymbol{a} = \boldsymbol{b} + \overline{\boldsymbol{b}}$. Thus, in the terminology of CKKS bootstrapping, the CoeffToSlot step maps the canonical slot vector $\boldsymbol{z}$ to the coefficient vector $\boldsymbol{a}$. More precisely, it homomorphically computes $\boldsymbol{b} = \frac{1}{n}\overline{U}^T\boldsymbol{z}$ and outputs $\boldsymbol{a} = \boldsymbol{b} + \overline{\boldsymbol{b}}$. Conversely, the SlotToCoeff step maps the coefficient vector $\boldsymbol{a}$ back to the canonical slot vector by homomorphically computing $U\boldsymbol{a}$.

Assume now that the sparsity multiplicity $r = N/n$ satisfies $r > 1$. In this case, the vector $\boldsymbol{b}$ has length $n \le N/2$, and hence it can be stored in a single ciphertext. Therefore, if one uses the standard linear-transform method without FFT-style factorization, as in [10,4], both CoeffToSlot and SlotToCoeff can be implemented with $n/2$ plaintext-ciphertext multiplications, $O(\sqrt{n/2})$ rotations via the baby-step giant-step optimization, and multiplicative depth 1. In this paper, we propose a new algorithm that further exploits the repeated-slot structure inherent in sparse packing. This allows us to reduce both the number of plaintext-ciphertext multiplications and the number of rotations, while preserving multiplicative depth 1. Our main result is summarized as follows.

Theorem 1 (Sparse CoeffToSlot and SlotToCoeff). *Let N be a power of two, let $n \mid N$, and set $r = N/n$. Assume $r \ge 4$ and that an effective $n/2$-slot vector is stored in the full $N/2$-slot space as r repeated copies. Then the proposed CoeffToSlot and SlotToCoeff transforms for sparse CKKS bootstrapping both have multiplicative depth 1. For each transform, the online complexity is as follows, where P-CMult denotes a plaintext-ciphertext multiplication:*

Regime	P-CMults	Rotations
$n \le r/2$	1	$O(\log n)$
$n > r/2$	$2n/r$	$O\left(\sqrt{2n/r} + \log r\right)$

† One additional homomorphic conjugation is required in CoeffToSlot.

See Sections 3.2 and 3.3 for exact per-transform operation counts.

In typical CKKS bootstrapping parameters, the number of available slots $N/2$ is often 32768 or 65536. Hence, when the sparsely encoded vector has dimension at most 1024, the sparsity multiplicity r is large, and the repeated-slot structure can lead to a substantial speedup. Moreover, this improvement does not require any additional multiplicative depth, i.e., the transforms still have depth 1.

1.2 Our techniques

We now explain the main idea behind the proposed algorithms. It is enough to study two linear transformations: $\boldsymbol{z} \mapsto \frac{1}{n}\overline{U}^T \boldsymbol{z}$ for $\mathsf{CoeffToSlot}$, together with the final conjugation and addition, and $\boldsymbol{a} \mapsto U\boldsymbol{a}$ for $\mathsf{SlotToCoeff}$. In the sparsely packed setting, the full slot vector consists of repeated copies of the same $n/2$-dimensional vector. Our algorithms exploit this repeated-slot structure to evaluate the above linear transformations more efficiently. We illustrate the procedure through a simple example. For convenience, we present the following procedure at the plaintext-vector level. The corresponding operations are implemented homomorphically on ciphertexts.

Suppose $N/2 = 64$, $n/2 = 4$, and $r = 16$. Then $\boldsymbol{m} = \mathbf{1}_{16} \otimes (z_0, z_1, z_2, z_3)^T = \mathbf{1}_{16} \otimes \boldsymbol{z} \in \mathbb{C}^{64}$. Let $U = (u_{ij}) \in \mathbb{C}^{4\times 8}$. Then $\frac{1}{n}\overline{U}^T \in \mathbb{C}^{8\times 4}$. Write $V = \frac{1}{n}\overline{U}^T = (v_{ij})$. We write $\boldsymbol{m}$ as

$$\begin{aligned}\boldsymbol{m} = (&z_0\ z_1\ z_2\ z_3\ z_0\ z_1\ z_2\ z_3\ z_0\ z_1\ z_2\ z_3\ z_0\ z_1\ z_2\ z_3\\ &z_0\ z_1\ z_2\ z_3\ z_0\ z_1\ z_2\ z_3\ z_0\ z_1\ z_2\ z_3\ z_0\ z_1\ z_2\ z_3\\ &z_0\ z_1\ z_2\ z_3\ z_0\ z_1\ z_2\ z_3\ z_0\ z_1\ z_2\ z_3\ z_0\ z_1\ z_2\ z_3\\ &z_0\ z_1\ z_2\ z_3\ z_0\ z_1\ z_2\ z_3\ z_0\ z_1\ z_2\ z_3\ z_0\ z_1\ z_2\ z_3)^T.\end{aligned}$$

Let $\boldsymbol{v} = (\boldsymbol{v}_0; \boldsymbol{v}_0; \boldsymbol{v}_1; \boldsymbol{v}_1; \boldsymbol{v}_2; \boldsymbol{v}_2; \boldsymbol{v}_3; \boldsymbol{v}_3; \boldsymbol{v}_4; \boldsymbol{v}_4; \boldsymbol{v}_5; \boldsymbol{v}_5; \boldsymbol{v}_6; \boldsymbol{v}_6; \boldsymbol{v}_7; \boldsymbol{v}_7)$, where $\boldsymbol{v}_i$ is the i-th row of V, and the semicolon denotes vertical concatenation.

We first compute $\boldsymbol{y}_0 = \boldsymbol{m} \odot \boldsymbol{v}$, where $\odot$ denotes the Hadamard product, i.e., entrywise multiplication. Let $\boldsymbol{y}_1 = \mathrm{Rot}(\boldsymbol{y}_0, 1) + \boldsymbol{y}_0$ and $\boldsymbol{y}_2 = \mathrm{Rot}(\boldsymbol{y}_1, 2) + \boldsymbol{y}_1$, where $\mathrm{Rot}(\boldsymbol{x}, k)$ denotes the cyclic left rotation of $\boldsymbol{x}$ by k slots. Then

$$\begin{aligned}\boldsymbol{y}_2 = (&b_0\ b_0\ b_0\ b_0\ b_0\ *\ *\ *\ b_1\ b_1\ b_1\ b_1\ b_1\ *\ *\ *\\ &b_2\ b_2\ b_2\ b_2\ b_2\ *\ *\ *\ b_3\ b_3\ b_3\ b_3\ b_3\ *\ *\ *\\ &b_4\ b_4\ b_4\ b_4\ b_4\ *\ *\ *\ b_5\ b_5\ b_5\ b_5\ b_5\ *\ *\ *\\ &b_6\ b_6\ b_6\ b_6\ b_6\ *\ *\ *\ b_7\ b_7\ b_7\ b_7\ b_7\ *\ *\ *)^T.\end{aligned},$$

where $(b_0, b_1, \ldots, b_7)^T = \frac{1}{n}\overline{U}^T \boldsymbol{z}$, and $*$ denotes some auxiliary entry in $\mathbb{C}$. The entries denoted by $*$ are not used in the subsequent $\mathsf{SlotToCoeff}$ step. In the precision analysis, one should ensure that these auxiliary slots remain within the approximation range of the modular reduction polynomial.

Next, we compute $\boldsymbol{y}_2 + \mathrm{Conj}(\boldsymbol{y}_2)$ and obtain

$$\begin{aligned}\boldsymbol{y}_2 + \mathrm{Conjugate}(\boldsymbol{y}_2) = (&a_0\ a_0\ a_0\ a_0\ a_0\ *\ *\ *\ a_1\ a_1\ a_1\ a_1\ a_1\ *\ *\ *\\ &a_2\ a_2\ a_2\ a_2\ a_2\ *\ *\ *\ a_3\ a_3\ a_3\ a_3\ a_3\ *\ *\ *\\ &a_4\ a_4\ a_4\ a_4\ a_4\ *\ *\ *\ a_5\ a_5\ a_5\ a_5\ a_5\ *\ *\ *\\ &a_6\ a_6\ a_6\ a_6\ a_6\ *\ *\ *\ a_7\ a_7\ a_7\ a_7\ a_7\ *\ *\ *)^T.\end{aligned}$$

In this way, we compute $\boldsymbol{a} = \frac{1}{n}(\overline{U}^T \boldsymbol{z} + U^T \overline{\boldsymbol{z}})$ in the slots using only one plaintext-ciphertext multiplication, two rotations, one conjugation, and additions.

We then perform modular reduction slotwise. Suppose the resulting vector is

$$\begin{aligned}\boldsymbol{g} = (&a'_0\ a'_0\ a'_0\ a'_0\ a'_0 * * * a'_1\ a'_1\ a'_1\ a'_1\ a'_1 * * *\\ &a'_2\ a'_2\ a'_2\ a'_2\ a'_2 * * * a'_3\ a'_3\ a'_3\ a'_3\ a'_3 * * *\\ &a'_4\ a'_4\ a'_4\ a'_4\ a'_4 * * * a'_5\ a'_5\ a'_5\ a'_5\ a'_5 * * *\\ &a'_6\ a'_6\ a'_6\ a'_6\ a'_6 * * * a'_7\ a'_7\ a'_7\ a'_7\ a'_7 * * *)^T.\end{aligned}$$

For SlotToCoeff, define

$$\begin{aligned}\boldsymbol{u} = (&u_{0,0}\ u_{1,0}\ u_{2,0}\ u_{3,0}\ 0\ 0\ 0\ 0\ u_{0,1}\ u_{1,1}\ u_{2,1}\ u_{3,1}\ 0\ 0\ 0\ 0\\ &u_{0,2}\ u_{1,2}\ u_{2,2}\ u_{3,2}\ 0\ 0\ 0\ 0\ u_{0,3}\ u_{1,3}\ u_{2,3}\ u_{3,3}\ 0\ 0\ 0\ 0\\ &u_{0,4}\ u_{1,4}\ u_{2,4}\ u_{3,4}\ 0\ 0\ 0\ 0\ u_{0,5}\ u_{1,5}\ u_{2,5}\ u_{3,5}\ 0\ 0\ 0\ 0\\ &u_{0,6}\ u_{1,6}\ u_{2,6}\ u_{3,6}\ 0\ 0\ 0\ 0\ u_{0,7}\ u_{1,7}\ u_{2,7}\ u_{3,7}\ 0\ 0\ 0\ 0)^T.\end{aligned}$$

Let $\boldsymbol{h}_0 = \boldsymbol{g} \odot \boldsymbol{u}$. We compute $\boldsymbol{h}_1 = \mathrm{Rot}(\boldsymbol{h}_0, 8) + \boldsymbol{h}_0$, $\boldsymbol{h}_2 = \mathrm{Rot}(\boldsymbol{h}_1, 16) + \boldsymbol{h}_1$, and $\boldsymbol{h}_3 = \mathrm{Rot}(\boldsymbol{h}_2, 32) + \boldsymbol{h}_2$. Finally, let $\boldsymbol{h}_{\mathrm{out}} = \mathrm{Rot}(\boldsymbol{h}_3, 4) + \boldsymbol{h}_3$. Then we obtain

$$\boldsymbol{h}_{\mathrm{out}} = \mathbf{1}_{16} \otimes \boldsymbol{z}', \qquad \boldsymbol{z}' = U(a'_0, a'_1, \ldots, a'_7)^T.$$

Thus SlotToCoeff in this example requires only one plaintext-ciphertext multiplication and $4 = \log_2 n + 1$ rotations.

1.3 Related Work

C2S/S2C in CKKS Bootstrapping Several works have identified CoeffToSlot and SlotToCoeff as a major bottleneck in CKKS bootstrapping. Chen et al. used FFT-like decompositions together with level-collapsing and reduced the amortized bootstrapping cost by about two orders of magnitude [8]. *EvalRound* and *EvalRound+* then revisited the same bottleneck from the viewpoint of level and modulus consumption [27,34]. More recent works optimize the two transforms more directly. *SCORE* accelerates SlotToCoeff, *PaCo* replaces the usual full CoeffToSlot by a partial version, and *OverModRaise* together with the less-modulus design of Yan et al. further reduces the modulus cost of the C2S/S2C part [33,14,26,37]. Our work is closest to this line, but focuses on sparsely encoded CKKS ciphertexts and exploits their repeated-slot structure to accelerate both transforms without increasing the multiplicative level.

Homomorphic Linear Transformations A foundational line of work on homomorphic linear algebra was developed by Halevi and Shoup [18,19]. They described generic SIMD matrix-vector algorithms, including the *Matrix in row-order* method, which stores the matrix rows as separate platform vectors and combines the rowwise products through the transposition principle, and the *Matrix in diagonal order* method, which stores generalized diagonals and evaluates the product through rotations and plaintext multiplications [18]. In later work, they introduced faster linear-transformation routines and reported substantial speedups over the earlier HElib implementation [19]. Related ideas also appear in encrypted machine learning. Juvekar et al. compared naive, diagonal, and

hybrid matrix-vector layouts, where the naive method encodes each row of the plaintext weight matrix into a separate plaintext vector, while the hybrid layout is typically the fastest in their experiments [24]. Our technique is closest in spirit to the row-order viewpoint, but differs in the matrix layout. Halevi and Shoup treat the rows as separate vectors, whereas we pack all rows into a single plaintext and exploit that packed structure in the subsequent transform design.

2 Preliminary

Throughout this paper, we mainly follow the notation of Cheon et al. [10] for CKKS bootstrapping. Unless otherwise specified, $\log(\cdot)$ denotes the binary logarithm. Vectors are written in bold lower case letters, and all vectors are regarded as column vectors. For matrices $A_1 \in \mathbb{C}^{n_1 \times m}$ and $A_2 \in \mathbb{C}^{n_2 \times m}$, we use $(A_1; A_2)$ to denote their vertical concatenation, namely the $(n_1+n_2)\times m$ matrix obtained by placing A_2 below A_1. For a positive integer r and a vector $\boldsymbol{z}$, we write $\mathbf{1}_r \otimes \boldsymbol{z}$ for the vertical concatenation of r copies of $\boldsymbol{z}$, namely $\mathbf{1}_r \otimes \boldsymbol{z} = (\boldsymbol{z}; \boldsymbol{z}; \ldots; \boldsymbol{z})$. We write $\langle \cdot, \cdot \rangle$ for the standard inner product. For a real number x, the notation $\lfloor x \rceil$ denotes the nearest integer to x, with ties rounded upward. For an integer modulus q, we identify $\mathbb{Z}_q$ with the centered representative set $\mathbb{Z} \cap (-q/2, q/2]$. Accordingly, for an integer z, we write $[z]_q$ for the reduction of z modulo q into this interval. The notation $x \leftarrow D$ means that x is sampled according to the distribution D, and $U(S)$ denotes the uniform distribution over a finite set S. The security parameter is denoted by λ.

2.1 CKKS scheme

Homomorphic encryption allows computations to be performed directly on encrypted data, and is therefore a natural tool for secure outsourced computation. A major difficulty in many applications, however, is that the data are real or complex numbers rather than elements of a small finite plaintext space. The CKKS scheme, originally introduced as HEAAN by Cheon et al. [12], addresses this issue by supporting approximate arithmetic over encrypted messages. Its rescaling procedure plays the role of rounding in fixed point computation, and keeps the size of encrypted plaintexts under control after homomorphic multiplications. We briefly recall the distributions used in the RLWE based description of the scheme.

For $\sigma > 0$, let $\mathcal{DG}(\sigma^2)$ denote the distribution over $\mathbb{Z}^N$ whose coordinates are sampled independently from the discrete Gaussian distribution with variance σ^2. For a finite set S, let $\mathcal{U}(S)$ denote the uniform distribution over S; in particular, $\mathcal{U}(R_Q)$ denotes the uniform distribution over the residue ring R_Q. For an interval (a, b), we also write $\mathcal{U}(a, b)$ for the continuous uniform distribution on (a, b). For a positive integer h, let $\mathcal{HWT}(h)$ denote the uniform distribution over signed binary vectors in $\{0, \pm 1\}^N$ with exactly h nonzero entries. For $0 \le \rho \le 1$, let $\mathcal{ZO}(\rho)$ be the distribution over $\{0, \pm 1\}^N$ in which each coordinate is independently sampled as 1 with probability $\rho/2$, as -1 with probability $\rho/2$, and as

0 with probability $1-\rho$. These distributions are used for the secret key, error terms, and auxiliary encryption randomness in the scheme.

Encoding and decoding We recall the sparse packing convention used in this paper. Let N be a power of two, and let $n \mid N$, with $r = N/n$ and $Y = X^r$. The substitution $Y \mapsto X^r$ embeds $\mathbb{Z}[Y]/(Y^n+1)$ into $\mathbb{Z}[X]/(X^N+1)$.

Let $\zeta = \exp(-\pi i/N)$ and $\xi = \zeta^r = \exp(-\pi i/n)$. For $0 \le j < n/2$, set $\xi_j = \xi^{5^j}$. For $m(Y) \in \mathbb{Z}[Y]/(Y^n+1)$, define

$$\tau_n(m) = (m(\xi_0), m(\xi_1), \ldots, m(\xi_{n/2-1}))^T \in \mathbb{C}^{n/2}.$$

If $\widetilde{m}(X) = m(X^r)$ is viewed as a polynomial in $\mathbb{Z}[X]/(X^N+1)$, its ordinary CKKS decoding in the full slot space satisfies $\tau_N(\widetilde{m}) = \mathbf{1}_r \otimes \tau_n(m)$, where τ_N is the analogous decoding map for $\mathbb{Z}[X]/(X^N+1)$. Thus, sparse packing stores an effective vector $\boldsymbol{z} \in \mathbb{C}^{n/2}$ in $N/2$ slots as $\mathbf{1}_r \otimes \boldsymbol{z}$, i.e., by repeating the same vector $r = N/n$ times. This is the sparsely packed ciphertext setting introduced in Cheon et al. [10].

The CKKS encoding and decoding operations can be described as follows:

- $\mathsf{Ecd}(\boldsymbol{\mu})$ (Encoding): For $\boldsymbol{\mu} \in \mathbb{C}^{n/2}$, multiply by the scale Δ, i.e., $\boldsymbol{z} = \Delta\boldsymbol{\mu}$, then compute the inverse canonical map $m(Y) = \lfloor \tau_n^{-1}(\boldsymbol{z}) \rceil$. Return $\widetilde{m}(X) = m(X^r)$.
- $\mathsf{Dcd}(\widetilde{m})$ (Decoding): For $\widetilde{m}(X) \in \mathbb{Z}[X]/(X^N+1)$, restrict to the subring to get $m(Y)$, then compute $\boldsymbol{z} = \tau_n(m(Y))$ and recover the original vector as $\boldsymbol{\mu} = \boldsymbol{z}/\Delta$.

Basic operations. We briefly recall the CKKS operations used in this paper, following [10,8]. Let $R = \mathbb{Z}[X]/(X^N+1)$ and $R_Q = R/QR$. We use a modulus chain $p_0, \ldots, p_{L-1}$, where the p_i's may have different bit sizes, and set $Q_\ell = \prod_{i=0}^{\ell-1} p_i$ for $1 \le \ell \le L$. A ciphertext at level ℓ is a pair $\mathsf{ct} = (c_0, c_1) \in R_{Q_\ell}^2$, and its decryption value under the secret key $s \in R$ is $c_0 + c_1 s$ modulo Q_ℓ.

- $\mathsf{Setup}(1^\lambda)$. Choose the parameters N, h, P, σ and a modulus chain $p_0, \ldots, p_{L-1}$ so that the underlying RLWE problem achieves the desired security level.
- KeyGen. Sample $s \leftarrow \mathcal{HWT}(h)$, $a \leftarrow U(R_{Q_L})$, and $e \leftarrow \mathcal{DG}(\sigma^2)$. Set $\mathsf{sk} = (1, s)$ and $\mathsf{pk} = (b, a) \in R_{Q_L}^2$, where $b = -as + e \pmod{Q_L}$.
- $\mathsf{KSwitchGen}_s(s')$. For $s' \in R$, sample $a' \leftarrow U(R_{PQ_L})$ and $e' \leftarrow \mathcal{DG}(\sigma^2)$. Output the switching key $\mathsf{swk}_{s' \to s} = (b', a') \in R_{PQ_L}^2$, where $b' = -a's + e' + Ps' \pmod{PQ_L}$. In particular, the evaluation key is $\mathsf{evk} = \mathsf{KSwitchGen}_s(s^2)$.
- $\mathsf{Enc}_{\mathsf{pk}}(m)$. For a plaintext polynomial $m \in R$, sample $v \leftarrow \mathcal{ZO}(0.5)$ and $e_0, e_1 \leftarrow \mathcal{DG}(\sigma^2)$. Output

$$\mathsf{ct} = v \cdot \mathsf{pk} + (m + e_0, e_1) \pmod{Q_L}.$$

  Then $c_0 + c_1 s = m + e \pmod{Q_L}$ for a small error polynomial e.
- $\mathsf{Dec}_s(\mathsf{ct})$. For $\mathsf{ct} = (c_0, c_1) \in R_{Q_\ell}^2$, output $[c_0 + c_1 s]_{Q_\ell}$.

- $\mathsf{Add}(\mathsf{ct}, \mathsf{ct}')$. For two ciphertexts at the same level ℓ, output $\mathsf{ct}+\mathsf{ct}' \pmod{Q_\ell}$.
- $\mathsf{CMult}(\mathsf{ct}, u)$. For a plaintext polynomial $u \in R_{Q_\ell}$ and $\mathsf{ct} = (c_0, c_1) \in R_{Q_\ell}^2$, output $(uc_0, uc_1) \pmod{Q_\ell}$. When u is specified by a slot vector, we use the corresponding encoded plaintext polynomial.
- $\mathsf{Mult}_{\mathsf{evk}}(\mathsf{ct}, \mathsf{ct}')$. For $\mathsf{ct} = (c_0, c_1)$ and $\mathsf{ct}' = (c_0', c_1')$ in $R_{Q_\ell}^2$, compute

$$(d_0, d_1, d_2) = (c_0c_0',\ c_0c_1' + c_1c_0',\ c_1c_1') \pmod{Q_\ell}.$$

  Output $(d_0, d_1) + \mathsf{KeySwitch}_{\mathsf{evk}}(d_2)$.
- $\mathsf{RS}_{\ell\to\ell'}(\mathsf{ct})$. For $1 \le \ell' < \ell \le L$, output

$$\mathsf{ct}' = \left\lfloor \frac{Q_{\ell'}}{Q_\ell}\mathsf{ct} \right\rceil \pmod{Q_{\ell'}},$$

  where the rounding is applied coefficientwise. Equivalently, this divides the ciphertext by $Q_\ell/Q_{\ell'} = \prod_{i=\ell'}^{\ell-1} p_i$ and reduces the modulus to $Q_{\ell'}$. We write $\mathsf{RS}(\mathsf{ct})$ when $\ell' = \ell - 1$, in which case the divisor is $p_{\ell-1}$.

We also use slot rotations and conjugation. For an integer k coprime to $2N$, let σ_k be the automorphism of R defined by $\sigma_k(a(X)) = a(X^k)$. Given a ciphertext ct, applying σ_k to both components gives a ciphertext under the transformed secret key $\sigma_k(s)$. A key switching step brings it back to the original key s.

- $\mathsf{Rot}(\mathsf{ct}, t)$. Let $\mathsf{rk}_t = \mathsf{KSwitchGen}_s(\sigma_{5^t}(s))$. The operation

$$\mathsf{Rot}(\mathsf{ct}, t) = \mathsf{KeySwitch}_{\mathsf{rk}_t}(\sigma_{5^t}(\mathsf{ct}))$$

  outputs a ciphertext whose slots are cyclically shifted by t positions.
- $\mathsf{Conj}(\mathsf{ct})$. Let $\mathsf{ck} = \mathsf{KSwitchGen}_s(\sigma_{-1}(s))$. The operation

$$\mathsf{Conj}(\mathsf{ct}) = \mathsf{KeySwitch}_{\mathsf{ck}}(\sigma_{-1}(\mathsf{ct}))$$

  outputs a ciphertext encrypting the slotwise complex conjugate.

As usual in CKKS, encryption, key switching, rescaling, and homomorphic multiplication introduce small approximation errors. We do not reanalyze these standard errors in this paper. Our later analysis focuses on the additional range issue caused by the auxiliary slots produced by our sparse $\mathsf{CoeffToSlot}$ layout.

2.2 CKKS bootstrapping

We recall the sparse CKKS bootstrapping pipeline only at the level needed for our linear-transform algorithms; a more detailed derivation is given in Appendix A. Let $\mathsf{ct} = (c_0, c_1) \in R_q^2$ encrypt a sparse plaintext $m(Y) = \sum_{i=0}^{n-1} m_i Y^i \in \mathbb{Z}[Y]/(Y^n + 1)$, where $Y = X^r$ and $r = N/n$. The CKKS scale is included in the coefficients m_i. After choosing centered integer representatives, the decryption value can be written, up to the usual CKKS approximation errors, as

$$c_0(X) + c_1(X)s(X) \approx qI(X) + m(Y)$$

in $\mathbb{Z}[X]/(X^N+1)$, for some integer polynomial $I(X)$. Thus, after ModRaise to the larger modulus Q_L, bootstrapping amounts to removing the extra term $qI(X)$ while preserving $m(Y)$.

In the sparse setting, $m(Y)$ belongs to the subring generated by $Y = X^r$, whereas $I(X)$ need not. The SubSum step extracts the corresponding subring component $\widetilde{I}(Y) = I_0 + I_r Y + \cdots + I_{(n-1)r} Y^{n-1}$, up to a public normalization that can be absorbed into the surrounding linear maps. We then write

$$a(Y) = q\widetilde{I}(Y) + m(Y) = \sum_{i=0}^{n-1} a_i Y^i, \qquad a_i = q\widetilde{I}_i + m_i.$$

The CoeffToSlot step moves the coefficient vector $\boldsymbol{a} = (a_0, \ldots, a_{n-1})^T$ into slots, EvalMod applies an approximate centered reduction modulo q slotwise, and SlotToCoeff maps the resulting coefficients back to the sparse slot encoding. Hence, up to CKKS approximation errors,

$$\begin{aligned} \mathsf{ct} &\xrightarrow{\mathsf{ModRaise}} qI(X) + m(Y) \xrightarrow{\mathsf{SubSum}} a(Y) = q\widetilde{I}(Y) + m(Y) \\ &\xrightarrow{\mathsf{CoeffToSlot}} (a_i)_i \xrightarrow{\mathsf{EvalMod}} (a'_i)_i \xrightarrow{\mathsf{SlotToCoeff}} m'(Y), \end{aligned}$$

where $a'_i \approx [a_i]_q = m_i$ and $m'(Y) = \sum_i a'_i Y^i \approx m(Y)$. The implementation-specific normalization before EvalMod must keep all entries entering the approximation range, including the auxiliary entries created by the sparse layout. Our contribution improves the two linear transformations CoeffToSlot and SlotToCoeff by exploiting the repeated-slot structure $\mathbf{1}_r \otimes \tau_n(\cdot)$. This reduces rotations and plaintext-ciphertext multiplications without increasing the multiplicative depth.

3 SlotToCoeff and CoeffToSlot for Sparse Packed Ciphertexts

3.1 Basic Algorithm: Homomorphic Partial Sums over Slot Blocks

In this section, we recall a blockwise rotate-and-add idea similar to the SubSum procedure used in [32]. Suppose that a ciphertext ct encrypts a slot vector divided into b consecutive blocks of the same length k,

$$\boldsymbol{m} = (\boldsymbol{u}_0; \boldsymbol{u}_1; \cdots; \boldsymbol{u}_{b-1}) \in \mathbb{C}^{N/2}, \qquad \boldsymbol{u}_i \in \mathbb{C}^k,$$

where $bk = N/2$. For a power-of-two window length $\ell = 2^d \le b$, our goal is to compute a ciphertext encrypting

$$\boldsymbol{m}^{(\ell)} = (\boldsymbol{s}_0; \boldsymbol{s}_1; \cdots; \boldsymbol{s}_{b-1}), \qquad \boldsymbol{s}_i = \sum_{t=0}^{\ell-1} \boldsymbol{u}_{(i+t) \bmod b}.$$

In other words, each output block is the coordinate-wise sum of ℓ consecutive input blocks, where block indices are taken modulo b. In particular, when $\ell = b$,

the procedure computes the global block sum and replicates it to every block. That is,

$$\boldsymbol{s}_0 = \boldsymbol{s}_1 = \cdots = \boldsymbol{s}_{b-1} = \sum_{i=0}^{b-1} \boldsymbol{u}_i.$$

The algorithm is given in Algorithm 1. The procedure uses only $\log_2 \ell$ rotations and additions, and hence does not consume any multiplicative level.

Algorithm 1 $\mathsf{BlockPartialSum}(\mathsf{ct}, k, \ell)$

Input: A ciphertext ct whose slots are divided into blocks of length k, and a power-of-two window length ℓ.

Output: A ciphertext encrypting $(\boldsymbol{s}_0; \ldots; \boldsymbol{s}_{b-1})$, where $\boldsymbol{s}_i = \sum_{t=0}^{\ell-1} \boldsymbol{u}_{(i+t) \bmod b}$.

1: $\mathsf{ct}' \leftarrow \mathsf{ct}$
2: **for** $p = 0$ to $\log_2 \ell - 1$ **do**
3: $\quad \mathsf{ct}' \leftarrow \mathsf{ct}' + \mathrm{Rot}(\mathsf{ct}', 2^p k)$
4: **end for**
5: **return** ct'

For example, let $b = 8$, $\ell = 4$, and $k = 2$. Suppose that the encrypted slot vector is divided into eight blocks

$$\boldsymbol{m} = ((1,2),(3,4),(5,6),(7,8),(9,10),(11,12),(13,14),(15,16))^T.$$

Then Algorithm 1 outputs

$$((16,20),(24,28),(32,36),(40,44),(48,52),(40,44),(32,36),(24,28))^T.$$

For instance, the first output block is $(1,2)+(3,4)+(5,6)+(7,8) = (16,20)$, and the last output block is $(15,16)+(1,2)+(3,4)+(5,6) = (24,28)$, where the block indices are taken modulo 8.

We will use a simple duplication trick in our sparse $\mathsf{CoeffToSlot}$ construction. Suppose we want $\ell+1$ consecutive output blocks to be equal to the sum of some ℓ input blocks. We place these ℓ blocks twice. Then each length-ℓ window inside the duplicated region contains the same blocks, only cyclically shifted. Hence Algorithm 1 makes the corresponding output blocks all equal to the same block sum. For example, let $b = 16$, $\ell = 4$, and $k = 2$. Let

$$\begin{aligned}\boldsymbol{m} = \big((1,2);(3,4);(5,6);(7,8);(1,2);(3,4);(5,6);(7,8);\\ (9,10);(11,12);(13,14);(15,16);(9,10);(11,12);(13,14);(15,16)\big).\end{aligned}$$

Here the first four blocks are duplicated in blocks $4,\ldots,7$, and the blocks $8,\ldots,11$ are duplicated in blocks $12,\ldots,15$. Applying Algorithm 1 with $\ell = 4$ gives

$$\begin{aligned}\boldsymbol{m}^{(4)} = \big((16,20);(16,20);(16,20);(16,20);(16,20);(24,28);(32,36);(40,44);\\ (48,52);(48,52);(48,52);(48,52);(48,52);(40,44);(32,36);(24,28)\big).\end{aligned}$$

In particular, duplicating a length-ℓ block pattern produces a run of identical partial sums. We will exploit this property in the sparse CoeffToSlot procedure below.

3.2 CoeffToSlot and SlotToCoeff for Sparse Packed Ciphertexts when $n \le r/2$

Let N be the ring dimension, and let $n \mid N$. Set $r = N/n$ and $Y = X^r$. In the sparse packing considered here, an effective vector $\boldsymbol{z} \in \mathbb{C}^{n/2}$ is stored in the full CKKS slot space as

$$\mathbf{1}_r \otimes \boldsymbol{z} \in \mathbb{C}^{N/2}.$$

After ModRaise and SubSum, let $a(Y) = q\widetilde{I}(Y) + m(Y)$, and write $\boldsymbol{a} = (a_0, \ldots, a_{n-1})^T$ for its coefficient vector. Its sparse slot encoding is $\boldsymbol{z} = U\boldsymbol{a}$, where $U \in \mathbb{C}^{(n/2)\times n}$ is the Fourier-type matrix defined above. Thus the task of CoeffToSlot is to recover

$$\boldsymbol{a} = \frac{1}{n}\left(\overline{U}^T\boldsymbol{z} + U^T\overline{\boldsymbol{z}}\right)$$

from the repeated slot vector $\mathbf{1}_r \otimes \boldsymbol{z}$. After EvalMod, we obtain $\boldsymbol{a}' = (a'_0, \ldots, a'_{n-1})^T$, where $\boldsymbol{a}' \approx [\boldsymbol{a}]_q$. Then SlotToCoeff maps the refreshed coefficients back to the sparse slot encoding by computing

$$\boldsymbol{z}' = U\boldsymbol{a}' = \tau_n(m'),$$

where $m'(Y) = \sum_{i=0}^{n-1} a'_i Y^i$. Therefore the desired output slot vector is $\mathbf{1}_r \otimes \boldsymbol{z}'$.

We now give the concrete algorithms for the case $n \le r/2$. The main idea is to use the repeated sparse layout $\mathbf{1}_r \otimes \boldsymbol{z}$ together with the duplication-and-partial-sum trick from Section 1.2. We first describe CoeffToSlot, which produces repeated copies of the desired coefficient entries before EvalMod. We then describe SlotToCoeff, which uses these repeated positions after EvalMod to reconstruct the sparse slot vector $\mathbf{1}_r \otimes \boldsymbol{z}'$.

CoeffToSlot. The case $n \le r/2$ can be handled using essentially the same duplication-and-partial-sum technique introduced in Section 1.2. We write

$$V = \frac{1}{n}\overline{U}^T = (v_{i,j})_{0\le i<n,\ 0\le j<n/2} \in \mathbb{C}^{n\times(n/2)}.$$

Let $\boldsymbol{v}_i^T$ be the i-th row of V, and let $t = r/n \ge 2$. We construct the plaintext vector

$$\boldsymbol{v}_{\mathrm{pack}} = (\mathbf{1}_t \otimes \boldsymbol{v}_0; \mathbf{1}_t \otimes \boldsymbol{v}_1; \ldots; \mathbf{1}_t \otimes \boldsymbol{v}_{n-1}) \in \mathbb{C}^{N/2},$$

where the semicolon denotes block concatenation in the slot vector. Then we compute $\mathsf{ct}' = \mathsf{CMult}(\mathsf{ct}, \boldsymbol{v}_{\mathrm{pack}})$. If ct encrypts the sparsely packed vector $\mathbf{1}_r \otimes \boldsymbol{z}$, then ct' encrypts

$$(\mathbf{1}_t \otimes (\boldsymbol{v}_0 \odot \boldsymbol{z}); \mathbf{1}_t \otimes (\boldsymbol{v}_1 \odot \boldsymbol{z}); \ldots; \mathbf{1}_t \otimes (\boldsymbol{v}_{n-1} \odot \boldsymbol{z})).$$

For each i, let $b_i = \sum_{j=0}^{n/2-1} v_{i,j} z_j$. We then invoke Algorithm 1 on ct' with $\ell = n/2$ and $k = 1$. The resulting ciphertext ct'' encrypts a vector of the form

$$(\mathbf{1}_\kappa \otimes b_0; \boldsymbol{c}_0; \mathbf{1}_\kappa \otimes b_1; \boldsymbol{c}_1; \cdots; \mathbf{1}_\kappa \otimes b_{n-1}; \boldsymbol{c}_{n-1}),$$

where $\kappa = (t-1)n/2$ and $\boldsymbol{c}_i \in \mathbb{C}^{n/2}$ contains slots that will not be used. Finally, we compute $\mathsf{ct}_{\mathrm{out}} = \mathsf{Add}(\mathsf{ct}'', \mathrm{Conj}(\mathsf{ct}''))$. Then $\mathsf{ct}_{\mathrm{out}}$ encrypts

$$(\mathbf{1}_\kappa \otimes a_0; \boldsymbol{d}_0; \mathbf{1}_\kappa \otimes a_1; \boldsymbol{d}_1; \cdots; \mathbf{1}_\kappa \otimes a_{n-1}; \boldsymbol{d}_{n-1}),$$

where $a_i = b_i + \overline{b_i}$ and $\boldsymbol{d}_i = \boldsymbol{c}_i + \overline{\boldsymbol{c}}_i$. Thus, the desired values $a_0, \ldots, a_{n-1}$ appear in the slots of $\mathsf{ct}_{\mathrm{out}}$. We call the entries in the blocks $\boldsymbol{d}_i$ the *auxiliary slots*, since they are not the desired output values but are still carried by the ciphertext. For clarity, we summarize the sparse $\mathsf{CoeffToSlot}$ procedure for the case $n \le r/2$ in Algorithm 2.

In the paper, we fix one convention. In all algorithms below, CMult does not include an automatic $\mathsf{ReScale}$. All plaintext-ciphertext products are first accumulated at the same ciphertext modulus, and the scale factor Δ is removed only by the explicit final RS operation.

Algorithm 2 $\mathsf{SparseCoeffToSlot}_{n \le r/2}(\mathsf{ct}, V)$

Input: A ciphertext ct encrypting $\mathbf{1}_r \otimes \boldsymbol{z}$, where $\boldsymbol{z} \in \mathbb{C}^{n/2}$, and the matrix $V = \frac{1}{n}\overline{U}^T \in \mathbb{C}^{n \times (n/2)}$.

Output: A ciphertext encrypting

$$(\mathbf{1}_\kappa \otimes a_0; \boldsymbol{d}_0; \mathbf{1}_\kappa \otimes a_1; \boldsymbol{d}_1; \cdots; \mathbf{1}_\kappa \otimes a_{n-1}; \boldsymbol{d}_{n-1}),$$

where $\kappa = (t-1)n/2$. Here $\boldsymbol{a} = (a_0, \ldots, a_{n-1})^T$ is the coefficient vector corresponding to $\boldsymbol{z}$, namely

$$\boldsymbol{a} = \frac{1}{n}\left(\overline{U}^T \boldsymbol{z} + U^T \overline{\boldsymbol{z}}\right),$$

or equivalently, $\boldsymbol{z} = U\boldsymbol{a}$. The vectors $\boldsymbol{d}_i \in \mathbb{C}^{n/2}$, for $0 \le i < n$, are auxiliary slots.

1: $t \leftarrow r/n$; $L \leftarrow n/2$;
2: Let $\boldsymbol{v}_i$ be the i-th row of V, for $0 \le i < n$;
3: Construct

$$\boldsymbol{v}_{\mathrm{pack}} = (\mathbf{1}_t \otimes \boldsymbol{v}_0; \mathbf{1}_t \otimes \boldsymbol{v}_1; \ldots; \mathbf{1}_t \otimes \boldsymbol{v}_{n-1})$$

4: $\mathsf{ct}' \leftarrow \mathsf{CMult}(\mathsf{ct}, \boldsymbol{v}_{\mathrm{pack}})$
5: $\mathsf{ct}'' \leftarrow \mathsf{BlockPartialSum}(\mathsf{ct}', 1, L)$
6: $\mathsf{ct}_{\mathrm{out}} \leftarrow \mathsf{Add}(\mathsf{ct}'', \mathrm{Conj}(\mathsf{ct}''))$
7: **return** $\mathsf{RS}(\mathsf{ct}_{\mathrm{out}})$

Remark 1 (Auxiliary-slot convention). Let $L = n/2$. The block partial sum in Algorithm 2 produces $\kappa + 1$ consecutive copies of each desired value a_i. In the output description, however, we write the corresponding segment as

$$(\mathbf{1}_\kappa \otimes a_i;\ \boldsymbol{d}_i), \quad \boldsymbol{d}_i \in \mathbb{C}^{n/2}.$$

Thus the first coordinate of $\boldsymbol{d}_i$ may also be a valid copy of a_i. We intentionally include this copy in the auxiliary block in order to keep the layout aligned with the $n/2$ slot blocks used by the following SlotToCoeff algorithm.

Remark 2 (Auxiliary slots before EvalMod). The ciphertext produced by Algorithm 2 will next be processed by EvalMod. Therefore, all its slots should lie in the approximation range $[-K, K]$, including both the desired values a_i and the auxiliary entries in $\boldsymbol{d}_i$. The size of the desired vector $\boldsymbol{a}$ has been studied in previous CKKS bootstrapping analyses. The new point in our construction is to control the auxiliary blocks $\boldsymbol{d}_i$. If these entries are outside $[-K, K]$, EvalMod may output large values on the corresponding slots, which can later be mixed by linear transforms and amplify the approximation error. We address this issue in Section 3.4.

SlotToCoeff. Suppose that, after EvalMod, the ciphertext ct encrypts

$$(\mathbf{1}_\kappa \otimes a'_0; \boldsymbol{d}'_0; \mathbf{1}_\kappa \otimes a'_1; \boldsymbol{d}'_1; \cdots; \mathbf{1}_\kappa \otimes a'_{n-1}; \boldsymbol{d}'_{n-1}).$$

Our goal is to compute a ciphertext encrypting $\mathbf{1}_r \otimes \boldsymbol{x}$, where $\boldsymbol{x} = U\boldsymbol{a}'$ and $\boldsymbol{a}' = (a'_0, \ldots, a'_{n-1})^T$. Let $\boldsymbol{u}_i$ be the i-th column of U. We construct the plaintext vector

$$\boldsymbol{u}_{\mathrm{pack}} = (\mathbf{1}_{t/2} \otimes \boldsymbol{u}_0; \mathbf{1}_{t/2} \otimes \mathbf{0}_{n/2}; \cdots; \mathbf{1}_{t/2} \otimes \boldsymbol{u}_{n-1}; \mathbf{1}_{t/2} \otimes \mathbf{0}_{n/2}) \in \mathbb{C}^{N/2}.$$

Then we compute $\mathsf{ct}' = \mathsf{CMult}(\mathsf{ct}, \boldsymbol{u}_{\mathrm{pack}})$. Since the nonzero part of each plaintext block is placed only on slots carrying a'_i, the ciphertext ct' encrypts

$$(\mathbf{1}_{t/2} \otimes (a'_0\boldsymbol{u}_0); \mathbf{1}_{t/2} \otimes \mathbf{0}_{n/2}; \cdots; \mathbf{1}_{t/2} \otimes (a'_{n-1}\boldsymbol{u}_{n-1}); \mathbf{1}_{t/2} \otimes \mathbf{0}_{n/2}).$$

We then invoke Algorithm 1 on ct' with block length $r/2$ and window length n. This gives a ciphertext ct'' encrypting

$$(\mathbf{1}_{t/2} \otimes \boldsymbol{x}; \mathbf{1}_{t/2} \otimes \mathbf{0}_{n/2}; \cdots; \mathbf{1}_{t/2} \otimes \boldsymbol{x}; \mathbf{1}_{t/2} \otimes \mathbf{0}_{n/2}).$$

Finally, we compute $\mathsf{ct}_{\mathrm{out}} = \mathsf{Add}\big(\mathsf{ct}'', \mathrm{Rot}\big(\mathsf{ct}'', \frac{r}{4}\big)\big)$. Then $\mathsf{ct}_{\mathrm{out}}$ encrypts $\mathbf{1}_r \otimes \boldsymbol{x}$, as desired. The above sparse SlotToCoeff procedure is summarized in Algorithm 3.

Complexity. Algorithm 2 requires $\log(n/2)$ rotations, one plaintext-ciphertext multiplication and one homomorphic conjugate. Algorithm 3 requires $\log(n) + 1$ rotations and one plaintext-ciphertext multiplication.

3.3 CoeffToSlot and SlotToCoeff for Sparse Packed Ciphertexts when $n > r/2 \geq 2$

In this section, we first use a concrete example to illustrate the main idea of our technique for the case $n > r/2$. This example shows how to extract several diagonal-like plaintext vectors, combine them with rotations, and then recover the sparse CoeffToSlot and SlotToCoeff transforms in the slot domain. For convenience, we present the following procedure at the plaintext-vector level. The corresponding operations are implemented homomorphically on ciphertexts.

Algorithm 3 $\mathsf{SparseSlotToCoeff}_{n\le r/2}(\mathsf{ct}, U)$

Input: A ciphertext ct encrypting

$$(\mathbf{1}_\kappa \otimes a'_0; \boldsymbol{d}'_0; \cdots; \mathbf{1}_\kappa \otimes a'_{n-1}; \boldsymbol{d}'_{n-1}),$$

and a matrix $U = (\boldsymbol{u}_0, \ldots, \boldsymbol{u}_{n-1}) \in \mathbb{C}^{(n/2)\times n}$.

Output: A ciphertext encrypting $\mathbf{1}_r \otimes \boldsymbol{x}$, where $\boldsymbol{x} = U\boldsymbol{a}'$ and $\boldsymbol{a}' = (a'_0, \ldots, a'_{n-1})^T$.

1: $t \leftarrow r/n$;
2: Construct

$$\boldsymbol{u}_{\mathrm{pack}} = (\mathbf{1}_{t/2} \otimes \boldsymbol{u}_0; \mathbf{1}_{t/2} \otimes \mathbf{0}_{n/2}; \cdots; \mathbf{1}_{t/2} \otimes \boldsymbol{u}_{n-1}; \mathbf{1}_{t/2} \otimes \mathbf{0}_{n/2})$$

3: $\mathsf{ct}' \leftarrow \mathsf{CMult}(\mathsf{ct}, \boldsymbol{u}_{\mathrm{pack}})$
4: $\mathsf{ct}'' \leftarrow \mathsf{BlockPartialSum}(\mathsf{ct}', r/2, n)$
5: $\mathsf{ct}_{\mathrm{out}} \leftarrow \mathsf{Add}(\mathsf{ct}'', \mathrm{Rot}(\mathsf{ct}'', r/4))$
6: **return** $\mathsf{RS}(\mathsf{ct}_{\mathrm{out}})$

Example 1. Suppose $N/2 = 32$, $n/2 = 4$ and $r = 8$. Then $\boldsymbol{m} = \mathbf{1}_8 \otimes (z_0, z_1, z_2, z_3)^T$. Let $s = n/(r/2) = 2$. Write $V = \frac{1}{n}\overline{U}^T = (v_{ij}) \in \mathbb{C}^{8\times 4}$. We write $\boldsymbol{m}$ as

$$\begin{aligned}\boldsymbol{m} = (&z_0\ z_1\ z_2\ z_3\ z_0\ z_1\ z_2\ z_3\ z_0\ z_1\ z_2\ z_3\ z_0\ z_1\ z_2\ z_3\\ &z_0\ z_1\ z_2\ z_3\ z_0\ z_1\ z_2\ z_3\ z_0\ z_1\ z_2\ z_3\ z_0\ z_1\ z_2\ z_3)^T.\end{aligned}$$

We construct the plaintext vectors $\boldsymbol{v}_0, \ldots, \boldsymbol{v}_{s-1}$ by a blockwise diagonal extraction of V, as illustrated in Figure 1. More precisely, we first split the rows of V into $r/2$ consecutive row blocks. In this example, $r/2 = 4$ and $s = n/(r/2) = 2$, so each row block is a 2×4 submatrix. From each row block, we extract s cyclic diagonals. For $i = 0, 1$, the i-th extracted diagonal of each row block is repeated twice and then appended, in order, to the plaintext vector $\boldsymbol{v}_i$. Then we have

$$\begin{aligned}\boldsymbol{v}_0 = (&v_{0,0}\ v_{1,1}\ v_{0,2}\ v_{1,3}\ v_{0,0}\ v_{1,1}\ v_{0,2}\ v_{1,3}\ v_{2,0}\ v_{3,1}\ v_{2,2}\ v_{3,3}\ v_{2,0}\ v_{3,1}\ v_{2,2}\ v_{3,3}\\ &v_{4,0}\ v_{5,1}\ v_{4,2}\ v_{5,3}\ v_{4,0}\ v_{5,1}\ v_{4,2}\ v_{5,3}\ v_{6,0}\ v_{7,1}\ v_{6,2}\ v_{7,3}\ v_{6,0}\ v_{7,1}\ v_{6,2}\ v_{7,3})^T\\ \boldsymbol{v}_1 = (&v_{0,1}\ v_{1,2}\ v_{0,3}\ v_{1,0}\ v_{0,1}\ v_{1,2}\ v_{0,3}\ v_{1,0}\ v_{2,1}\ v_{3,2}\ v_{2,3}\ v_{3,0}\ v_{2,1}\ v_{3,2}\ v_{2,3}\ v_{3,0}\\ &v_{4,1}\ v_{5,2}\ v_{4,3}\ v_{5,0}\ v_{4,1}\ v_{5,2}\ v_{4,3}\ v_{5,0}\ v_{6,1}\ v_{7,2}\ v_{6,3}\ v_{7,0}\ v_{6,1}\ v_{7,2}\ v_{6,3}\ v_{7,0})^T\end{aligned}$$

We compute $\boldsymbol{y}_0 = \boldsymbol{v}_0 \odot \boldsymbol{m} + \boldsymbol{v}_1 \odot \mathrm{Rot}(\boldsymbol{m}, 1)$, and then set

$$\boldsymbol{y}_1 = \boldsymbol{y}_0 + \mathrm{Rot}(\boldsymbol{y}_0, s) = \boldsymbol{y}_0 + \mathrm{Rot}(\boldsymbol{y}_0, 2).$$

Then $\boldsymbol{y}_1$ has the form

$$\begin{aligned}\boldsymbol{y}_1 = (&b_0\ b_1\ b_0\ b_1\ b_0\ b_1\ *\ *\ b_2\ b_3\ b_2\ b_3\ b_2\ b_3\ *\ *\\ &b_4\ b_5\ b_4\ b_5\ b_4\ b_5\ *\ *\ b_6\ b_7\ b_6\ b_7\ b_6\ b_7\ *\ *)^T,\end{aligned}$$

where the entries marked by $*$ are auxiliary slots, which will be discussed in Section 3.4. Next, we compute $\boldsymbol{y}_1 + \mathrm{Conj}(\boldsymbol{y}_1)$ and obtain

$$\begin{aligned}\boldsymbol{y}_1 + \mathrm{Conjugate}(\boldsymbol{y}_1) = (&a_0\ a_1\ a_0\ a_1\ a_0\ a_1\ *\ *\ a_2\ a_3\ a_2\ a_3\ a_2\ a_3\ *\ *\\ &a_4\ a_5\ a_4\ a_5\ a_4\ a_5\ *\ *\ a_6\ a_7\ a_6\ a_7\ a_6\ a_7\ *\ *)^T.\end{aligned}$$

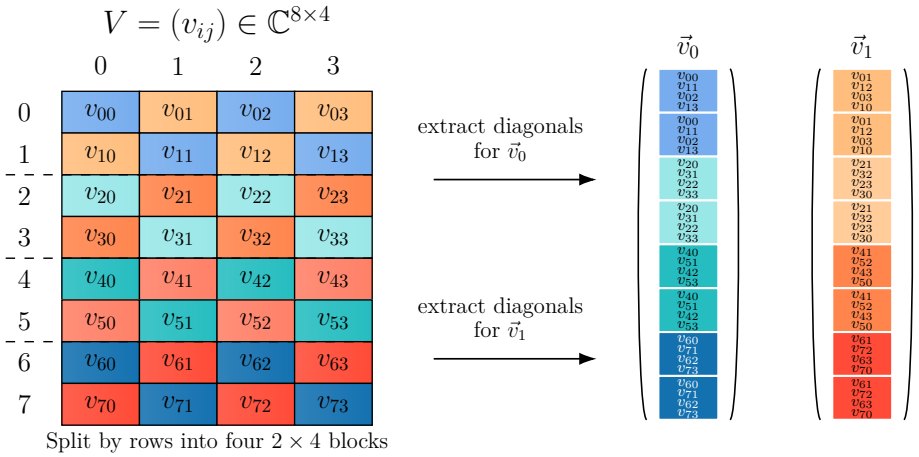

Fig. 1. Diagonal extraction of $V \in \mathbb{C}^{8\times 4}$ in the sparse CoeffToSlot example.

In this way, we compute $\boldsymbol{a} = \frac{1}{n}(\overline{U}^T \boldsymbol{z} + U^T \overline{\boldsymbol{z}})$ in the slots using two plaintext-ciphertext multiplications, 2 rotations, one conjugation.

We then perform modular reduction slotwise. Suppose the resulting vector is

$$\boldsymbol{g} = (a'_0\ a'_1\ a'_0\ a'_1\ a'_0\ a'_1 * *\ a'_2\ a'_3\ a'_2\ a'_3\ a'_2\ a'_3 * *\\ a'_4\ a'_5\ a'_4\ a'_5\ a'_4\ a'_5 * *\ a'_6\ a'_7\ a'_6\ a'_7\ a'_6\ a'_7 * *)^T.$$

For SlotToCoeff, define

$$\boldsymbol{u}_0 = (u_{0,0}\ u_{1,1}\ u_{2,0}\ u_{3,1}\ 0\ 0\ 0\ 0\ u_{0,2}\ u_{1,3}\ u_{2,2}\ u_{3,3}\ 0\ 0\ 0\ 0\\ u_{0,4}\ u_{1,5}\ u_{2,4}\ u_{3,5}\ 0\ 0\ 0\ 0\ u_{0,6}\ u_{1,7}\ u_{2,6}\ u_{3,7}\ 0\ 0\ 0\ 0)^T.$$
$$\boldsymbol{u}_1 = (u_{0,1}\ u_{1,0}\ u_{2,1}\ u_{3,0}\ 0\ 0\ 0\ 0\ u_{0,3}\ u_{1,2}\ u_{2,3}\ u_{3,2}\ 0\ 0\ 0\ 0\\ u_{0,5}\ u_{1,4}\ u_{2,5}\ u_{3,4}\ 0\ 0\ 0\ 0\ u_{0,7}\ u_{1,6}\ u_{2,7}\ u_{3,6}\ 0\ 0\ 0\ 0)^T.$$

Let $\boldsymbol{h}_0 = \boldsymbol{g} \odot \boldsymbol{u}_0 + \mathrm{Rot}(\boldsymbol{g}, 1) \odot \boldsymbol{u}_1$. We compute

$$\boldsymbol{h}_1 = \mathrm{Rot}(\boldsymbol{h}_0, n) + \boldsymbol{h}_0, \text{ and } \boldsymbol{h}_2 = \mathrm{Rot}(\boldsymbol{h}_1, 2n) + \boldsymbol{h}_1.$$

Finally, let $\boldsymbol{h}_{\mathrm{out}} = \mathrm{Rot}(\boldsymbol{h}_2, n/2) + \boldsymbol{h}_2$. Then we obtain

$$\boldsymbol{h}_{\mathrm{out}} = \mathbf{1}_8 \otimes \boldsymbol{z}', \qquad \boldsymbol{z}' = U(a'_0, a'_1, \ldots, a'_7)^T.$$

Thus SlotToCoeff in this example requires two plaintext-ciphertexts multiplications and $4 = s - 1 + \log(r)$ rotations.

We now generalize the above example to arbitrary parameters with $n > r/2$. The construction has two parts. We first specify how to extract the diagonal-like

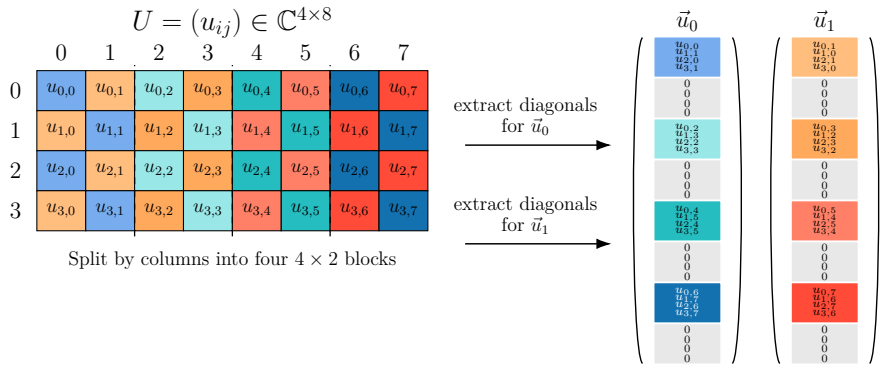

Fig. 2. Diagonal extraction of $U \in \mathbb{C}^{4\times 8}$ by column blocks.

plaintext vectors from V and U, and then give the corresponding CoeffToSlot and SlotToCoeff algorithms.

Let $s = \frac{n}{r/2} = \frac{2n}{r} = \frac{2n^2}{N}$. For CoeffToSlot, we split the rows of $V = (v_{i,j}) \in \mathbb{C}^{n\times(n/2)}$ into $r/2$ consecutive row blocks, each containing s rows. For $0 \le i < s$, we prepare a plaintext vector $\boldsymbol{p}_i \in \mathbb{C}^{N/2}$ by extracting the i-th cyclic diagonal from each row block. More precisely, from the α-th row block, where $0 \le \alpha < r/2$, we define

$$\boldsymbol{p}_{i,\alpha} = \left(v_{\alpha s+(j \bmod s),\,(i+j) \bmod (n/2)}\right)_{0\le j<n/2} \in \mathbb{C}^{n/2}.$$

We then duplicate each extracted diagonal and append them in row-block order:

$$\boldsymbol{p}_i = (\boldsymbol{p}_{i,0}; \boldsymbol{p}_{i,0}; \boldsymbol{p}_{i,1}; \boldsymbol{p}_{i,1}; \cdots; \boldsymbol{p}_{i,r/2-1}; \boldsymbol{p}_{i,r/2-1}).$$

This is the general form of the diagonal extraction illustrated in Figure 1.

For SlotToCoeff, we use an analogous extraction from the column blocks of $U = (u_{j,i}) \in \mathbb{C}^{(n/2)\times n}$. We split the columns of U into $r/2$ consecutive column blocks, each containing s columns. For $0 \le i < s$, we precompute a plaintext vector $\boldsymbol{q}_i \in \mathbb{C}^{N/2}$. From the α-th column block, where $0 \le \alpha < r/2$, we define

$$\boldsymbol{q}_{i,\alpha} = \left(u_{j,\,\alpha s+((j+i) \bmod s)}\right)_{0\le j<n/2} \in \mathbb{C}^{n/2}.$$

We then insert a zero block after each extracted diagonal:

$$\boldsymbol{q}_i = (\boldsymbol{q}_{i,0}; \mathbf{0}_{n/2}; \boldsymbol{q}_{i,1}; \mathbf{0}_{n/2}; \cdots; \boldsymbol{q}_{i,r/2-1}; \mathbf{0}_{n/2}).$$

The zero blocks match the slot layout after EvalMod, as shown in Figure 2.

With these precomputed plaintext vectors, the online homomorphic computation is simple. The sparse CoeffToSlot algorithm first combines s rotated copies of the input ciphertext with the vectors $\boldsymbol{p}_i$, then applies a block partial sum, and finally adds the conjugate. The procedure is summarized in Algorithm 4.

Algorithm 4 $\mathsf{SparseCoeffToSlot}_{n>r/2}(\mathsf{ct}, \{\boldsymbol{p}_i\}_{i=0}^{s-1})$

Input: A ciphertext ct encrypting $\mathbf{1}_r \otimes \boldsymbol{z}$, where $\boldsymbol{z} \in \mathbb{C}^{n/2}$, and precomputed plaintext vectors $\boldsymbol{p}_0, \ldots, \boldsymbol{p}_{s-1}$.

Output: A ciphertext encrypting

$$(\mathbf{1}_{r/4+1} \otimes \boldsymbol{a}_0; \boldsymbol{d}_0; \mathbf{1}_{r/4+1} \otimes \boldsymbol{a}_1; \boldsymbol{d}_1; \cdots; \mathbf{1}_{r/4+1} \otimes \boldsymbol{a}_{r/2-1}; \boldsymbol{d}_{r/2-1}),$$

where, for $0 \le \alpha < r/2$, $\boldsymbol{a}_\alpha = (a_{\alpha s}, \ldots, a_{\alpha s+s-1})^T$. Here $\boldsymbol{a} = (a_0, \ldots, a_{n-1})^T$ is the coefficient vector corresponding to $\boldsymbol{z}$, namely

$$\boldsymbol{a} = \frac{1}{n}\left(\overline{U}^T \boldsymbol{z} + U^T \overline{\boldsymbol{z}}\right),$$

or equivalently, $\boldsymbol{z} = U\boldsymbol{a}$. The vectors $\boldsymbol{d}_\alpha \in \mathbb{C}^{(r/4-1)s}$ are auxiliary slots.

1: $\mathsf{ct}_0 \leftarrow 0$
2: **for** $i = 0$ to $s-1$ **do**
3: $\quad \mathsf{ct}_0 \leftarrow \mathsf{ct}_0 + \mathsf{CMult}(\mathrm{Rot}(\mathsf{ct}, i), \boldsymbol{p}_i)$
4: **end for**
5: $\mathsf{ct}_1 \leftarrow \mathsf{BlockPartialSum}(\mathsf{ct}_0, s, r/4)$
6: $\mathsf{ct}_{\mathrm{out}} \leftarrow \mathsf{Add}(\mathsf{ct}_1, \mathrm{Conj}(\mathsf{ct}_1))$
7: **return** $\mathsf{RS}(\mathsf{ct}_{\mathrm{out}})$

Similarly, the sparse SlotToCoeff algorithm combines s rotated copies of the post-EvalMod ciphertext with the vectors $\boldsymbol{q}_i$. It then sums the $r/2$ column-block contributions and uses one final rotation-and-add to obtain the repeated sparse output. The procedure is given in Algorithm 5.

Algorithm 5 $\mathsf{SparseSlotToCoeff}_{n>r/2}(\mathsf{ct}, \{\boldsymbol{q}_i\}_{i=0}^{s-1})$

Input: A ciphertext ct encrypting

$$(\mathbf{1}_{r/4+1} \otimes \boldsymbol{a}'_0; \boldsymbol{d}'_0; \mathbf{1}_{r/4+1} \otimes \boldsymbol{a}'_1; \boldsymbol{d}'_1; \cdots; \mathbf{1}_{r/4+1} \otimes \boldsymbol{a}'_{r/2-1}; \boldsymbol{d}'_{r/2-1}),$$

where $\boldsymbol{a}'_\alpha = (a'_{\alpha s}, \ldots, a'_{\alpha s+s-1})^T$, and precomputed plaintext vectors $\boldsymbol{q}_0, \ldots, \boldsymbol{q}_{s-1}$.

Output: A ciphertext encrypting $\mathbf{1}_r \otimes \boldsymbol{x}$, where $\boldsymbol{x} = U\boldsymbol{a}'$ and $\boldsymbol{a}' = (a'_0, \ldots, a'_{n-1})^T$.

1: $\mathsf{ct}_0 \leftarrow 0$
2: **for** $i = 0$ to $s-1$ **do**
3: $\quad \mathsf{ct}_0 \leftarrow \mathsf{ct}_0 + \mathsf{CMult}(\mathrm{Rot}(\mathsf{ct}, i), \boldsymbol{q}_i)$
4: **end for**
5: $\mathsf{ct}_1 \leftarrow \mathsf{BlockPartialSum}(\mathsf{ct}_0, n, r/2)$
6: $\mathsf{ct}_{\mathrm{out}} \leftarrow \mathsf{Add}(\mathsf{ct}_1, \mathrm{Rot}(\mathsf{ct}_1, n/2))$
7: **return** $\mathsf{RS}(\mathsf{ct}_{\mathrm{out}})$

Proposition 1. *Algorithm 4 and Algorithm 5 correctly compute the sparse CoeffToSlot and SlotToCoeff specified in their outputs.*

Proof. We argue at the plaintext-slot level. For $\mathsf{CoeffToSlot}$, the vector $\boldsymbol{p}_i$ contains the i-th cyclic diagonal from each s-row block of V, duplicated once. For $0 \le \alpha < r/2$ and $0 \le \beta < r/4$, define

$$\boldsymbol{w}_{\alpha,\beta} = \left(\sum_{i=0}^{s-1} v_{\alpha s+q,\,(\beta s+q+i) \bmod (n/2)} z^{(\beta s+q+i) \bmod (n/2)} \right)_{0 \le q < s} \in \mathbb{C}^s.$$

After multiplying $\mathrm{Rot}(\mathsf{ct}, i)$ by $\boldsymbol{p}_i$ and summing over $i = 0, \ldots, s-1$, the ciphertext ct_0 encrypts a vector of the form

$$\begin{aligned}(&\boldsymbol{w}_{0,0}; \ldots; \boldsymbol{w}_{0,r/4-1}; \boldsymbol{w}_{0,0}; \ldots; \boldsymbol{w}_{0,r/4-1};\\ &\boldsymbol{w}_{1,0}; \ldots; \boldsymbol{w}_{1,r/4-1}; \boldsymbol{w}_{1,0}; \ldots; \boldsymbol{w}_{1,r/4-1}; \ldots;\\ &\boldsymbol{w}_{r/2-1,0}; \ldots; \boldsymbol{w}_{r/2-1,r/4-1}; \boldsymbol{w}_{r/2-1,0}; \ldots; \boldsymbol{w}_{r/2-1,r/4-1}).\end{aligned}$$

For each row block α, the sum of these $r/4$ consecutive s-dimensional blocks is

$$\sum_{\beta=0}^{r/4-1} \boldsymbol{w}_{\alpha,\beta} = (b_{\alpha s}, b_{\alpha s+1}, \ldots, b_{\alpha s+s-1})^T =: \boldsymbol{b}_\alpha, \text{ where } b_i = \sum_{j=0}^{n/2-1} v_{i,j} z_j.$$

Then $\mathsf{BlockPartialSum}(\mathsf{ct}_0, s, r/4)$ sums the corresponding consecutive blocks. Because each sequence $(\boldsymbol{w}_{\alpha,0}; \ldots; \boldsymbol{w}_{\alpha,r/4-1})$ is duplicated once, the resulting ciphertext ct_1 encrypts

$$(\mathbf{1}_{r/4+1} \otimes \boldsymbol{b}_0; \boldsymbol{c}_0; \mathbf{1}_{r/4+1} \otimes \boldsymbol{b}_1; \boldsymbol{c}_1; \cdots; \mathbf{1}_{r/4+1} \otimes \boldsymbol{b}_{r/2-1}; \boldsymbol{c}_{r/2-1}),$$

where $\boldsymbol{c}_\alpha \in \mathbb{C}^{(r/4-1)s}$ consists of auxiliary slots. Adding the conjugate gives

$$(\mathbf{1}_{r/4+1} \otimes \boldsymbol{a}_0; \boldsymbol{d}_0; \mathbf{1}_{r/4+1} \otimes \boldsymbol{a}_1; \boldsymbol{d}_1; \cdots; \mathbf{1}_{r/4+1} \otimes \boldsymbol{a}_{r/2-1}; \boldsymbol{d}_{r/2-1}),$$

where $\boldsymbol{a}_\alpha = \boldsymbol{b}_\alpha + \overline{\boldsymbol{b}_\alpha}$. Equivalently, $a_i = b_i + \overline{b_i}$, and the remaining positions are auxiliary slots.

For $\mathsf{SlotToCoeff}$, the vector $\boldsymbol{q}_i$ is constructed from the i-th cyclic diagonal of each column block of U, with zero blocks inserted to match the post-$\mathsf{EvalMod}$ layout. We again argue at the plaintext-slot level. For each column block $0 \le \alpha < r/2$, define

$$\boldsymbol{x}_\alpha = \sum_{\ell=0}^{s-1} a'_{\alpha s+\ell} \boldsymbol{u}_{\alpha s+\ell} \in \mathbb{C}^{n/2},$$

where $\boldsymbol{u}_j$ is the j-th column of U. Then $\boldsymbol{x} = \sum_{\alpha=0}^{r/2-1} \boldsymbol{x}_\alpha = U\boldsymbol{a}'$. After the first loop of Algorithm 5, the ciphertext ct_0 encrypts a vector of the form

$$(\boldsymbol{x}_0; \mathbf{0}_{n/2}; \boldsymbol{x}_1; \mathbf{0}_{n/2}; \cdots; \boldsymbol{x}_{r/2-1}; \mathbf{0}_{n/2}).$$

The call $\mathsf{BlockPartialSum}(\mathsf{ct}_0, n, r/2)$ then sums the $r/2$ segments of length n. Hence the resulting ciphertext ct_1 encrypts

$$(\boldsymbol{x}; \mathbf{0}_{n/2}; \boldsymbol{x}; \mathbf{0}_{n/2}; \cdots; \boldsymbol{x}; \mathbf{0}_{n/2}),$$

where the pair $(\boldsymbol{x}; \mathbf{0}_{n/2})$ is repeated $r/2$ times. Finally, the rotation by $n/2$ moves each copy of $\boldsymbol{x}$ into the zero half of the same segment, so $\mathsf{ct}_1 + \mathrm{Rot}(\mathsf{ct}_1, n/2)$ encrypts $(\boldsymbol{x}; \boldsymbol{x}; \boldsymbol{x}; \boldsymbol{x}; \cdots; \boldsymbol{x}) = \mathbf{1}_r \otimes \boldsymbol{x}$. This proves the correctness of the sparse SlotToCoeff algorithm.

Complexity. The online cost of Algorithm 4 is s plaintext-ciphertext multiplications, $s - 1 + \log_2(r/4)$ rotations, and one conjugation. The online cost of Algorithm 5 is s plaintext-ciphertext multiplications and $s-1+\log_2 r$ rotations.

Moreover, the first summation in both algorithms can be implemented with a baby-step giant-step strategy. If we choose a baby-step size b, then the $s-1$ rotations in the first loop can be reduced to $(b-1)+(\lceil s/b \rceil - 1)$ rotations, after precomputing the corresponding rotated plaintext vectors. In particular, taking $b \approx \sqrt{s}$ reduces this part of the rotation cost to $O(\sqrt{s})$, while the number of plaintext-ciphertext multiplications remains s. Thus the rotation costs become $O(\sqrt{s}) + \log_2(r/4)$ for CoeffToSlot, and $O(\sqrt{s}) + \log_2 r$ for SlotToCoeff.

3.4 Range of Auxiliary Slots and the Choice of K for EvalMod

The goal of this subsection is to justify that the auxiliary slots produced by our sparse CoeffToSlot algorithms do not create an additional range problem for EvalMod. In CKKS bootstrapping, the modular reduction polynomial is only accurate on a prescribed interval, usually denoted by $[-K, K]$ after normalization. Therefore, not only the desired coefficient slots, but also the auxiliary slots that remain in the ciphertext, should stay in this interval. We now explain the normalization and the probabilistic model used to choose K.

Suppose that $\mathsf{ct} = (c_0(X), c_1(X)) \in (\mathbb{Z}_q[X]/(X^N+1))^2$ is a CKKS ciphertext encrypting a sparse plaintext polynomial $m_0(Y) \in \mathbb{Z}_q[Y]/(Y^n+1)$, where $Y = X^{N/n}$, under the secret key $s(X)$. We view $m_0(Y)$ as a polynomial in $\mathbb{Z}_q[X]/(X^N+1)$ through the embedding $Y \mapsto X^{N/n}$. For notational consistency, write $m(Y) = m_0(Y)$. After choosing centered integer representatives for the coefficients of c_0, c_1, and m, the decryption relation can be lifted to an equality in $\mathbb{Z}[X]/(X^N+1)$ of the form

$$c_0(X) + c_1(X)s(X) = qI(X) + m(Y) + e(X),$$

where $I(X) = \sum_{i=0}^{N-1} I_i X^i$ is an integer polynomial and $e(X)$ is the CKKS approximation error. Since $e(X)$ is small compared with the main term $qI(X) + m(Y)$, it is absorbed into the usual CKKS approximation error and will not affect the following range estimate.

Suppose that $s(X)$ is a signed binary secret key with exactly h nonzero coefficients. Following the heuristic model used in CKKS bootstrapping analysis [10,4], the centered and normalized coefficients of the decryption value are modeled as sums of $h+1$ independent uniform variables in $(-1/2, 1/2)$. We adopt the same model in this paper.

Let $r = N/n$, and define $\widetilde{I}(Y) = I_0 + I_r Y + \cdots + I_{(n-1)r} Y^{n-1}$. The SubSum step maps $qI(X)+m(Y)$ to a public multiple of $q\widetilde{I}(Y)+m(Y)$. This public factor

can be canceled by rescaling or folded into the following linear transformation. The normalization by $1/q$ is public and can be folded into the plaintext matrices of CoeffToSlot, or equivalently applied as an adjacent plaintext multiplication. Thus, in Algorithm 2 and Algorithm 4, the vector $\boldsymbol{a} = (a_0, \ldots, a_{n-1})^T$ appearing in the output refers to the normalized coefficient vector

$$a(Y) = \frac{q\widetilde{I}(Y) + m(Y)}{q} = \sum_{i=0}^{n-1} a_i Y^i.$$

Under the standard heuristic model used in CKKS bootstrapping, each coefficient a_i is modeled as a sum of $h+1$ independent random variables sampled from $\mathcal{U}(-1/2, 1/2)$. The auxiliary slots produced by our sparse CoeffToSlot algorithms are additional linear combinations generated by the same linear transformations and partial sums, and we analyze their range below.

Proposition 2 (Magnitude of the slots after CoeffToSlot). *Let $U \in \mathbb{C}^{(n/2)\times n}$ be the Fourier-type matrix $U_{j,\ell} = \xi_j^\ell$, and suppose $\boldsymbol{z} = U\boldsymbol{a}$ with $\boldsymbol{a} \in \mathbb{R}^n$. Then the following hold.*

1. *Every slot produced by Algorithm 2 or Algorithm 4 before EvalMod can be written as $\boldsymbol{\alpha}^T\boldsymbol{a}$ for some $\boldsymbol{\alpha} \in \mathbb{R}^n$ satisfying $\|\boldsymbol{\alpha}\|_2 = 1$.*
2. *Assume that each coefficient of $a(Y) = \sum_{\ell=0}^{n-1} a_\ell Y^\ell$ is modeled as $a_\ell = \sum_{\rho=1}^{h+1} X_{\ell,\rho}$ with $X_{\ell,\rho} \leftarrow \mathcal{U}(-1/2, 1/2)$, where all $X_{\ell,\rho}$ are independent. Then every output slot $y = \boldsymbol{\alpha}^T\boldsymbol{a}$ has the tail bound*

$$\Pr[|y| \geq K] \leq 2\exp\left(-\frac{6K^2}{h+1}\right).$$

Proof. The proof is deferred to Appendix B.2.

Consequently, let $\mathcal{S}$ denote the set of all slot positions in the ciphertext produced by CoeffToSlot before EvalMod, including both the desired coefficient slots and the auxiliary slots. For each $\eta \in \mathcal{S}$, let y_η be the corresponding slot value. By the above tail bound, for every $\eta \in \mathcal{S}$,

$$\Pr[|y_\eta| \geq K] \leq 2\exp\left(-\frac{6K^2}{h+1}\right).$$

Applying the union bound, over all slot positions, we obtain

$$\Pr\left[\max_{\eta\in\mathcal{S}} |y_\eta| \geq K\right] \leq \sum_{\eta\in\mathcal{S}} \Pr[|y_\eta| \geq K] \leq 2|\mathcal{S}|\exp\left(-\frac{6K^2}{h+1}\right).$$

This argument does not require the slot values y_η to be independent. Therefore, for any $0 < \delta < 1$, it is enough to choose $K \geq \sqrt{\frac{h+1}{6}\ln\frac{2|\mathcal{S}|}{\delta}}$ to ensure $\Pr\left[\max_{\eta\in\mathcal{S}} |y_\eta| \geq K\right] \leq \delta$.

Since one ciphertext contains $N/2$ slots, we may take $|\mathcal{S}| = N/2$. Hence it is sufficient to choose

$$K \geq \sqrt{\frac{h+1}{6} \ln \frac{N}{\delta}}.$$

With this choice, all slots produced by CoeffToSlot, not only the desired coefficient slots, lie in the EvalMod approximation range $[-K, K]$ with probability at least $1-\delta$. The additional factor caused by the auxiliary slots is only logarithmic in the number of slot positions.

Using this bound, we choose K so that all slots produced by CoeffToSlot lie in the EvalMod approximation range $[-K, K]$.

Instantiation with sparse-secret encapsulation. In our experiments, all variants use OpenFHE's `SPARSE_ENCAPSULATED` setting with encapsulated-secret Hamming weight $\widetilde{h} = 32$, and all end-to-end timings include both encapsulation key switches. To achieve $p_{\text{fail}} < 2^{-128}$, the standard layout uses $K = 16$ under the model of Bossuat et al. [5], whereas the bound above requires $K = 24$ for our layout. To obtain essentially the same output precision as the default OpenFHE CKKS implementation, we increase the degree d of the base Chebyshev polynomial from 32 to 42, while retaining three double-angle iterations. At $n/2 = 64$, the resulting end-to-end average output precision differs by only 0.03 bits, as shown in Table 3 of Section 4. The corresponding EvalMod overhead is reported in Table 1.

Table 1. EvalMod parameters and overhead.

Layout	K	d	R	Mult	CMult	Depth	EvalMod time (s)
Standard	16	32	3	14	26	10	0.9689
Ours	24	42	3	16	36	10	1.1639

* Both rows use the same cryptographic parameters, modulus chain, input level, scaling technique, and threading configuration as Ours in Section 4; only K and d are changed. Here d denotes the degree of the base Chebyshev polynomial, and R is the number of double-angle iterations. Mult and CMult denote ciphertext-ciphertext and ciphertext-by-constant multiplications, respectively.

Thus, increasing (K, d) from $(16, 32)$ to $(24, 42)$ adds two Mult and ten CMult operations and increases the EvalMod time by 0.1950 seconds, without increasing its multiplicative depth. The experiments in Section 4 use these layout-specific, K-minimized EvalMod parameters under sparse-secret encapsulation for all tested sparse dimensions, and report the resulting end-to-end bootstrapping running time, remaining levels, and output precision. Consequently, the reported runtime–level tradeoffs explicitly account for the larger K required by our layout and its associated EvalMod overhead, rather than relying on a fixed-K comparison.

3.5 Error estimation of CoeffToSlot and SlotToCoeff

We analyze the error growth of the proposed sparse linear transforms in the case $n > r/2$, namely Algorithms 4 and 5. This is the case where the online computation uses $s = 2n/r$ diagonal branches followed by block partial sums, and it is the more delicate case for the error analysis. The case $n \le r/2$, corresponding to Algorithms 2 and 3, is handled by the same argument and is slightly simpler, since each transform uses only one plaintext-ciphertext multiplication before the block-sum stage. We therefore omit the details.

Assume that the dropped modulus factor in one linear-transform layer is close to the plaintext scaling factor Δ. All errors below are measured in the decoded slot domain after the final rescaling. Let B_{ks} bound one fresh key-switching error before this rescaling. Thus a rotation or conjugation performed after plaintext multiplication contributes at most B_{ks}/Δ after RS. Let B_{rs} bound the rounding error introduced by the final rescaling.

As in the bootstrapping analysis of Cheon et al. [10], we use the ideal-plaintext convention for public linear transforms: the diagonal plaintexts are encoded with sufficient precision, and their deterministic encoding errors are not included in the stated ciphertext-error bounds. In particular, entries that are zero in the ideal sparse diagonals are treated as exact zeros. This is consistent with our scaling, since the values of magnitude $q/(2\pi)$ occurring around EvalMod are normalized by a factor $\Delta \approx q$, and hence have decoded magnitude about $1/(2\pi)$ after the linear-transform rescaling.

Proposition 3 (Error of sparse CoeffToSlot). *Suppose that the input ciphertext to Algorithm 4 has decoded error at most B_{in} in each slot. Then every desired output slot has error at most*

$$B_{\mathrm{in}} + B_{\mathrm{ks}} + \left(\frac{r}{2} - 1\right)\frac{B_{\mathrm{ks}}}{\Delta} + B_{\mathrm{rs}}.$$

The proof is deferred to Appendix C.1. Compared with the standard diagonal-order CoeffToSlot of Cheon et al. [10], the ideal error propagation is the same. The only sparse-layout term is $(\frac{r}{2} - 1)B_{\mathrm{ks}}/\Delta$, which comes from the post-multiplication block sums and conjugation. We choose Δ, within the available modulus budget, so that this term is negligible compared with B_{rs}.

Proposition 4 (Error of sparse SlotToCoeff). *Suppose that the input ciphertext to Algorithm 5 has decoded error at most B_{in} in each desired coefficient slot, under the above unnormalized* EvalMod *convention. Then every output slot has error at most*

$$nB_{\mathrm{in}} + \left(n - \frac{r}{2}\right)B_{\mathrm{ks}} + (r-1)\frac{B_{\mathrm{ks}}}{\Delta} + B_{\mathrm{rs}}.$$

The proof is deferred to Appendix C.2. Compared with the standard diagonal-order SlotToCoeff of Cheon et al. [10], the dominant ideal accumulation term is still nB_{in}, reflecting the ℓ_∞-operator norm of the unnormalized coefficient-to-slot matrix. The additional sparse-layout contribution is $(r-1)B_{\mathrm{ks}}/\Delta$, coming from post-multiplication block sums. Again, this term is negligible for the same choice of Δ.

4 Experimental Evaluation

We evaluate the proposed sparse CoeffToSlot and SlotToCoeff algorithms inside the CKKS bootstrapping implementation of OpenFHE v1.5.1 [1]. All variants use the same ring dimension, raised-modulus parameters, input scale, and scaling technique. All variants use the same OpenFHE EvalMod implementation and three double-angle iterations, but use layout-dependent values of K and d.

We compare four variants. The baseline $B_{1,1}$ is the OpenFHE sparse bootstrapping baseline in which each transform consumes one level. Our method, denoted by Ours, replaces these two linear transformations with the algorithms proposed in this paper while keeping the same depth $(1, 1)$. Thus, $B_{1,1}$ is the depth-preserving baseline for our method. We also include the OpenFHE baselines $B_{2,2}$ and $B_{3,3}$, in which each linear transformation consumes two and three levels, respectively. These variants follow the collapsed FFT-like realization of CoeffToSlot and SlotToCoeff introduced by Chen et al. [8]. Several butterfly layers of the homomorphic DFT are merged, reducing the numbers of rotations and plaintext-ciphertext multiplications at the cost of additional levels. Consequently, $B_{2,2}$ and $B_{3,3}$ enter EvalMod one and two levels lower, respectively, and leave four and two levels after bootstrapping, compared with six levels for $B_{1,1}$ and Ours. We report the linear-transform depth and the remaining levels separately in Table 2.

Except for the small-regime case of Ours at $n/2 = 64$, all linear transformations are evaluated using BSGS with double hoisting, following Bossuat et al. [4]. Hence the running time is more informative than the raw number of rotations alone, since different rotations have different practical costs.

Implementation and platform. All experiments were conducted on a Windows desktop machine running Microsoft Windows 11 Home, version 10.0.22631, build 22631. The machine was equipped with an Intel(R) Core(TM) i9-14900K processor with 24 physical cores and 32 logical processors, and 31.69 GiB of RAM. It was built from OpenFHE v1.5.1 in release mode with `WITH_OPENMP=ON`. Thus, the reported timings use OpenFHE's internal OpenMP parallelism. Unless otherwise stated, all reported timings are wall-clock online times for one invocation of the corresponding bootstrapping procedure. No application-level parallelization across independent bootstrapping instances was used.

The columns Mod/Sum, CtS, Eval, and StC report the online homomorphic evaluation times of ModRaise/SubSum, CoeffToSlot, EvalMod, and SlotToCoeff, respectively. The columns CtS and StC exclude the offline construction and encoding of the plaintext constants used by the corresponding linear transformations, including the plaintext-side DFT computations; these costs are included in the preprocessing time. The column Boot reports the complete online bootstrapping time and includes both encapsulation key switches. Therefore, Boot is slightly larger than the sum of the four displayed phase times.

[1] Available at https://github.com/xiaopeng-stu/SlotToCoeff-and-CoeffToSlot

Table 2. Parameters used in the experiments for $N = 2^{16}$.

Cryptographic and modulus parameters									
Key mode	h_{main}	h_{eph}	**Depth**	$\log_2 q_0$	$\log_2 q_i$	$\log_2 p_j$	**Digits**	$\log_2(PQ)$	$\log_2 \Delta$
Sparse-encapsulated	192	32	18	60	59	60	3	1543	59

Linear-transform and EvalMod parameters									
Method	K_{EM}	d_{EM}	d_{lin}	**Levels after boot.**	$b(64)$	$b(128)$	$b(256)$	$b(512)$	$b(1024)$
$B_{1,1}$	16	32	$(1,1)$	6	13	16	20	32	32
$B_{2,2}$	16	32	$(2,2)$	4	8	8	8	8	8
$B_{3,3}$	16	32	$(3,3)$	2	4	8	8	8	8
Ours	24	42	$(1,1)$	6	-	2	8	8	16

* All experiments use $N = 2^{16}$, HYBRID key switching with three decomposition digits, security mode `HEStd_128_classic`, preprocessing mode `pre`, and `sparse_encapsulated` keys with $(h_{\text{main}}, h_{\text{eph}}) = (192, 32)$. Here $P = \prod_j p_j$, where each p_j is a 60-bit special prime. The parameters K_{EM} and d_{EM} specify EvalMod, while d_{lin} gives the level budgets of CoeffToSlot and SlotToCoeff. Finally, $b(m)$ denotes the BSGS baby-step size for $n/2 = m$.

Parameter settings. The reported experiments use ring dimension $N = 2^{16}$, hence $N/2 = 2^{15}$ full CKKS slots. We test effective sparse dimensions $n/2 \in \{64, 128, 256, 512, 1024\}$. For $n/2 = 64$, we have $n = 128$ and $r = N/n = 512$, so $n \leq r/2$. This case therefore evaluates the small-regime algorithm. All experiments use sparse-secret encapsulation with main-secret Hamming weight $h = 192$ and encapsulated-secret Hamming weight $\widetilde{h} = 32$.

We require $p_{\text{fail}} < 2^{-128}$. The model of Bossuat et al. [5] gives $K = 16$ for the standard layout used by $B_{1,1}$, $B_{2,2}$, and $B_{3,3}$, whereas the bound in Section 3.4, applied to all $N/2$ desired and auxiliary slots, gives $K = 24$ for our layout. All variants use three double-angle iterations ($R = 3$). To maintain comparable EvalMod precision, the degree d of the base Chebyshev polynomial is set to 32 for the standard layout and 42 for our layout. Thus, the OpenFHE baselines use $(K, d, R) = (16, 32, 3)$, whereas Ours uses $(K, d, R) = (24, 42, 3)$. The corresponding operation counts and isolated same-level EvalMod overhead are reported in Table 1 in Section 3.4. All end-to-end experiments below in Table 3 use these layout-specific, K-minimized parameters. Consequently, all reported end-to-end bootstrapping times already account for the larger K required by our layout and its associated EvalMod overhead.

Experimental results. Table 3 reports the running-time and precision results. The ratios LT/Ours and Boot/Ours use Ours as the denominator under the same key distribution and the same value of $n/2$. Thus, a ratio larger than 1 means that Ours is faster by that factor.

Table 4 reports the operation counts for the two linear transformations. Since BSGS and double hoisting make different rotations have different practical costs, these counts should be read together with the timing results in Table 3. The corresponding single-threaded results are reported in Appendix D.

Discussion. The timing breakdown confirms that the improvement comes from the two linear transformations. The proposed algorithms replace only the two linear transformations. The surrounding ModRaise/SubSum procedures and the OpenFHE EvalMod implementation are retained, while EvalMod uses the layout-specific parameters reported in Table 2. Thus, the end-to-end results already include the additional EvalMod cost caused by the larger range required by our layout. Against $B_{1,1}$, Ours gives a 2.08× to 8.87× linear-transform speedup and a 1.22× to 6.07× full bootstrapping speedup in the sparse-encapsulated setting.

The comparisons with $B_{2,2}$ and $B_{3,3}$ highlight the tradeoff between running time and level consumption. For $64 \leq n/2 \leq 512$, Ours is slightly faster in end-to-end bootstrapping than $B_{2,2}$ (whose end-to-end bootstrapping time is 1.03× to 1.25× that of Ours) while using two fewer levels, and has an end-to-end bootstrapping time comparable to $B_{3,3}$ (whose time is 0.86× to 1.07× that of Ours) while using four fewer levels. At $n/2 = 1024$, Ours remains faster than the depth-1 baseline but is slower than both deeper baselines, while still leaving two and four more levels than $B_{2,2}$ and $B_{3,3}$, respectively.

Table 4 complements the wall-clock timings with operation counts for the two linear transforms. In BSGS, a diagonal index is written as $i = kb + j$, where $0 \leq j < b$, and the baby-step size b controls the tradeoff between reusable baby-step rotations and giant-step rotations. Except for the small-regime case of Ours at $n/2 = 64$, all transforms use double hoisting [4]: the baby-step rotations share the digit decomposition and basis extension of the input ciphertext, and the resulting plaintext products are accumulated in the extended basis before key-switch-down. Accordingly, FastRot counts these hoisted baby-step rotations, whereas GenRot counts giant-step rotations and ordinary EvalRotate-style rotations. Across $n/2 = 64, 128, 256, 512, 1024$, the P-CMult count drops from $128, 256, 512, 1024, 2048$ in $B_{1,1}$ to $2, 4, 16, 64, 256$ in Ours. After this reduction, rotations become the main remaining cost of Ours under the tested parameters. This is consistent with the theoretical reduction in Theorem 1.

Table 3: Experimental results for $N = 2^{16}$.

$n/2$	Method	Time					Ratio		d_{lin}	Bits (Avg)
		Mod/ Sum	CtS	Eval	StC	Boot	LT/ Ours	Boot/ Ours		
$N = 2^{16}$, **sparse-encapsulated**, $h = 192$, $h_{\text{eph}} = 32$										
64	$B_{1,1}$	0.4919	0.8781	0.9765	0.4135	2.8168	2.08×	1.22×	(1, 1)	22.72
	$B_{2,2}$	0.5044	0.6555	0.9295	0.2286	2.3749	1.43×	1.03×	(2, 2)	22.74
	$B_{3,3}$	0.4827	0.6200	0.8117	0.1966	2.1650	1.32×	0.94×	(3, 3)	22.72
	Ours	0.4726	0.4027	1.1587	0.2172	2.3046	1.00×	1.00×	(1, 1)	22.69
128	$B_{1,1}$	0.4628	1.5940	0.9992	0.7588	3.8702	3.53×	1.69×	(1, 1)	22.23
	$B_{2,2}$	0.4584	0.9075	0.9697	0.3216	2.7104	1.85×	1.18×	(2, 2)	22.21
	$B_{3,3}$	0.4524	0.8098	0.8829	0.2576	2.4560	1.60×	1.07×	(3, 3)	22.22
	Ours	0.4278	0.4308	1.1498	0.2348	2.2959	1.00×	1.00×	(1, 1)	22.19
256	$B_{1,1}$	0.4769	3.3757	1.1702	1.6628	6.7512	6.27×	2.82×	(1, 1)	22.71
	$B_{2,2}$	0.4089	1.1370	0.9795	0.4087	2.9909	1.92×	1.25×	(2, 2)	22.72
	$B_{3,3}$	0.4005	0.8510	0.8897	0.2729	2.4684	1.40×	1.03×	(3, 3)	22.72
	Ours	0.3733	0.5247	1.1600	0.2787	2.3935	1.00×	1.00×	(1, 1)	22.69

Continued on next page

Table 3: Experimental results for $N = 2^{16}$, continued.

$n/2$	Method	Time					Ratio		d_{lin}	Bits (Avg)
		Mod/ Sum	CtS	Eval	StC	Boot	LT/ Ours	Boot/ Ours		
512	$B_{1,1}$	0.5349	7.8167	1.4398	4.0038	13.8812	**8.87**×	**4.65**×	$(1,1)$	22.23
	$B_{2,2}$	0.3688	1.5991	1.0063	0.6100	3.6438	1.66×	1.22×	$(2,2)$	22.23
	$B_{3,3}$	0.3518	0.9551	0.9039	0.3083	2.5761	0.95×	0.86×	$(3,3)$	22.23
	Ours	0.3374	0.8757	1.2548	0.4573	2.9821	1.00×	1.00×	$(\mathbf{1},\mathbf{1})$	22.20
1024	$B_{1,1}$ †	0.6821	19.2573	1.4928	10.7603	32.3165	8.31×	6.07×	$(1,1)$	21.73
	$B_{2,2}$	0.3114	2.0721	1.0158	0.7787	4.2380	0.79×	0.80×	$(2,2)$	21.73
	$B_{3,3}$	0.2965	1.2934	0.8955	0.4366	2.9802	0.48×	0.56×	$(3,3)$	21.73
	Ours	0.3134	2.3782	1.3377	1.2323	5.3235	1.00×	1.00×	$(1,1)$	21.70

* All reported times are arithmetic means of wall-clock online times over 50 trials, measured in seconds; setup and key-generation times are excluded. The column Boot reports the mean full online bootstrapping time. We define LT = CtS + StC. The ratios LT/Ours and Boot/Ours use Ours as the denominator for the same value of $n/2$. The column d_{lin} gives the level budgets of CoeffToSlot and SlotToCoeff. Bits(Avg) $= -\log_2(\text{MeanAvgErr})$, where MeanAvgErr is the mean, over the 50 trials, of the per-trial average error over the first $n/2$ effective slots.

** All variants use the OpenFHE EvalMod implementation, but Ours uses $(K, d) = (24, 42)$, whereas $B_{1,1}$, $B_{2,2}$, and $B_{3,3}$ use $(K, d) = (16, 32)$. All four variants allocate the same EvalMod depth of 10. Hence, at the same input level and in isolation, EvalMod for $B_{1,1}$ should be slightly faster than for Ours. However, the reported values are raw pipeline timings and are affected by the system state and memory pressure. Since $B_{1,1}$ has a substantially larger overall memory footprint, this pressure can offset its lower algorithmic cost and make its measured EvalMod time slightly higher than that of Ours.

† Owing to the memory requirement of the $n/2 = 1024$, $B_{1,1}$ case, this row is the arithmetic mean of 50 independent one-run process invocations using OpenFHE's low-memory/release-precomputation path. All other rows were obtained from a single invocation with runs = 50. Averaging the 50 per-run average errors gives MeanAvgErr $= 2.8784 \times 10^{-7}$ and hence Bits(Avg) = 21.73.

5 Conclusion

We presented depth-one algorithms for CoeffToSlot and SlotToCoeff in sparse CKKS bootstrapping. The main idea is to exploit the repeated-slot structure of sparse packing directly, rather than only reducing the effective transform dimension. For an effective $n/2$-slot vector repeated $r = N/n$ times in the full slot space, the proposed transforms reduce the number of plaintext-ciphertext multiplications and rotations while keeping multiplicative depth 1.

The construction only replaces the two linear transformations, while the surrounding ModRaise/SubSum procedures and the OpenFHE EvalMod implementation are retained. We also showed that the auxiliary slots introduced by the new CoeffToSlot layout satisfy the same sub-Gaussian range bound as the desired coefficient slots. Consequently, the additional range required to account for the auxiliary slots grows only logarithmically with the number of slot positions. The error analysis further shows that the additional sparse-layout error terms are suppressed by the plaintext scaling factor after rescaling.

Our OpenFHE implementation confirms that the proposed method substantially accelerates the depth-one sparse linear-transform stage for $N = 2^{16}$ and $64 \le n/2 \le 1024$. For $64 \le n/2 \le 512$, it is also competitive with the depth-two and depth-three OpenFHE baselines while saving two or four levels in the linear-transform stage. When the sparse dimension grows, the repetition factor r

Table 4. Operation-count comparison for the two linear transforms.

$n/2$	Method	Counts			
		P-CMult	**FastRot**	**GenRot**	**TotalRot**
64	$B_{1,1}$	128	24	9	33
	$B_{2,2}$	60	28	5	33
	$B_{3,3}$	42	18	7	25
	Ours	2	0	14	14
128	$B_{1,1}$	256	30	15	45
	$B_{2,2}$	92	28	9	37
	$B_{3,3}$	66	30	7	37
	Ours	4	2	14	16
256	$B_{1,1}$	512	38	25	63
	$B_{2,2}$	124	28	13	41
	$B_{3,3}$	74	34	7	41
	Ours	16	14	12	26
512	$B_{1,1}$	1024	62	31	93
	$B_{2,2}$	188	44	17	61
	$B_{3,3}$	90	42	7	49
	Ours	64	14	16	30
1024	$B_{1,1}$	2048	62	63	125
	$B_{2,2}$	252	28	29	57
	$B_{3,3}$	138	34	15	49
	Ours	256	30	22	52

* Counts are summed over CoeffToSlot and SlotToCoeff. P-CMult denotes plaintext-ciphertext multiplications. FastRot counts hoisted baby-step rotations, while GenRot counts giant-step BSGS rotations and ordinary EvalRotate-style rotations. TotalRot is the sum of FastRot and GenRot. Conjugations are excluded. The counts are deterministic and independent of the secret-key distribution.

becomes smaller and the advantage over depth-three transforms decreases. This suggests that the proposed approach is most useful for sparse or low-dimensional encrypted workloads where both bootstrapping time and the remaining level budget are important.

Acknowledgements

This research was supported by the STU Scientific Research Initiation Grant No. NTF24023T.

References

1. Bae, Y., Cheon, J.H., Cho, W., Kim, J., Kim, T.: META-BTS: Bootstrapping precision beyond the limit. In: Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security. pp. 223–234. ACM (2022). https://doi.org/10.1145/3548606.3560696
2. Bae, Y., Cheon, J.H., Kim, J., Stehlé, D.: Bootstrapping bits with CKKS. In: Advances in Cryptology – EUROCRYPT 2024. pp. 94–123. Springer (2024). https://doi.org/10.1007/978-3-031-58723-8_4
3. Bae, Y., Kim, J., Stehlé, D., Suvanto, E.: Bootstrapping small integers with CKKS. In: Advances in Cryptology – ASIACRYPT 2024. pp. 330–360. Springer (2024). https://doi.org/10.1007/978-981-96-0875-1_11

4. Bossuat, J., Mouchet, C., Troncoso-Pastoriza, J.R., Hubaux, J.: Efficient bootstrapping for approximate homomorphic encryption with non-sparse keys. In: Advances in Cryptology – EUROCRYPT 2021. pp. 587–617. Springer (2021). https://doi.org/10.1007/978-3-030-77870-5_21
5. Bossuat, J.P., Troncoso-Pastoriza, J.R., Hubaux, J.P.: Bootstrapping for approximate homomorphic encryption with negligible failure-probability by using sparse-secret encapsulation. In: Ateniese, G., Venturi, D. (eds.) Applied Cryptography and Network Security. Lecture Notes in Computer Science, vol. 13269, pp. 521–541. Springer, Cham (Jun 2022). https://doi.org/10.1007/978-3-031-09234-3_26
6. Boucheron, S., Lugosi, G., Massart, P.: Concentration Inequalities: A Nonasymptotic Theory of Independence. Oxford University Press (2013). https://doi.org/10.1093/acprof:oso/9780199535255.001.0001
7. Brakerski, Z., Gentry, C., Vaikuntanathan, V.: (leveled) fully homomorphic encryption without bootstrapping. In: Proceedings of the 3rd Innovations in Theoretical Computer Science Conference. pp. 309–325. ITCS '12, ACM (2012). https://doi.org/10.1145/2090236.2090262
8. Chen, H., Chillotti, I., Song, Y.: Improved bootstrapping for approximate homomorphic encryption. In: Advances in Cryptology – EUROCRYPT 2019. pp. 34–54. Lecture Notes in Computer Science, Springer (2019). https://doi.org/10.1007/978-3-030-17656-3_2
9. Cheon, J.H., Han, K., Hhan, M.: Faster homomorphic discrete fourier transforms and improved FHE bootstrapping. IACR Cryptology ePrint Archive **2018**, 1073 (2018), https://eprint.iacr.org/2018/1073
10. Cheon, J.H., Han, K., Kim, A., Kim, M., Song, Y.: Bootstrapping for approximate homomorphic encryption. In: Advances in Cryptology – EUROCRYPT 2018. Lecture Notes in Computer Science, vol. 10820, pp. 360–384. Springer (2018). https://doi.org/10.1007/978-3-319-78381-9_14
11. Cheon, J.H., Hanrot, G., Kim, J., Stehlé, D.: SHIP: A shallow and highly parallelizable CKKS bootstrapping algorithm. In: Advances in Cryptology – EUROCRYPT 2025. pp. 398–428. Springer (2025). https://doi.org/10.1007/978-3-031-91131-6_14
12. Cheon, J.H., Kim, A., Kim, M., Song, Y.: Homomorphic encryption for arithmetic of approximate numbers. In: Advances in Cryptology – ASIACRYPT 2017. Lecture Notes in Computer Science, vol. 10624, pp. 409–437. Springer (2017). https://doi.org/10.1007/978-3-319-70694-8_15
13. Choe, H., Kim, J., Stehlé, D., Suvanto, E.: Leveraging discrete CKKS to bootstrap in high precision. In: Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security. pp. 1083–1097. ACM (2025). https://doi.org/10.1145/3719027.3765080
14. Coron, J., Seuré, T.: PaCo: Bootstrapping for CKKS via partial CoeffToSlot. In: Advances in Cryptology – ASIACRYPT 2025. pp. 36–67. Springer (2025). https://doi.org/10.1007/978-981-95-5122-4_2
15. Fan, J., Vercauteren, F.: Somewhat practical fully homomorphic encryption. IACR Cryptology ePrint Archive **2012**, 144 (2012), https://eprint.iacr.org/2012/144
16. Gentry, C.: A Fully Homomorphic Encryption Scheme. Ph.D. thesis, Stanford University (2009), https://crypto.stanford.edu/craig/craig-thesis.pdf
17. Gilad-Bachrach, R., Dowlin, N., Laine, K., Lauter, K., Naehrig, M., Wernsing, J.: CryptoNets: Applying neural networks to encrypted data with high throughput and accuracy. In: Proceedings of the 33rd International Conference on Machine Learning (ICML 2016). Proceedings of Machine Learning Research, vol. 48, pp. 201–210. PMLR (2016)

18. Halevi, S., Shoup, V.: Algorithms in HElib. In: Advances in Cryptology – CRYPTO 2014. Lecture Notes in Computer Science, vol. 8616, pp. 554–571. Springer (2014). https://doi.org/10.1007/978-3-662-44371-2_31
19. Halevi, S., Shoup, V.: Faster homomorphic linear transformations in HElib. In: Advances in Cryptology – CRYPTO 2018. Lecture Notes in Computer Science, vol. 10991, pp. 93–120. Springer (2018). https://doi.org/10.1007/978-3-319-96884-1_4
20. Han, K., Hhan, M., Cheon, J.H.: Improved homomorphic discrete fourier transforms and FHE bootstrapping. IEEE Access **7**, 57361–57370 (2019). https://doi.org/10.1109/ACCESS.2019.2913850
21. Han, K., Ki, D.: Better bootstrapping for approximate homomorphic encryption. In: Topics in Cryptology – CT-RSA 2020. pp. 364–390. Lecture Notes in Computer Science, Springer (2020). https://doi.org/10.1007/978-3-030-40186-3_16
22. Hoeffding, W.: Probability inequalities for sums of bounded random variables. Journal of the American Statistical Association **58**(301), 13–30 (1963). https://doi.org/10.1080/01621459.1963.10500830
23. Jutla, C.S., Manohar, N.: Sine series approximation of the mod function for bootstrapping of approximate HE. In: Advances in Cryptology – EUROCRYPT 2022. pp. 491–520. Springer (2022). https://doi.org/10.1007/978-3-031-06944-4_17
24. Juvekar, C., Vaikuntanathan, V., Chandrakasan, A.: GAZELLE: A low latency framework for secure neural network inference. In: 27th USENIX Security Symposium (USENIX Security 2018). pp. 1651–1669. USENIX Association (2018), https://www.usenix.org/conference/usenixsecurity18/presentation/juvekar, also available as IACR Cryptology ePrint Archive, Paper 2018/073
25. Kim, J., Noh, T.: Modular reduction in CKKS. IACR Communications in Cryptology **2**(2) (2025). https://doi.org/10.62056/aevur-iuc
26. Kim, J., Cheon, J.H., Yeo, Y.: OverModRaise: Reducing modulus consumption of CKKS bootstrapping. IACR Communications in Cryptology **2**(3), 32 (2025). https://doi.org/10.62056/a3n5qjp10
27. Kim, S., Park, M., Kim, J., Kim, T., Min, C.: EvalRound algorithm in CKKS bootstrapping. In: Advances in Cryptology – ASIACRYPT 2022. pp. 161–187. Springer (2022). https://doi.org/10.1007/978-3-031-22966-4_6
28. Lee, E., Lee, J.W., Lee, J., Kim, Y.S., Kim, Y., No, J.S., Choi, W.: Low-complexity deep convolutional neural networks on fully homomorphic encryption using multiplexed parallel convolutions. In: Proceedings of the 39th International Conference on Machine Learning. Proceedings of Machine Learning Research, vol. 162, pp. 12403–12422. PMLR (2022), https://proceedings.mlr.press/v162/lee22e.html
29. Lee, J.W., Kang, H., Lee, Y., Choi, W., Eom, J., Deryabin, M., Lee, E., Lee, J., Yoo, D., Kim, Y.S., No, J.S.: Privacy-preserving machine learning with fully homomorphic encryption for deep neural network. IEEE Access **10**, 30039–30054 (2022). https://doi.org/10.1109/ACCESS.2022.3159694
30. Lee, J., Lee, E., Lee, Y., Kim, Y., No, J.: High-precision bootstrapping of RNS-CKKS homomorphic encryption using optimal minimax polynomial approximation and inverse sine function. In: Advances in Cryptology – EUROCRYPT 2021. pp. 618–647. Springer (2021). https://doi.org/10.1007/978-3-030-77870-5_22
31. Lee, Y., Lee, J., Kim, Y., Kim, Y., No, J., Kang, H.: High-precision bootstrapping for approximate homomorphic encryption by error variance minimization. In: Advances in Cryptology – EUROCRYPT 2022. pp. 551–580. Springer (2022). https://doi.org/10.1007/978-3-031-06944-4_19
32. Panda, S.: Principal component analysis using CKKS homomorphic scheme. In: Dolev, S., Margalit, O., Pinkas, B., Schwarzmann, A. (eds.) Cyber Security Cryp-

tography and Machine Learning. Lecture Notes in Computer Science, vol. 12716, pp. 52–70. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-78086-9_4
33. Seuré, T.: SCORE: A SlotToCoeff optimization for real-vector encryption in CKKS. IACR Cryptology ePrint Archive **2025**, 1983 (2025), https://eprint.iacr.org/2025/1983
34. Sung, H., Seo, S., Kim, T., Min, C.: EvalRound+ bootstrapping and its rigorous analysis for CKKS scheme. IEEE Access **13**, 140847–140866 (2025). https://doi.org/10.1109/ACCESS.2025.3595177
35. Vershynin, R.: High-Dimensional Probability: An Introduction with Applications in Data Science. Cambridge Series in Statistical and Probabilistic Mathematics, Cambridge University Press (2018). https://doi.org/10.1017/9781108231596
36. Wainwright, M.J.: High-Dimensional Statistics: A Non-Asymptotic Viewpoint. Cambridge University Press (2019). https://doi.org/10.1017/9781108627771
37. Yan, L., Zeng, P., Cao, H., Song, P., Wang, M.: Faster bootstrapping for CKKS with less modulus consumption. In: Public-Key Cryptography – PKC 2026. Lecture Notes in Computer Science, vol. 16554, pp. 335–366. Springer (2026). https://doi.org/10.1007/978-3-032-26740-5_11
38. Yi, X., Kaosar, M.G., Paulet, R., Bertino, E.: Single-database private information retrieval from fully homomorphic encryption. IEEE Transactions on Knowledge and Data Engineering **25**(5), 1125–1134 (2013). https://doi.org/10.1109/TKDE.2012.90

Auxiliary Supporting Material

A Details of sparse CKKS bootstrapping

We give the derivation behind the overview in Section 2.2. Throughout this discussion, $\approx$ absorbs the standard CKKS errors from encryption, key switching, rescaling, and homomorphic approximation.

Let $\mathsf{ct} = (c_0, c_1) \in R_q^2$ encrypt the sparse plaintext $m(Y) = \sum_{i=0}^{n-1} m_i Y^i$, where $Y = X^r$ and $r = N/n$. After choosing centered integer representatives for the coefficients of c_0 and c_1, we may write

$$c_0(X) + c_1(X)s(X) = qI(X) + m(Y) + e(X)$$

in $\mathbb{Z}[X]/(X^N + 1)$, for an integer polynomial $I(X)$. Equivalently, modulo q, the ciphertext decrypts approximately to $m(Y)$, while after lifting the same decryption value to modulus Q_L, the overflow term $qI(X)$ becomes visible.

ModRaise and SubSum. The ModRaise step views the same ciphertext under the larger modulus Q_L, so the lifted decryption value is approximately $qI(X)+m(Y)$. Since $m(Y)$ lies in the subring generated by $Y = X^r$, only the coefficients of $I(X)$ at powers $1, Y, \ldots, Y^{n-1}$ are relevant for the following sparse bootstrapping circuit. We therefore define

$$\widetilde{I}(Y) = I_0 + I_r Y + \cdots + I_{(n-1)r} Y^{n-1}.$$

The SubSum operation maps $qI(X)+m(Y)$ to a public multiple of $q\widetilde{I}(Y)+m(Y)$. This public factor is canceled by rescaling or folded into the adjacent plaintext linear transformation. After this normalization, the value entering CoeffToSlot is

$$a(Y) = q\widetilde{I}(Y) + m(Y) = \sum_{i=0}^{n-1} a_i Y^i, \qquad a_i = q\widetilde{I}_i + m_i.$$

Thus the entries a_i are precisely the values on which the modular reduction has to be evaluated.

CoeffToSlot. Let $\boldsymbol{a} = (a_0, \ldots, a_{n-1})^T$, and let $\boldsymbol{z} = \tau_n(a) = U\boldsymbol{a}$ be the corresponding sparse slot vector before the repetition by $\mathbf{1}_r$. The inverse relation is

$$\boldsymbol{a} = \frac{1}{n}\left(\overline{U}^T \boldsymbol{z} + U^T \overline{\boldsymbol{z}}\right).$$

Accordingly, CoeffToSlot homomorphically applies this inverse linear map and produces slots containing approximations of $a_i = q\widetilde{I}_i + m_i$.

EvalMod. The modular reduction step removes the multiples of q. Since $a_i = q\widetilde{I}_i + m_i$, one has $[a_i]_q = m_i$. CKKS bootstrapping replaces the centered reduction by a polynomial approximation to a periodic analytic function. In the original sine-based form,

$$[x]_q \approx \frac{q}{2\pi} \sin\left(\frac{2\pi x}{q}\right)$$

when the centered representative $[x]_q$ is small compared with q. Modern implementations often use Chebyshev or related approximations with full-RNS scale management. Our sparse linear transformations are independent of this choice. After EvalMod, the coefficient slots contain $a'_0, \ldots, a'_{n-1}$, where $a'_i \approx [a_i]_q = m_i$. For correctness, after the public normalization used by the implementation, every slot entering EvalMod must lie in its approximation interval, usually denoted $[-K, K]$. In our sparse layout this condition also applies to the auxiliary slots produced by CoeffToSlot.

SlotToCoeff. Let $\boldsymbol{a}' = (a'_0, \ldots, a'_{n-1})^T$ and define $m'(Y) = \sum_{i=0}^{n-1} a'_i Y^i$. Since $a'_i \approx m_i$, we have $m'(Y) \approx m(Y)$. The final linear transformation computes $\boldsymbol{z}' = U\boldsymbol{a}' = \tau_n(m')$, so the output encrypts an approximation of the repeated sparse slot vector

$$\mathbf{1}_r \otimes \boldsymbol{z}' = \mathbf{1}_r \otimes \tau_n(m') \approx \mathbf{1}_r \otimes \tau_n(m).$$

The output modulus is Q_{L-k}, where k is the number of levels consumed by the bootstrapping circuit.

B Proofs of Subgaussian Range Bound of Auxiliary Slots

B.1 Sub-Gaussian Random Variables

We recall some standard facts about sub-Gaussian random variables. We use the moment-generating-function (MGF) formulation; see, e.g., [35, Chapter 2] and [36, Chapter 2]. A centered real random variable X is called σ^2-sub-Gaussian if, for every $\theta \in \mathbb{R}$,

$$\mathbb{E}\big[e^{\theta X}\big] \le \exp\left(\frac{\sigma^2\theta^2}{2}\right).$$

The parameter σ^2 is called a variance proxy. If X is not centered, we say that X is σ^2-sub-Gaussian if $X - \mathbb{E}[X]$ satisfies the above inequality.

The MGF definition implies the usual Gaussian-type tail bound. Indeed, for any $\theta > 0$, Markov's inequality gives

$$\Pr[X \ge t] = \Pr[e^{\theta X} \ge e^{\theta t}] \le e^{-\theta t}\mathbb{E}[e^{\theta X}] \le \exp\left(\frac{\sigma^2\theta^2}{2} - \theta t\right).$$

Optimizing the right-hand side over $\theta > 0$, namely taking $\theta = t/\sigma^2$, gives

$$\Pr[X \ge t] \le \exp\left(-\frac{t^2}{2\sigma^2}\right).$$

Applying the same argument to $-X$, and then using the union bound, yields

$$\Pr[|X| \geq t] \leq 2\exp\left(-\frac{t^2}{2\sigma^2}\right), \qquad t > 0.$$

This is the standard Chernoff-bound argument; see also [6, Chapter 2].

We will also use the standard closure property under independent sums [35, Chapter 2]. Let $X_1, \ldots, X_m$ be independent centered random variables, where X_i is σ_i^2-sub-Gaussian. Then, for any real coefficients $a_1, \ldots, a_m$, the weighted sum $Y = \sum_{i=1}^m a_i X_i$ is $\sum_{i=1}^m a_i^2\sigma_i^2$-sub-Gaussian. Equivalently,

$$\mathbb{E}[e^{\theta Y}] \leq \exp\left(\frac{\theta^2}{2}\sum_{i=1}^m a_i^2\sigma_i^2\right).$$

This follows directly from independence

$$\mathbb{E}[e^{\theta Y}] = \prod_{i=1}^m \mathbb{E}[e^{\theta a_i X_i}] \leq \prod_{i=1}^m \exp\left(\frac{\theta^2 a_i^2\sigma_i^2}{2}\right).$$

We now record the special case of centered uniform variables. If $X \sim \mathcal{U}(-1/2, 1/2)$, then its moment generating function is explicit

$$\mathbb{E}[e^{\theta X}] = \int_{-1/2}^{1/2} e^{\theta x}\, dx = \frac{\sinh(\theta/2)}{\theta/2},$$

where the value at $\theta = 0$ is understood by continuity. We use the elementary inequality

$$\frac{\sinh x}{x} \leq \exp\left(\frac{x^2}{6}\right), \qquad x \in \mathbb{R},$$

with the left-hand side understood as 1 at $x = 0$. Therefore,

$$\mathbb{E}[e^{\theta X}] = \frac{\sinh(\theta/2)}{\theta/2} \leq \exp\left(\frac{\theta^2}{24}\right).$$

Thus $X \sim \mathcal{U}(-1/2, 1/2)$ is $1/12$-sub-Gaussian. This is sharper than the bound obtained from Hoeffding's lemma for arbitrary random variables supported on $[-1/2, 1/2]$, which would give a $1/4$-sub-Gaussian proxy [22].

The following proposition will be used in Section 3.4.

Proposition 5 (Sub-Gaussian bound for uniform sums). *Let $X_{\ell,\rho}$ be independent random variables sampled from $\mathcal{U}(-1/2, 1/2)$, where $0 \leq \ell < n$ and $1 \leq \rho \leq h+1$. For any $\boldsymbol{\alpha} \in \mathbb{R}^n$, define $Y = \sum_{\ell=0}^{n-1}\sum_{\rho=1}^{h+1} \alpha_\ell X_{\ell,\rho}$. Then, for every $\theta \in \mathbb{R}$,*

$$\mathbb{E}[e^{\theta Y}] \leq \exp\left(\frac{\theta^2(h+1)}{24}\|\boldsymbol{\alpha}\|_2^2\right).$$

Proof. Since $X_{\ell,\rho} \sim \mathcal{U}(-1/2, 1/2)$, the MGF bound in Section B.1 gives

$$\mathbb{E}[e^{\theta\alpha_\ell X_{\ell,\rho}}] \leq \exp\left(\frac{\theta^2\alpha_\ell^2}{24}\right).$$

Using independence, we obtain

$$\mathbb{E}[e^{\theta Y}] = \prod_{\ell=0}^{n-1}\prod_{\rho=1}^{h+1}\mathbb{E}[e^{\theta\alpha_\ell X_{\ell,\rho}}] \leq \prod_{\ell=0}^{n-1}\exp\left(\frac{(h+1)\theta^2\alpha_\ell^2}{24}\right).$$

Thus

$$\mathbb{E}[e^{\theta Y}] \leq \exp\left(\frac{\theta^2(h+1)}{24}\sum_{\ell=0}^{n-1}\alpha_\ell^2\right) = \exp\left(\frac{\theta^2(h+1)}{24}\|\boldsymbol{\alpha}\|_2^2\right).$$

□

B.2 Proof of Proposition 2

Proof. 1. If the output slot y is one of the desired entries a_i, the claim is immediate by taking $\boldsymbol{\alpha} = \boldsymbol{e}_i$, where $\boldsymbol{e}_i \in \mathbb{R}^n$ is the i-th standard basis vector. Then $y = \boldsymbol{e}_i^T\boldsymbol{a}$ and $\|\boldsymbol{e}_i\|_2 = 1$.

It remains to consider an auxiliary output slot y, i.e., a slot contained in one of the auxiliary blocks. Let $\widetilde{y}$ be the value of the same slot just before the final addition with its conjugate. By the construction of Algorithm 2 and Algorithm 4, the value $\widetilde{y}$ is a sum of terms $v_{i,j}z_j$, with exactly one term for each column index j. Hence there exists a map $\pi : \{0, \ldots, n/2-1\} \to \{0, \ldots, n-1\}$ such that

$$\widetilde{y} = \sum_{j=0}^{n/2-1} v_{\pi(j),j}z_j.$$

Set $c = (c_0, \ldots, c_{n/2-1})^T$, where $c_j = v_{\pi(j),j}$. Since $V = \frac{1}{n}\overline{U}^T$, each entry of c has the form $c_j = \frac{1}{n}\overline{\xi_j^{\pi(j)}}$. After adding the conjugate, the output slot is

$$y = c^T\boldsymbol{z} + \overline{c}^T\overline{\boldsymbol{z}} = c^TU\boldsymbol{a} + \overline{c}^T\overline{U}\boldsymbol{a}.$$

Let $W = (U; \overline{U})$ and $\widehat{c} = (c; \overline{c})$. Then $y = \widehat{c}^TW\boldsymbol{a}$, and hence $y = \boldsymbol{\alpha}^T\boldsymbol{a}$ with $\boldsymbol{\alpha} = W^T\widehat{c}$. Moreover, $\boldsymbol{\alpha} \in \mathbb{R}^n$, because its ℓ-th entry is

$$\sum_{j=0}^{n/2-1} c_j\xi_j^\ell + \sum_{j=0}^{n/2-1}\overline{c_j}\,\overline{\xi_j^\ell} = 2\,\mathrm{Re}\left(\sum_{j=0}^{n/2-1} c_j\xi_j^\ell\right).$$

It remains to compute the norm of $\boldsymbol{\alpha}$. Since W is the complete Fourier-type evaluation matrix for Y^n+1, it satisfies $WW^* = nI$, and hence also $\overline{W}W^T = nI$. Therefore,

$$\|\boldsymbol{\alpha}\|_2^2 = \|W^T\widehat{c}\|_2^2 = \widehat{c}^*\overline{W}W^T\widehat{c} = n\|\widehat{c}\|_2^2.$$

Since $|c_j| = 1/n$ for all $0 \le j < n/2$, we have $\|c\|_2^2 = (n/2)/n^2 = 1/(2n)$ and $\|\widehat{c}\|_2^2 = 2\|c\|_2^2 = 1/n$. Thus $\|\boldsymbol{\alpha}\|_2 = 1$. This proves the deterministic claim.

2. For the probabilistic claim, let y be any output slot. By the first part, we can write $y = \boldsymbol{\alpha}^T \boldsymbol{a}$ for some $\boldsymbol{\alpha} \in \mathbb{R}^n$ with $\|\boldsymbol{\alpha}\|_2 = 1$. Under the assumption in the second part, each coefficient of $a(Y)$ can be written as $a_\ell = \sum_{\rho=1}^{h+1} X_{\ell,\rho}$, where the $X_{\ell,\rho}$'s are independent and sampled from $\mathcal{U}(-1/2, 1/2)$. Therefore,

$$y = \boldsymbol{\alpha}^T \boldsymbol{a} = \sum_{\ell=0}^{n-1} \sum_{\rho=1}^{h+1} \alpha_\ell X_{\ell,\rho}.$$

For $X \leftarrow \mathcal{U}(-1/2, 1/2)$, we have the moment-generating-function bound $\mathbb{E}e^{\theta X} \le \exp(\theta^2/24)$. Hence, by independence,

$$\mathbb{E}e^{\theta y} \le \exp\left(\frac{\theta^2(h+1)}{24} \sum_{\ell=0}^{n-1} \alpha_\ell^2\right) = \exp\left(\frac{\theta^2(h+1)}{24}\right),$$

where we used $\|\boldsymbol{\alpha}\|_2 = 1$. Thus y is $(h+1)/12$-sub-Gaussian. Applying the standard sub-Gaussian tail bound gives

$$\Pr[|y| \ge K] \le 2\exp\left(-\frac{6K^2}{h+1}\right).$$

This proves the second claim. □

C Proofs of the Error Estimates

Lemma 1 (Error propagation of BlockPartialSum). *Suppose that the input ciphertext of Algorithm 1 decrypts to* $(\boldsymbol{u}_0 + \boldsymbol{e}_0; \boldsymbol{u}_1 + \boldsymbol{e}_1; \ldots; \boldsymbol{u}_{R-1} + \boldsymbol{e}_{R-1})$*, where the slots are divided into* R *blocks. Let* $\ell = 2^d$*. Then the output decrypts to*

$$(\boldsymbol{s}_0 + \boldsymbol{e}'_0; \boldsymbol{s}_1 + \boldsymbol{e}'_1; \ldots; \boldsymbol{s}_{R-1} + \boldsymbol{e}'_{R-1}),$$

where

$$\boldsymbol{s}_i = \sum_{t=0}^{\ell-1} \boldsymbol{u}_{(i+t) \bmod R}, \qquad \boldsymbol{e}'_i = \sum_{t=0}^{\ell-1} \boldsymbol{e}_{(i+t) \bmod R} + \boldsymbol{e}_{\mathrm{KS},i},$$

and $\|\boldsymbol{e}_{\mathrm{KS},i}\|_\infty \le (\ell-1)B_{\mathrm{ks}}$.

Proof. Let $\boldsymbol{e}_i^{(p)}$ be the error in the i-th block after the p-th iteration, with $\boldsymbol{e}_i^{(0)} = \boldsymbol{e}_i$. In the p-th iteration, Algorithm 1 adds a rotation by 2^p blocks, so

$$\boldsymbol{e}_i^{(p+1)} = \boldsymbol{e}_i^{(p)} + \boldsymbol{e}_{(i+2^p) \bmod R}^{(p)} + \boldsymbol{\nu}_i^{(p)},$$

where $\|\boldsymbol{\nu}_i^{(p)}\|_\infty \le B_{\mathrm{ks}}$. The ideal error part is obtained by the same rotate-and-add recurrence, and after p iterations it is $\sum_{t=0}^{2^p-1} \boldsymbol{e}_{(i+t) \bmod R}$. If C_p bounds the accumulated key-switching error after p iterations, then $C_0 = 0$ and $C_{p+1} \le 2C_p + B_{\mathrm{ks}}$. Hence $C_p \le (2^p - 1)B_{\mathrm{ks}}$. Taking $p = d$ gives the claim.

C.1 Proof of Proposition 3

Proof. Let $s = 2n/r$. The plaintext vectors $\boldsymbol{p}_i$ are extracted from $V = \frac{1}{n}\overline{U}^T$, so their entries have absolute value $1/n$. We estimate all errors before the final rescaling.

After the first loop, each relevant slot is a sum of s branches. Since the plaintext vectors are encoded with scale Δ, the propagated input error is at most $\Delta(s/n)B_{\mathrm{in}}$. The rotations $\mathsf{Rot}(\mathsf{ct}, i)$, $1 \le i < s$, are performed before plaintext multiplication. Their key-switching errors are multiplied by plaintext coefficients of size Δ/n, and hence contribute at most $\Delta((s-1)/n)B_{\mathrm{ks}}$.

The call $\mathsf{BlockPartialSum}(\mathsf{ct}_0, s, r/4)$, followed by the final addition with the conjugate, multiplies these propagated errors by $r/2$. Thus, before rescaling, the propagated input error is at most

$$\frac{r}{2} \cdot \Delta \frac{s}{n} B_{\mathrm{in}} = \Delta B_{\mathrm{in}},$$

and the propagated pre-multiplication key-switching error is at most

$$\frac{r}{2} \cdot \Delta \frac{s-1}{n} B_{\mathrm{ks}} = \Delta \frac{s-1}{s} B_{\mathrm{ks}} \le \Delta B_{\mathrm{ks}}.$$

After RS, these become B_{in} and at most B_{ks}.

It remains to bound the fresh key-switching errors introduced after plaintext multiplication. By Lemma 1, the rotations inside $\mathsf{BlockPartialSum}(\mathsf{ct}_0, s, r/4)$ contribute at most $((r/4)-1)B_{\mathrm{ks}}$ before rescaling. The final conjugation copies this accumulated error once more and introduces one additional key-switching error. Thus the post-multiplication key-switching error before rescaling is at most

$$2\left(\frac{r}{4} - 1\right) B_{\mathrm{ks}} + B_{\mathrm{ks}} = \left(\frac{r}{2} - 1\right) B_{\mathrm{ks}}.$$

After RS, this becomes $(\frac{r}{2} - 1)B_{\mathrm{ks}}/\Delta$. Adding the rescaling error B_{rs} gives

$$|e_i'| \le B_{\mathrm{in}} + B_{\mathrm{ks}} + \left(\frac{r}{2} - 1\right) \frac{B_{\mathrm{ks}}}{\Delta} + B_{\mathrm{rs}}$$

for every desired output slot.

C.2 Proof of Proposition 4

Proof. Let $s = 2n/r$. The plaintext vectors $\boldsymbol{q}_i$ are extracted from the sparse $\mathsf{SlotToCoeff}$ matrices, whose nonzero entries have absolute value 1. We estimate all errors before the final rescaling. As in the main text, the public plaintext vectors are treated as ideal encodings, so the zero entries in the sparse diagonals introduce no additional leakage.

For each of the $r/2$ block positions, the first loop forms a local sum of s desired coefficient-slot branches. Since the plaintext vectors are encoded with scale Δ, the propagated input error in one block is at most $\Delta s B_{\mathrm{in}}$. The rotations

used in this local sum are performed before plaintext multiplication. There are $s-1$ nontrivial such rotations in each block, and their key-switching errors are multiplied by plaintext coefficients of size Δ. Hence they contribute at most

$$\Delta(s-1)B_{\mathrm{ks}}$$

per block before the final rescaling.

Each desired output slot receives contributions from all $r/2$ block positions. Therefore, before rescaling, the propagated input error is at most

$$\frac{r}{2}\cdot \Delta s B_{\mathrm{in}} = \Delta n B_{\mathrm{in}},$$

and the propagated pre-multiplication key-switching error is at most

$$\frac{r}{2}\cdot \Delta(s-1)B_{\mathrm{ks}} = \Delta\left(n-\frac{r}{2}\right)B_{\mathrm{ks}}.$$

After RS, these become nB_{in} and $(n-\frac{r}{2})B_{\mathrm{ks}}$, respectively.

It remains to bound the fresh key-switching errors introduced after plaintext multiplication. By Lemma 1, the post-multiplication block-sum stage in Algorithm 5 contributes at most $(r-1)B_{\mathrm{ks}}$ before rescaling. These errors are introduced after plaintext multiplication, so they are not multiplied by a plaintext factor of scale Δ. Hence, after RS, their contribution is bounded by

$$(r-1)\frac{B_{\mathrm{ks}}}{\Delta}.$$

Adding the final rescaling error B_{rs}, we obtain

$$|e_i'| \le nB_{\mathrm{in}} + \left(n-\frac{r}{2}\right)B_{\mathrm{ks}} + (r-1)\frac{B_{\mathrm{ks}}}{\Delta} + B_{\mathrm{rs}}$$

for every output slot.

D Single-threaded Experiments

Table 5: Single-threaded experimental results for $N = 2^{16}$.

$n/2$	Method	Time					Ratio		d_{lin}	Bits (Avg)
		Mod/ Sum	CtS	Eval	StC	Boot	LT/ Ours	Boot/ Ours		
$N = 2^{16}$, **sparse-encapsulated**, **single-threaded**, $h = 192$, $h_{\text{eph}} = 32$										
64	$B_{1,1}$	1.4932	2.6255	3.1847	1.0682	8.5373	1.96×	1.16×	$(1, 1)$	22.71
	$B_{2,2}$	1.5068	1.8527	2.9681	0.5622	7.0545	1.28×	0.96×	$(2, 2)$	22.73
	$B_{3,3}$	1.4954	1.7123	2.6770	0.4635	6.5094	1.15×	0.89×	$(3, 3)$	22.73
	Ours	1.4578	1.2496	3.8231	0.6356	7.3294	1.00×	1.00×	$(1, 1)$	22.71
128	$B_{1,1}$	1.4196	4.6767	3.3356	1.9391	11.5463	3.35×	1.59×	$(1, 1)$	22.22
	$B_{2,2}$	1.3670	2.4812	2.9805	0.7646	7.7631	1.64×	1.07×	$(2, 2)$	22.21
	$B_{3,3}$	1.3626	2.2221	2.7148	0.6041	7.0707	1.43×	0.97×	$(3, 3)$	22.22
	Ours	1.3013	1.3111	3.8395	0.6629	7.2843	1.00×	1.00×	$(1, 1)$	22.19
256	$B_{1,1}$	1.4541	9.7115	3.8061	4.3332	19.5057	6.20×	2.62×	$(1, 1)$	22.73
	$B_{2,2}$	1.2150	3.0990	3.0270	0.9611	8.4724	1.79×	1.14×	$(2, 2)$	22.73
	$B_{3,3}$	1.2161	2.3812	2.7466	0.6495	7.1638	1.34×	0.96×	$(3, 3)$	22.73
	Ours	1.1599	1.5186	3.8448	0.7484	7.4398	1.00×	1.00×	$(1, 1)$	22.70
512	$B_{1,1}$	1.6603	22.0749	4.7905	10.7589	39.5624	9.77×	4.68×	$(1, 1)$	22.23
	$B_{2,2}$	1.1602	4.9083	3.2780	1.5369	11.0687	1.92×	1.31×	$(2, 2)$	22.23
	$B_{3,3}$	1.0822	2.7943	2.8655	0.7500	7.6696	1.05×	0.91×	$(3, 3)$	22.22
	Ours	1.0206	2.2473	3.9084	1.1140	8.4604	1.00×	1.00×	$(1, 1)$	22.19
1024	$B_{1,1}$ ‡	2.0839	54.6232	6.0543	31.8669	95.0257	**10.99**×	**7.25**×	$(1, 1)$	21.73
	$B_{2,2}$	0.9516	5.8715	3.2360	1.8993	12.1398	0.99×	0.93×	$(2, 2)$	21.73
	$B_{3,3}$	0.9214	3.8241	2.8715	1.0563	8.8549	0.62×	0.68×	$(3, 3)$	21.73
	Ours	0.9105	5.2692	4.1526	2.5993	13.1099	1.00×	1.00×	$(1, 1)$	21.70

* Except for the row marked by ‡, all reported times are arithmetic means of single-threaded wall-clock online times over 50 trials, measured in seconds; setup and key-generation times are excluded. The column Boot reports the mean full online bootstrapping time. We define LT = CtS + StC. The ratios LT/Ours and Boot/Ours use Ours as the denominator for the same value of $n/2$. The column d_{lin} gives the level budgets of CoeffToSlot and SlotToCoeff. Bits(Avg) $= -\log_2(\text{MeanAvgErr})$, where MeanAvgErr is the mean, over the timed trials for that row, of the per-trial average error over the first $n/2$ effective slots.

** All variants use the OpenFHE EvalMod implementation, but Ours uses $(K, d) = (24, 42)$, whereas $B_{1,1}$, $B_{2,2}$, and $B_{3,3}$ use $(K, d) = (16, 32)$. All four variants allocate the same EvalMod depth of 10. Hence, at the same input level and in isolation, EvalMod for $B_{1,1}$ should be slightly faster than for Ours. However, the reported values are raw pipeline timings and can be affected by system state and memory pressure. The $B_{2,2}$ and $B_{3,3}$ variants enter EvalMod one and two levels lower, respectively.

‡ Owing to the memory requirement of the $n/2 = 1024$, $B_{1,1}$ case, this row is the arithmetic mean of five independent one-run process invocations using OpenFHE's low-memory/release-precomputation path. The other rows were obtained from a single invocation with runs = 50. Averaging the five per-run average errors gives MeanAvgErr $= 2.8760 \times 10^{-7}$ and hence Bits(Avg) = 21.73.