Character Block Encodings for Discrete CKKS: Single-Level LUTs and Low-Depth Arithmetic

Jules Dumezy[1] and Elias Suvanto[2]

[1] Université Paris-Saclay, CEA-List, France
jules.dumezy@cea.fr
[2] University of Luxembourg, Luxembourg
elias.suvanto@uni.lu

Abstract. Functional bootstrapping has made discrete computation practical in the Cheon–Kim–Kim–Song (CKKS) scheme, but it fuses four distinct tasks – lookup table (LUT) evaluation, modular reduction, noise cleaning, and ciphertext refreshing – into a single rigid pipeline. As a consequence, a generic LUT over an alphabet of size t costs multiplicative depth proportional to $\log_2 t$ and consumes a large share of the modulus budget during a fixed bootstrapping procedure, invoked each time a LUT evaluation or modular reduction is needed.

We show that this pipeline can be unbundled by changing the representation, rather than optimizing the bootstrapping, through block encodings. A finite-alphabet value is carried across several CKKS slots whose coordinates form a basis of functions on the alphabet, typically the characters of a finite abelian group. In such a basis, every LUT is an affine plaintext map evaluated in a single multiplicative level, with depth independent of t. Modular reduction comes for free: a block encoding cannot represent anything but a residue, so arithmetic modulo t is native. Because the encoded values lie on the unit circle, noise growth is independent of the alphabet size t. In the worst case, it matches the noise growth of standard discrete CKKS on the smallest alphabet $\mathbb{Z}_2$, and in more typical workloads it is linear in the number of operations, exponentially better than discrete CKKS at every $t > 2$. Noise cleaning becomes a constant-depth procedure of at most four levels, because the alphabet-dependent part is an LUT and only a fixed-degree-3 smoothstep is nonlinear. Finally, since LUTs are no longer part of the bootstrapping, refreshing reverts to its classical role as a maintenance operation invoked only to regain multiplicative depth. Any CKKS bootstrapping can be used, rather than a constrained and expensive pipeline.

We instantiate the framework with several block encodings that make modular addition, modular multiplication, xor or min/max possible with a single CKKS multiplication. We use them to build CRT arithmetic over large composite moduli, and finite-state prefix scans for radix addition and subtraction in depth $4 + \lceil \log_2 d \rceil$ and for equality and comparison in depth $3 + \lceil \log_2 d \rceil$ for d radix digits. For example, a 256-bit CRT modular addition or multiplication consumes a single multiplicative level and has a latency of 4.7 ms on a single thread.

Keywords: FHE · Discrete CKKS · Lookup Table · Character Encoding

Table of Contents

1 Introduction

Fully homomorphic encryption (FHE) allows computations to be evaluated directly on encrypted data [Gen09]. Modern FHE schemes offer several complementary plaintext models. BGV and BFV support exact arithmetic over plaintext rings and are natural choices for algebraic workloads with high SIMD throughput [BGV12, Bra12, FV12]. CKKS supports approximate arithmetic over real or complex slots, with similar SIMD capabilities [CKKS17]. In parallel, FHEW and TFHE showed that bootstrapping can serve as a programmable primitive, enabling functional bootstrapping over small plaintext spaces [DM15, CGGI16, CGGI17, CGGI20, CLOT21].

Discrete CKKS reuses the SIMD structure of CKKS for messages that are intended to lie in a finite alphabet. BLEACH introduced cleaning techniques for discrete CKKS computations, in particular for bits [DMPS24]. Subsequent works developed CKKS functional bootstrapping procedures for bits and small integers, allowing an LUT over a small or medium plaintext space to be evaluated in CKKS [BCKS24, BKSS24, AKP25, DAP+26]. These works show that CKKS supports discrete computation well beyond its original approximate-arithmetic setting, and that it is highly competitive even in latency against TFHE, for instance for 256-bit multiplication [Kim25b, CPL26, GZ26].

However, these functional bootstrapping procedures share the same shape and the same cost structure. A generic LUT on an alphabet of size t is an interpolation problem of degree $t-1$, and it is evaluated in a fixed bootstrapping pipeline: a half bootstrapping to raise the modulus and put the overflow in the slots, an EvalMod evaluation through the complex exponential to remove overflow, and the interpolation polynomial itself. This pipeline is convenient because the single, expensive trigonometric step does several jobs at once: it removes overflow, performs the modular reduction the discrete computation needs, and, through Hermite-type zero-derivative conditions, folds in noise cleaning. The price of this convenience is rigidity. LUT evaluation is tied to a specific bootstrapping, the LUT consumes $\lceil \log_2(t-1) \rceil$ multiplicative depth [BKSS24, AKP25] and a large fraction of the refreshed modulus budget, and the approximation of the complex exponential is expensive [DAP+26, Table 2]. Because LUT evaluation and modular reduction are performed inside the functional bootstrapping, a full bootstrapping is invoked at every LUT or modular-reduction site, whether or not the level budget calls for it. In short, LUT evaluation, modular reduction, cleaning, and refreshing are bundled into one modulus-hungry primitive whose full cost is incurred at every LUT.

This paper unbundles that pipeline by changing the representation rather than by further optimizing the bootstrapping. We encode a finite-alphabet value as a block of CKKS slots whose coordinates form a basis of functions on the alphabet, which we call a *block encoding*. As the ciphertext already provides a complete function basis for the alphabet, the discrete structure no longer needs to be manufactured in a bootstrapping, and the four bundled tasks come apart. A unary LUT is an affine plaintext map on the block, evaluated in a single multiplicative level, with depth independent of t. Modular reduction is intrinsic

and free: a block cannot represent anything but a residue. The multiplicative level spent on addition therefore pays for a true modular addition with the mod-t reduction included, not a lazy addition that defers the reduction to a separate step. Noise grows at most 2^k in a chain of k multiplications regardless of t, matching the floor of discrete CKKS noise growth on its smallest alphabet $\mathbb{Z}_2$. In practice, where BLT applications, encoding switches, and multiplications by fresh-noise operands are done, the growth is linear $O(k\varepsilon_0)$. Either way, the alphabet size t no longer enters the rate. Cleaning is a constant-depth procedure, independent of t. And bootstrapping reverts to its classical role: a maintenance operation invoked only to regain multiplicative depth, not a *computational* step, and free to use any CKKS bootstrapping, not a fixed one.

The nonlinear cost has been moved into the representation: block encodings make selected algebraic operations native under CKKS multiplication, and tensor-product features give low-depth multivariate LUTs. Table 1 summarizes the resulting depths and example timings for the primitives of the framework and for representative composite workloads built from them.

Table 1. Depth and example timing for the primitives and composite workloads of the framework, on a single thread. Parameters are chosen for 128 bits of security. All figures are leveled: the latency and amortized time cover one isolated operation and exclude the cost of refreshing the modulus budget, which a CKKS bootstrapping can perform between operations as needed. The upper block lists the three alphabet-dependent primitives, reported at $\log_2 t = 4$, i.e. a 4-bit plaintext moduli. The corresponding like-for-like functional-bootstrapping figures are reported in Table 6. The lower block lists three representative composite workloads.

Operation	$\log_2 N$	Depth	Latency	Amortized
Unary LUT	15	1 level	24.2 ms	22 μs
Bivariate LUT	15	3 levels	324.2 ms	5.1 ms
Cleaning	15	4 levels	123.6 ms	113 μs
256-bit CRT Add/Mul	15	1 level	4.7 ms	1.2 ms
64-bit radix Add	16	9 levels	2.11 s	50.2 ms
64-bit radix Compare	16	8 levels	1.78 s	42.5 ms

1.1 Our Contributions

A block encoding framework for finite alphabets. We represent a value by multiple CKKS slots whose value span contains all complex-valued functions on the alphabet, built from characters of finite abelian groups together with the dual indicator basis. Under this representation, any unary LUT is a linear map and is evaluated with one multiplicative level, independent of the alphabet size, and the group law becomes a CKKS multiplication.

Concrete encodings . The block root-of-unity encoding (BRU) represents residues modulo t by the nontrivial characters of $(\mathbb{Z}_t, +)$, so CKKS multiplication realizes addition modulo t. For prime t, the logarithmic block root-of-unity

encoding (L-BRU) represents $\mathbb{Z}_t^*$ together with an absorbing zero, so CKKS multiplication realizes multiplication modulo t. The absorbing zero is automatic, since the whole character block vanishes. The Walsh–Hadamard encoding (WH) represents $\mathbb{F}_2^k$ by its nontrivial characters, making bitwise `xor` native. The dual indicator encoding (IDCT) gives a convenient basis for equality, finite-state transitions, and cleaning. The thermometer encoding (TH) represents an ordered alphabet by its prefix indicators, making min native under CKKS multiplication and max native via $(a, b) \mapsto a + b - ab$.

Low-depth multivariate LUTs. The same basis viewpoint gives multivariate tables. A bivariate LUT is evaluated by generating mixed tensor features in a single multiplication layer and applying a final block linear transform. More generally, a d-variate LUT is obtained from a balanced product tree of tensor features and costs down to $\lceil \log_2 d \rceil + 1$ levels, at the expense of large tables. Since the output encoding is chosen in the final linear transform, a LUT and a following encoding switch fuse without additional depth.

Constant-depth cleaning. We give a cleaning procedure whose depth does not grow with the alphabet size. The ciphertext is switched linearly to IDCT encoding, the degree-three smoothstep is applied slot-wise, and the result is switched to the desired output encoding: at most four levels in total. The potentially t-dependent work is isolated in the two linear encoding switches, while the only nonlinear step is a fixed polynomial. This contrasts with prior approaches, where cleaning depth grows logarithmically with t.

Leveled, composable operations. LUTs, modular reduction and cleaning are leveled operations that stand on their own. Two properties make them composable without an interleaved (functional) bootstrapping: each costs $O(1)$ levels, independent of t, and because values are encoded on the unit circle, the noise growth is independent of t, bounded by 2^k in the worst case for k consecutive operations, matching discrete CKKS at $\mathbb{Z}_2$ and linear for typical use (Subsection 3.6). Empirically (Figure 3), a chain of operations is bounded by the multiplicative level budget, not by precision. Bootstrapping is therefore needed only to regain depth, and any CKKS bootstrapping – EvalRound+ [SSKM25], minimax [LLL+21], high-precision [CKSS25], SHIP [CHKS25]... – may be used, chosen for the workload rather than fixed by the LUT.

CRT arithmetic over large composite moduli. A modulus $T = t_1 \cdots t_n$ with pairwise coprime factors is represented by independent residue blocks. In BRU, CRT addition is native. In L-BRU over each prime CRT component, multiplication is native. Switching between BRU and L-BRU is a linear map, scheduled to make either addition or multiplication native as needed, and fixed polynomial maps over $\mathbb{Z}_T$ decompose component-wise into unary LUTs.

Finite-state prefix scans for radix arithmetic. Local digit operations emit transition tables in IDCT encoding together with state-indexed output values. Transition composition is a one-level selector operation, and a parallel-prefix network evaluates carries, borrows, equality states, and comparison

states in $\lceil \log_2 d \rceil$ rounds. This gives addition, subtraction, equality, and comparison on d radix digits in depth up to $4 + \lceil \log_2 d \rceil$.

1.2 Related Work

CKKS functional bootstrappings for discrete computation [BKSS24, AKP25] realize an LUT over $\mathbb{Z}_t$ through a bootstrapping, with an additional degree $t-1$ polynomial evaluation acting as both the cleaning and LUT evaluation procedure. We target the same primitive – LUT evaluation – but reach it by changing the encoding rather than tuning the bootstrapping. LUTs become cheap, one level affine plaintext maps. Modular reduction is automatic after each operation, cleaning is constant depth, and the noise growth is exponentially better than those previous approaches.

There are multiple papers building on this functional bootstrapping to build large integers operations. Among CRT-style constructions, Boneh–Kim build a CKKS-based system for arithmetic modulo large prescribed integers using nested residue number systems and homomorphic base conversion, supporting non-smooth moduli with strong performance on arithmetic-heavy workloads [BK25]. Their modular reduction emulates the residue map via CKKS functional bootstrapping inserted between arithmetic operations and is pinned to the identity LUT in that pipeline, leaving no room for general per-residue LUTs. Our CRT construction also uses residue components, but encodes each by a character block: per-residue addition or multiplication becomes CKKS multiplication, with no separate reduction step, and a single-level general-per-residue LUT. Like any RNS-style representation, comparison is not native and is handled separately via radix arithmetic (Section 6).

Among radix-style constructions, Kim's homomorphic integer computer and its faster successor represent integers by radix digits in CKKS for moduli of the form b^k [Kim25a, Kim25b], and Cha et al. reduce exact digit carry to logarithmically many bootstrappings in the number of digits, extending radix arithmetic toward arbitrary moduli and enabling comparison [CPL26]. REFHE instead makes arithmetic modulo 2^n native in a BGV-style plaintext algebra and exposes the bit decomposition during bootstrapping [BFG+25], while Triangle encoding targets SIMD machine-word computation with amortized arithmetic-to-Boolean conversion [GZ26]. These approaches share our use of digit decompositions, but manage carries and reductions through bootstrapping or a dedicated bootstrapping-bound logical mode. Our addition, subtraction, equality, and comparison instead use finite-state transition tables composed by a leveled prefix scan. We design neither a full ALU nor a new bootstrapping procedure, but block encodings that keep operations leveled and efficient.

1.3 Technical Overview

Every construction in the paper rests on one identity. For a finite abelian group $(G, \star)$ with character group $\widehat{G}$, each character satisfies

$$\chi(g \star h) = \chi(g)\chi(h)$$

and the characters span all complex-valued functions on G. Encoding g by the vector $\big(\chi(g)\big)_{\chi\neq 1}$ therefore has two consequences at once. For example, taking $G = (\mathbb{Z}_t, +)$ gives the character vector $(\zeta_t^m, \zeta_t^{2m}, \ldots, \zeta_t^{(t-1)m})$ for $m \in \mathbb{Z}_t$, the BRU encoding used throughout the paper. CKKS slot-wise multiplication evaluates the group law, and any unary map $f : G \to Y$ is the Fourier interpolation

$$f = \sum_{\chi} \widehat{f}(\chi)\chi$$

hence an affine plaintext map on the block, evaluated in one level, independent of the size of G. This is why the four tasks bundled into functional bootstrapping – lookup, modular reduction, cleaning, and modulus refreshing – come apart: each becomes a cheap operation in this representation, and bootstrapping reverts to a depth-maintenance step. Different operations are native in different block encodings, so workloads schedule encoding switches between them, each switch being itself a unary LUT and thus a single level.

The instantiation chooses the group to make a target operation native. $G = (\mathbb{Z}_t, +)$ gives BRU, where complex multiplication is addition mod t; $G = \mathbb{F}_2^k$ gives WH, where complex multiplication is `xor`. The one non-obvious choice is L-BRU: for prime t, fix a generator of $\mathbb{Z}_t^*$ and encode a nonzero residue through the characters of its discrete logarithm, so multiplying blocks adds discrete logs and hence multiplies residues. Zero is the all-zero block, absorbing under complex multiplication exactly as 0 is absorbing mod t, while an extra nonzero-indicator slot separates the value at zero so the block still spans all functions on $\mathbb{Z}_t$. BRU and L-BRU therefore encode the same alphabet $\mathbb{Z}_t$ through different bases, and switching between them is a unary LUT for the identity map.

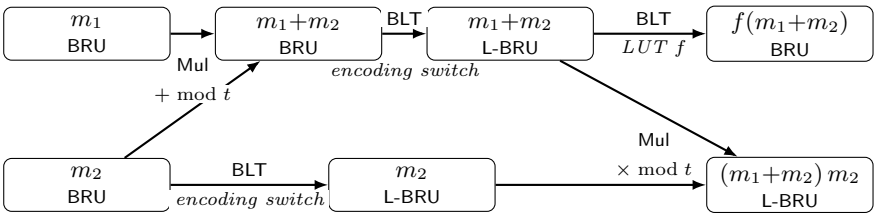

Fig. 1. An example of computation with block encodings. A Mul is a CKKS multiplication of two ciphertexts and evaluates the group law of the current encoding: modular addition in BRU, modular multiplication in L-BRU. A *block linear transform* (BLT) is a one-level affine plaintext map on a block of slots, serving both as an encoding switch and as a unary LUT. Every transition consumes one multiplicative level.

Multivariate LUTs are the tensor-product version of the same argument: the univariate basis functions are already in the input blocks, only the mixed products $\omega_j(x)\eta_k(y)$ are missing, and one multiplication layer generates them before a final linear transform writes any output encoding. A balanced product tree extends this to d inputs in $\lceil \log_2 d \rceil$ layers. Cleaning fits the same picture: in IDCT

a fixed degree-three polynomial reduces noise quadratically, so any encoding is cleaned by an LUT into IDCT, this fixed polynomial, and an LUT back. The only possibly t-dependent work is linear and isolated in the two switches, giving constant depth.

For large moduli, the characters apply per CRT component, making addition native in BRU and multiplication native in L-BRU, with the non-native operation a per-component low-depth LUT. For radix words the obstacle is carry propagation rather than the digit operation: addition, subtraction, and comparison become finite-state scans over a constant-size state space, where composing two transitions is a one-hot selection – evaluated in one level – so a parallel-prefix network resolves all carries in logarithmic depth.

2 Preliminaries

2.1 Notations

We write $\mathbb{N}$, $\mathbb{Z}$, $\mathbb{R}$ and $\mathbb{C}$ for the sets of nonnegative integers, integers, real numbers and complex numbers respectively. Complex conjugation of a complex number $z \in \mathbb{C}$ is denoted by $\bar{z}$. For $x \in \mathbb{R}$, $\lfloor x \rfloor$, $\lceil x \rceil$ and $\lfloor x \rceil$ denotes rounding to the previous, next and nearest integer respectively. Rounding is applied coefficient-wise when the input is a vector or a polynomial.

For an integer $t \geq 2$, we write

$$\mathbb{Z}_t = \mathbb{Z}/t\mathbb{Z}.$$

When an element of $\mathbb{Z}_t$ is used in an order comparison, a carry test, or a borrow test, it is identified with its canonical representative in $\{0, \ldots, t-1\}$.

For integers $a < b$, the interval notation is discrete:

$$[a, b] = \{a, a+1, \ldots, b\}, \qquad [a, b) = \{a, a+1, \ldots, b-1\}$$

and similarly for $(a, b]$ and (a, b). In particular, $[0, t)$ denotes the canonical representation of $\mathbb{Z}_t$. We write $\delta_{x,y}$ for the Kronecker delta, equal to 1 if $x = y$ and to 0 otherwise. More generally, for a predicate P, we write δ_P for its indicator.

Vectors are written in bold lower-case letters, for example $\boldsymbol{v}$. For this vector $\|\boldsymbol{v}\|_\infty$ is the maximum absolute coordinate. The all-ones vector is denoted $\mathbf{1}$, with dimension clear from context. The Hadamard product is denoted by $\odot$. Matrices are written in upper-case letters. The transpose of a matrix A is denoted as $A^\intercal$.

2.2 CKKS

We use the RNS variant of CKKS [CKKS17, CHK+19]. Let N be a power of two and let

$$R = \mathbb{Z}[X]/(X^N + 1), \qquad R_Q = R/QR.$$

For a scale Δ and a multiplicative depth budget L, we use a modulus chain

$$Q_\ell = \prod_{i=0}^{\ell} q_i, \qquad \ell \in [0, L],$$

where the q_i are pairwise coprime primes, $q_0 > \Delta$, and $q_\ell \approx \Delta$ for $\ell \geq 1$. A fresh ciphertext is at level L, and rescaling after a multiplication drops one prime from the chain.

Let ξ be a primitive $2N$-th root of unity. The canonical embedding gives the slot map

$$\tau : R \longrightarrow \mathbb{C}^{N/2}, \qquad P \longmapsto \big(P(\xi^{2j+1})\big)_{j=0}^{N/2-1},$$

where one root from each conjugate pair is chosen. Elements of R are said to be in coefficient representation, while their image under τ is in slot representation. Addition and multiplication in R correspond to component-wise addition and multiplication in the CKKS slots.

At scale Δ, a vector $\boldsymbol{z} \in \mathbb{C}^{N/2}$ is encoded as

$$\mathsf{Ecd}_\Delta(\boldsymbol{z}) = \lfloor \tau^{-1}(\Delta \boldsymbol{z}) \rceil \in R,$$

and a plaintext $m \in R$ is decoded as

$$\mathsf{Dcd}_\Delta(m) = \Delta^{-1}\tau(m).$$

A level-ℓ CKKS ciphertext is a pair $\mathsf{ct} = (b, a) \in R_{Q_\ell}^2$ such that, for secret key s,

$$b + as = m + e \bmod Q_\ell$$

for a small error polynomial e. Decoding gives $\boldsymbol{z} + \Delta^{-1}\tau(e)$ when $m = \mathsf{Ecd}_\Delta(\boldsymbol{z})$.

Homomorphic addition preserves the scaling factor and modulus. Homomorphic multiplication multiplies the encrypted slot values and raises the scaling factor from Δ to Δ^2. After relinearization and rescaling by a prime $q_\ell \approx \Delta$, the output returns to scale close to Δ at level $\ell - 1$, consuming one multiplicative level. CKKS operations and bootstrapping are detailed in Appendix A. CKKS slot-wise multiplication is the Hadamard product and consumes one level

$$\boldsymbol{u}, \boldsymbol{v} \longmapsto \boldsymbol{u} \odot \boldsymbol{v}.$$

3 Block Encodings of Finite Alphabets

We encode elements of a finite alphabet in multiple CKKS slots by evaluating a basis of functions on that alphabet. The main examples are character bases of finite abelian groups, together with the standard indicator basis. Let X be a finite alphabet. Abstractly, a *block encoding* of X is a tuple of functions

$$\boldsymbol{\omega}(x) = \big(\omega_\alpha(x)\big)_{\alpha \in I}$$

such that the augmented family $\{1\} \cup \{\omega_\alpha : \alpha \in I\}$ spans the vector space of complex-valued functions on X. In this work, we mostly use the reduced encoding where $|I| = |X| - 1$, and the augmented family is a basis. Thus, an element in an alphabet of size t is represented as $t - 1$ CKKS slots.

Block Linear Transform. A *block linear transform* (BLT) is an affine map

$$\boldsymbol{\omega} \longmapsto A\boldsymbol{\omega} + \boldsymbol{b}$$

with plaintext coefficients. A BLT is a CKKS linear map on slot blocks. It is evaluated as a sum of plaintext-masked rotations of the input ciphertexts, followed by the addition of a plaintext bias. Evaluating such a map consumes one multiplicative level for plaintext-ciphertext multiplication, as rotations and addition do not consume multiplicative depth. Algorithm 1 gives the diagonal evaluation we use throughout the paper.

Algorithm 1 Diagonal evaluation of a block linear transform

procedure $\mathsf{BLT}(\mathsf{ct} \in R^2_{Q_\ell}, A \in \mathbb{C}^{m\times m}, \boldsymbol{b} \in \mathbb{C}^m)$, where $\mathcal{D} \subseteq \mathbb{Z}_m$ is the set of indices of non-zero diagonals of A and $\tilde{\boldsymbol{v}}_d$ is the d-th diagonal of A placed at the slot positions of every replicated block of the layout, padded with zeros outside the blocks

1: $\mathsf{ct}_{\mathrm{out}} \leftarrow \mathsf{Encode}(\boldsymbol{b})$
2: **for** $d \in \mathcal{D}$ **do**
3: $\quad \mathsf{ct}_d \leftarrow \mathsf{Rotate}(\mathsf{ct}, d)$
4: $\quad \mathsf{ct}_{\mathrm{out}} \leftarrow \mathsf{Add}(\mathsf{ct}_{\mathrm{out}}, \mathsf{MulPlain}(\mathsf{ct}_d, \mathsf{Encode}(\tilde{\boldsymbol{v}}_d)))$
return $\mathsf{ct}_{\mathrm{out}} \in R^2_{Q_{\ell-1}}$

Each non-zero diagonal of A contributes one rotation and one plaintext multiplication. For n_{d} non-zero diagonals the cost is n_{d} rotations and one multiplicative level for plaintext multiplication. A baby-step giant-step factorization of the rotations [HS18] reduces this to $O(\sqrt{n_{\mathrm{d}}})$ rotations at the same multiplicative depth, which we use in the implementation. A rotation by d slots cyclically wraps slots from neighboring blocks into the block being operated on, so each diagonal plaintext $\tilde{\boldsymbol{v}}_d$ is built at full ciphertext size with the diagonal coefficients placed at the slot positions of every replicated block and zeros at every other slot. The MulPlain by $\tilde{\boldsymbol{v}}_d$ therefore both selects the correct diagonal contribution and zeros out the cross-block contamination introduced by the rotation. Subsection 3.1 instantiates the diagonals $\tilde{\boldsymbol{v}}_d$ as Fourier coefficients of f, so the BLT becomes the Fourier-interpolation evaluator of the LUT.

Layout choice A block encoding represents one alphabet value by several CKKS slots, and these slots can be arranged in two ways. In the *packed layout*, all coordinates of a block occupy contiguous slots of a single ciphertext. In the *split layout*, each coordinate is held in its own ciphertext. The two layouts are functionally equivalent and differ only in cost profile. A BLT costs one level in either case, but its rotation schedule differs: in the packed layout it is realized by masked rotations within a ciphertext, whereas in the split layout it becomes plaintext-weighted additions across ciphertexts with no rotations.

The difference matters for operations that combine coordinates from different positions of a block, such as forming the tensor-product features of a multivariate

LUT. In the split layout these cross terms are immediate slot-wise products, while in the packed layout the coordinates must first be aligned by masking and rotation, costing one additional level. A block of $t-1$ coordinates is held across $t-1$ ciphertexts in the split layout, multiplying memory and bandwidth costs by a factor of $t-1$ but freeing the schedule from intra-ciphertext rotations. The packed layout, on the other hand, keeps the ciphertext count low and reduces the number of operations thanks to SIMD, favoring latency rather than throughput. We use the packed layout and explicitly note where the split layout changes the level consumption. Switching between layouts can be achieved using plaintext-ciphertext multiplication and rotations, but is not pursued further in this paper.

3.1 Character Block Encodings

Let G be a finite abelian group, written multiplicatively, and let $\widehat{G}$ be its character group. The *character block encoding* of $g \in G$ is

$$\mathsf{Char}_G(g) = \big(\chi(g)\big)_{\chi \in \widehat{G},\, \chi \neq 1_G}$$

where 1_G is the trivial character. The omitted trivial character is the implicit constant slot. The key identity is $\chi(gh) = \chi(g)\chi(h)$. Therefore, CKKS multiplication evaluates the group law in one level

$$\mathsf{Char}_G(g) \odot \mathsf{Char}_G(h) = \mathsf{Char}_G(gh).$$

The characters of G are orthonormal under the inner product

$$\langle f, h \rangle = \frac{1}{|G|} \sum_{a \in G} f(a)\overline{h(a)}$$

and span the space of functions $G \to \mathbb{C}$. Any $f : G \to \mathbb{C}$ therefore admits a unique Fourier expansion

$$f(g) = \widehat{f}(1_G) + \sum_{\chi \neq 1_G} \widehat{f}(\chi)\chi(g), \qquad \widehat{f}(\chi) = \langle f, \chi \rangle = \frac{1}{|G|} \sum_{a \in G} f(a)\overline{\chi(a)}.$$

The slots of the character encoding are exactly the values $\chi(g)$ for $\chi \neq 1_G$, so evaluating f on the encoding of g is a slot-wise linear combination with coefficients $\widehat{f}(\chi)$ plus the constant bias $\widehat{f}(1_G)$. This is a BLT, so any unary LUT $f : G \to \mathbb{C}$ costs one level. We now describe concrete instantiations of interest.

Block root-of-unity encoding. Taking $G = (\mathbb{Z}_t, +)$, the characters are

$$\chi_k(m) = \zeta_t^{km}, \qquad k \in \mathbb{Z}_t$$

where $\zeta_t = \exp(2\pi i/t)$. Omitting the trivial character $k = 0$ – as it contains no information – gives the *block root-of-unity* encoding (BRU):

$$\mathsf{BRU}_t(m) := \left(\zeta_t^m, \zeta_t^{2m}, \ldots, \zeta_t^{(t-1)m}\right)$$

CKKS multiplication realizes addition modulo t:

$$\zeta_t^{km}\zeta_t^{km'} = \zeta_t^{k(m+m')}.$$

For a unary LUT $f : \mathbb{Z}_t \to \mathbb{C}$, the Fourier expansion is

$$f(m) = \widehat{f}(0) + \sum_{k=1}^{t-1} \widehat{f}(k)\zeta_t^{km}, \qquad \widehat{f}(k) = \frac{1}{t}\sum_{a=0}^{t-1} f(a)\zeta_t^{-ka}.$$

Logarithmic block root-of-unity encoding. Assume t is prime and fix a generator g of $\mathbb{Z}_t^*$. For $m \neq 0$, write $m = g^\ell$ with $\ell = \log_g m \in \mathbb{Z}_{t-1}$. The *logarithmic block root-of-unity* encoding (L-BRU) is the absorbing-zero extension of the character encoding of the multiplicative group $\mathbb{Z}_t^*$:

$$\mathsf{L\text{-}BRU}_t(m) := \begin{cases} \left(1, \zeta_{t-1}^{\ell}, \ldots, \zeta_{t-1}^{(t-2)\ell}\right) & m = g^\ell \neq 0 \\ (0, 0, \ldots, 0) & m = 0 \end{cases}$$

The first slot is the nonzero indicator. These coordinates still span all functions on $\mathbb{Z}_t$: the nonzero indicator separates the value at zero, while the remaining coordinates interpolate with Fourier arbitrary functions on $\mathbb{Z}_t^*$. If both inputs are nonzero, CKKS multiplication adds discrete logs. If either input is zero, the whole vector vanishes. Hence

$$\mathsf{L\text{-}BRU}_t(m) \odot \mathsf{L\text{-}BRU}_t(m') = \mathsf{L\text{-}BRU}_t(mm' \bmod t).$$

A unary LUT $f : \mathbb{Z}_t \to \mathbb{C}$ is obtained by Fourier interpolation on $\mathbb{Z}_t^*$, with the value at zero supplied separately. Define

$$c_k = \frac{1}{t-1}\sum_{\ell=0}^{t-2} f(g^\ell)\zeta_{t-1}^{-k\ell}.$$

Let $\mathsf{L\text{-}BRU}_t(m) = (\eta_0, \ldots, \eta_{t-2})$. Then $f(m)$ equals

$$f(0) + \big(c_0 - f(0)\big)\eta_0 + \sum_{k=1}^{t-2} c_k\eta_k.$$

Walsh–Hadamard encoding. For $t = 2^k$ with $k \in \mathbb{N} \setminus \{0\}$, we may identify the alphabet $\{0, \ldots, t-1\}$ with $\mathbb{F}_2^k$, i.e. vectors $(m_0, \ldots, m_{k-1})$ through the binary expansion

$$m \longleftrightarrow \boldsymbol{m} = (m_0, \ldots, m_{k-1}).$$

The native group law is bitwise `xor`. For $\boldsymbol{s} = (s_0, \ldots, s_{k-1}) \in \mathbb{F}_2^k$, we use the convention

$$\boldsymbol{s} \cdot \boldsymbol{m} = \sum_{j=0}^{k-1} s_j m_j \in \mathbb{F}_2.$$

The character encoding of $(\mathbb{F}_2^k, \oplus)$ is the reduced *Walsh–Hadamard* encoding (WH)

$$\mathsf{WH}_t(m) := \big((-1)^{\boldsymbol{s}\cdot\boldsymbol{m}}\big)_{\boldsymbol{s}\in\mathbb{F}_2^k\setminus\{\mathbf{0}\}}.$$

It has $2^k - 1 = t - 1$ explicit slots, like the other reduced encodings.

CKKS multiplication realizes bitwise `xor`

$$(-1)^{\boldsymbol{s}\cdot\boldsymbol{m}}(-1)^{\boldsymbol{s}\cdot\boldsymbol{m}'} = (-1)^{\boldsymbol{s}\cdot(\boldsymbol{m}\oplus\boldsymbol{m}')}.$$

Unary $\mathbb{F}_2$-linear maps have a particularly simple structure. For $L : \mathbb{F}_2^k \to \mathbb{F}_2^k$

$$(-1)^{\boldsymbol{s}\cdot L\boldsymbol{m}} = (-1)^{(L^\intercal\boldsymbol{s})\cdot\boldsymbol{m}}.$$

Hence, a linear map acts by reindexing character slots, possibly collapsing to the implicit constant slot when $L^\intercal\boldsymbol{s} = \mathbf{0}$.

3.2 Indicator encoding

The indicator encoding is the Fourier dual of the character block encoding on a finite abelian group: the indicator functions $\{\delta_a\}_{a\in G}$ and the characters $\{\chi\}_{\chi\in\widehat{G}}$ are two bases of the same function space $\mathbb{C}^G$, related by the Fourier matrix of G. The construction below applies to any finite alphabet X, with no group structure required. For any t-point alphabet X, choose an omitted value $a_\star \in X$.[3] The reduced dual *indicator* encoding (IDCT) is

$$\mathsf{IDCT}_t(m) := \big(\delta_{m,a}\big), \qquad a \in X \setminus \{a_\star\}.$$

The omitted coordinate is recovered as

$$\delta_{m,a_\star} = 1 - \sum_{a\neq a_\star} \delta_{m,a}.$$

On the non-reduced IDCT encoding, CKKS multiplication is an equality-gated identity: the product of two indicator vectors is the indicator of the common value if the two values are equal, and the zero vector otherwise. Indicator-to-indicator LUTs are realized via a BLT, with coefficients in $\{-1, 0, 1\}$ and biases in $\{0, 1\}$. The omitted coordinate can be a useful degree of freedom: choosing $a_\star$ can reduce the norms of the BLT rows (Subsection 3.5).

3.3 Thermometer encoding

Unlike the previous encodings, the thermometer encoding does not encode a group law: it makes the lattice operations min and max native. For an ordered alphabet $\{0, \ldots, t-1\}$, the reduced *thermometer* encoding (TH) is

$$\mathsf{TH}_t(m) := \big(\delta_{m\geq k}\big)_{k=1}^{t-1} = \big(\delta_{m\geq 1}, \delta_{m\geq 2}, \ldots, \delta_{m\geq t-1}\big),$$

[3] We present the reduced form, omitting one value, so that the BRU, $\mathsf{L\text{-}BRU}$, and IDCT encoding all use the same number of slots. The non-reduced form naturally follows from the reduced one.

the prefix of ones of length m followed by zeros. These $t-1$ coordinates, together with the implicit constant $\delta_{m\geq 0} = 1$, span all functions on the alphabet: the indicators are recovered by the finite difference $\delta_{m,k} = \delta_{m\geq k} - \delta_{m\geq k+1}$, with $\delta_{m\geq t} = 0$. The slots lie in $\{0,1\}$, so $\|\mathsf{TH}_t(m)\|_\infty \leq 1$ as for IDCT.

CKKS multiplication realizes the slot-wise `and`, which is exactly min

$$\delta_{m_1\geq k}\,\delta_{m_2\geq k} = \delta_{\min(m_1,m_2)\geq k}, \qquad \mathsf{TH}_t(m_1)\odot\mathsf{TH}_t(m_2) = \mathsf{TH}_t\big(\min(m_1,m_2)\big).$$

Its De Morgan dual, the slot-wise `or`, is max and is evaluated in one multiplicative level by

$$\delta_{m_1\geq k} + \delta_{m_2\geq k} - \delta_{m_1\geq k}\,\delta_{m_2\geq k} = \delta_{\max(m_1,m_2)\geq k}.$$

Both min and max are therefore native on the same encoding, each in a single level. A unary LUT $f : \{0,\dots,t-1\} \to \mathbb{C}$ is the finite-difference BLT

$$f(m) = f(0) + \sum_{k=1}^{t-1} \big(f(k) - f(k-1)\big)\,\delta_{m\geq k},$$

with bias $f(0)$ and linear coefficients the consecutive differences of f, evaluated in one level.

Table 2. Summary of presented block encodings.

Encoding	CKKS multiplication	Applicability
BRU	$m_1 + m_2 \bmod t$	$t \geq 2$
L-BRU	$m_1 m_2 \bmod t$	t prime
WH	$m_1 \oplus m_2$	$t = 2^k$
IDCT	Equality-gated identity	Any finite alphabet
TH	$\min(m_1, m_2)$ (and max via $a + b - ab$)	Ordered alphabet

3.4 Lookup tables and encoding switches

A unary LUT is a single BLT and can be evaluated with one level. The affine term is precisely the contribution of the implicit constant coordinate. This naturally includes encoding switches: switching from one encoding to another is the unary LUT for the identity map, with the result written in the chosen output encoding. The output encoding of a LUT is chosen in its final BLT. Consequently, a switch after a LUT fuses into the LUT at no additional level. More generally, adjacent BLTs compose into a single BLT. Encoding is, therefore, a scheduling decision.

Bivariate and multivariate LUTs. For clarity, let's consider the BRU case first. Let $f : \mathbb{Z}_t \times \mathbb{Z}_{t'} \to \mathbb{C}$ and write $z = \zeta_t^m$, $z' = \zeta_{t'}^{m'}$. Tensor-product interpolation gives

$$f(m,m') = d_{0,0} + \sum_{a=1}^{t-1} d_{a,0} z^a + \sum_{b=1}^{t'-1} d_{0,b}(z')^b + \sum_{a=1}^{t-1}\sum_{b=1}^{t'-1} d_{a,b} z^a (z')^b.$$

The purely univariate terms are already available from the two input blocks. If each coordinate is stored in its own ciphertext, the cross monomials $z^a(z')^b$ require one multiplicative level, and the final linear combination is one BLT. This gives a two-level bivariate LUT in the split layout. In the packed layout an additional level is required to form the tensor-product features, by masking and rotating the packed coordinates. Thus, in the cost model used in the rest of the paper, a bivariate LUT costs three levels.

The same argument is basis-independent. Let $\boldsymbol{\omega}$ and $\boldsymbol{\eta}$ be reduced block encodings of alphabets X and Y. Since $\{1\} \cup \{\omega_j\}$ spans all functions on X and $\{1\} \cup \{\eta_k\}$ spans all functions on Y, the tensor-product family

$$\{1\} \cup \{\omega_j(x)\}_j \cup \{\eta_k(y)\}_k \cup \{\omega_j(x)\eta_k(y)\}_{j,k}$$

spans all functions on $X \times Y$. The first three families are already present in the input blocks, while the mixed terms are generated by a CKKS multiplication layer after the required packed-coordinate alignment. A final BLT writes the result in the chosen output encoding. Thus, bivariate LUTs cost three levels for any of the reduced encodings above in the packed layout.

More generally, a d-variate LUT is evaluated by building the required tensor-product features with a balanced product tree. The highest-order features have degree d, so this costs at least $\lceil \log_2 d \rceil$ levels, followed by a final BLT. In the packed layout, coordinate alignment adds one level. Thus, for $d \geq 2$, a d-variate LUT costs $\lceil \log_2 d \rceil + 2$ levels in the packed layout, and $\lceil \log_2 d \rceil + 1$ levels in the split layout.

Generalization to BFV/BGV. The block-encoding framework presented so far rests on three structural facts: characters of a finite abelian group form a multiplicative basis of its function space, slot-wise multiplication evaluates the group law, and a BLT realizes any linear map on a block in one multiplicative level. None of these are specific to CKKS. The same constructions – unary LUTs, multivariate LUTs, encoding switches, and the CRT and radix arithmetic of Sections 4 and 6 – port directly to BFV/BGV whenever the plaintext ring contains the relevant roots of unity, with depth and level counts unchanged. Only the precision, noise-growth, and cleaning analyses of Subsections 3.5–3.7 are CKKS-specific. In BFV/BGV plaintext slots carry exact residues, so noise tracking and cleaning are not needed. Explicit construction and analysis is left open as future work.

3.5 Precision and coefficient norms

A BLT $\boldsymbol{\omega} \mapsto A\boldsymbol{\omega} + \boldsymbol{b}$ amplifies the input CKKS error through its linear part A, by an amount that depends on the direction of the error. A worst-case bound is the operator norm of A, the row-sum maximum $\|A\|_{\infty\to\infty}$ for an ℓ_∞ analysis or the spectral norm $\|A\|_2$ for an ℓ_2 analysis. CKKS encryption noise is isotropic: its expected magnitude is the same along every slot direction. The operative estimate is therefore the average $\|A\|_F^2/(t-1)$ over all input directions, with $\|A\|_F$ the Frobenius norm of A. A BLT with large interpolation coefficients loses

about $\log_2(\|A\|_F/\sqrt{t-1})$ bits of precision in expectation, and up to $\log_2 \|A\|_2$ bits in the worst case.

A unary LUT $f : \mathbb{Z}_t \to \mathbb{Z}_t$ in a character basis has BLT matrix $A_f = W^* S_f W$, the value-domain substitution $S_f[y, x] = \delta_{y,f(x)}$ conjugated by the unitary character transform W. The matrix S_f is 0/1 with exactly one nonzero per column, and its spectrum is determined entirely by f.

Proposition 1 (Spectral structure of unary BLTs in a character basis). *For every $f : \mathbb{Z}_t \to \mathbb{Z}_t$, $\|A_f\|_F^2 = t$, A_f is unitary if and only if f is bijective. For non-bijective f,* $\operatorname{rank}(A_f) = |\mathrm{Image}(f)| < t$.

Proof. Unitarity of W makes Frobenius invariant under conjugation, so $\|A_f\|_F = \|S_f\|_F$, and $\|S_f\|_F^2$ equals the number of nonzero entries of S_f, namely t. Unitarity of A_f is equivalent to unitarity of S_f, that is to $S_f S_f^* = I$. The matrix $S_f S_f^*$ is diagonal with $|f^{-1}(y)|$ on diagonal y, so it equals I if and only if every preimage has size 1, that is, f is bijective. The rank is $\operatorname{rank}(S_f) = |\mathrm{Image}(f)|$. □

In the reduced character encoding the BLT acts on $t-1$ slots (the nontrivial characters), with the trivial-character coordinate carried as the bias $\boldsymbol{b}$. By Parseval, each function $\chi_k \circ f$ has unit Fourier energy, $|b_k|^2 + \sum_{j=1}^{t-1} |A_f^{\mathrm{red}}[k, j]|^2 = 1$, and summing over $k = 1, \ldots, t-1$ gives the energy identity

$$\|A_f^{\mathrm{red}}\|_F^2 + \|\boldsymbol{b}\|^2 = t - 1.$$

The per-slot RMS amplification under isotropic noise is therefore

$$\sqrt{\|A_f^{\mathrm{red}}\|_F^2/(t-1)} = \sqrt{1 - \|\boldsymbol{b}\|^2/(t-1)} \leq 1.$$

The bias vanishes exactly when f is bijective, in which case A_f^{red} is unitary on $\mathbb{C}^{t-1}$ and the per-slot RMS amplification is 1. For non-bijective f the bias absorbs some of the energy, so the average per-slot amplification drops strictly below 1. The operator norm $\|A_f^{\mathrm{red}}\|_2$ may still exceed 1, so individual input directions can be amplified more than the average. Bijective LUTs are therefore the worst case for the per-slot RMS amplification but the most regular across directions, which matches the experimental observation that bijective chains show slightly worse but more regular noise growth than non-bijective ones.

The L-BRU encoding has the same Fourier behavior on $\mathbb{Z}_t^*$, together with an additional nonzero indicator coordinate. IDCT encoding, by contrast, has very simple coefficients for indicator-to-indicator LUTs, but switches from or to other bases may have row norms depending on the omitted value $a_\star$. The choice of encoding therefore affects precision, and the omitted coordinate in reduced encodings can be selected to minimize the relevant BLT row norms.

3.6 Noise growth under chained multiplications

Character block encodings and standard discrete CKKS encodings differ in how noise grows along a chain of operations. The difference is structural and visible in

the per-multiplication noise bound. Write a plaintext slot value as $m = m^* + \varepsilon$, where m^* is the noise-free slot value and ε is the slot error. In discrete CKKS the slot value is an integer in $\{0, \ldots, t-1\}$. In a block encoding it is a root of unity for BRU and WH, lies in $\{0\} \cup \{\zeta_{t-1}^k\}$ for L-BRU, and lies in $\{0, 1\}$ for IDCT and TH. For two slot values m_1, m_2 the homomorphic product before rescale satisfies

$$m_1 m_2 = m_1^* m_2^* + m_1^* \varepsilon_2 + m_2^* \varepsilon_1 + \varepsilon_1 \varepsilon_2$$

so the slot-wise infinity norm of the new error is bounded by

$$\|\varepsilon_{\text{new}}\|_\infty \leq \|m_2^*\|_\infty \|\varepsilon_1\|_\infty + \|m_1^*\|_\infty \|\varepsilon_2\|_\infty + \|\varepsilon_1\|_\infty \|\varepsilon_2\|_\infty. \qquad (*)$$

The cross-term coefficient is the operand magnitude $\|m^*\|_\infty$, which is what differs between the two encoding families.

We compare the two families on the chain operative for our framework, namely k consecutive multiplications of an accumulator by an operand of fresh-noise budget ε_0. This covers BLT and LUT applications, encoding switches, and chains of group-law operations against fresh ciphertexts.

Discrete CKKS encoding. Plaintexts are encoded integers $m^* \in \{0, \ldots, t-1\}$, so $\|m^*\|_\infty \leq t$. The per-step recurrence is $\varepsilon_{n+1} \leq t\varepsilon_n + t\varepsilon_0 + \varepsilon_n \varepsilon_0$. Dropping the quadratic term and iterating from ε_0 gives $\varepsilon_k \approx t^k \varepsilon_0$. Correct decoding requires $\|\varepsilon_k\|_\infty < \frac{1}{2}$, hence $k \leq \log_t(1/2\varepsilon_0)$. For $\varepsilon_0 = 2^{-\beta}$ with $\beta \approx 13$ and $t = 256$, only $\lfloor 12/8 \rfloor = 1$ multiplication is permitted before cleaning becomes necessary.

Character block encoding. Plaintexts are roots of unity, $m^* \in \{\zeta_t^k : k \in \mathbb{Z}_t\}$ for BRU, and analogously for L-BRU, WH and IDCT. The defining magnitude bound is $\|m^*\|_\infty \leq 1$, so $(*)$ reduces to

$$\|\varepsilon_{\text{new}}\|_\infty \leq \|\varepsilon_1\|_\infty + \|\varepsilon_2\|_\infty + \|\varepsilon_1\|_\infty \|\varepsilon_2\|_\infty.$$

The recurrence is $\varepsilon_{n+1} \leq \varepsilon_n + \varepsilon_0 + \varepsilon_n \varepsilon_0$. Dropping the quadratic term gives $\varepsilon_k \leq k\varepsilon_0(1 + O(\varepsilon_0))$. Correct decoding requires $\|\varepsilon_k\|_\infty < \sin(\pi/t) \approx \pi/t$, hence $k \leq \pi/(t\varepsilon_0)$. For $\varepsilon_0 = 2^{-\beta}$ and $t = 256$ this gives $k \leq \pi \cdot 2^{\beta-8}$, exponentially better in β than the discrete CKKS bound. At $\beta \approx 13$ the bound is roughly $k \approx 100$.

Self-multiplications. For the special case of self-multiplication $m \mapsto m^2$, inequality $(*)$ specializes to $\varepsilon_{n+1} \leq 2\|m^*\|_\infty \varepsilon_n + \varepsilon_n^2$, so the chain bound is $\varepsilon_k \leq (2\|m^*\|_\infty)^k \varepsilon_0$. This is $2^k \varepsilon_0$ in a block encoding and $(2t)^k \varepsilon_0$ in discrete CKKS. Both are exponential in k, but block encoding's base is 2 rather than $2t$, an exponential improvement that pushes practical squaring chains into the tens rather than one or two. The 2^k self-multiplication bound is the worst case for any block encoding, matching the noise growth of discrete CKKS on $\mathbb{Z}_2$, which is its smallest alphabet and best case.

Cross multiplications. Inequality $(*)$ holds for any pair of ciphertext operands, so mixed chains obey per-step bounds of the same shape, with the operand magnitude $\|m^*\|_\infty$ entering as the structural difference between the two settings. In a block encoding every slot has magnitude at most 1 across BRU, L-BRU, WH, IDCT and TH, so $\|m^*\|_\infty \le 1$ is preserved exactly under arbitrary products and the linear chain bound is unconditional. In discrete CKKS the integer slot grows as $m_1^* m_2^* = d_1 d_2$, up to $(t-1)^2$, so $\|m^*\|_\infty$ must be brought back below t by an explicit modular reduction between multiplications. The t^k chain bound therefore assumes such a reduction at every step, even though each reduction is itself a noisy and depth-consuming operation.

This stability also extends to the shape of the noise. In the Fourier-type encodings BRU, L-BRU, and WH every slot of a nonzero operand carries a unit-modulus value, so multiplication rotates each slot's noise rather than rescaling it and the error covariance remains a scalar multiple of the identity. The isotropy of Subsection 3.5 therefore carries through the chain, and the average per-slot amplification stays close to 1 at every step. In discrete CKKS slots are scaled by different integer plaintexts at each multiplication, so per-slot variances diverge across the block and the error becomes anisotropic after a few multiplications, with the most amplified slots dominated by the operator-norm bound rather than the Frobenius average.

Table 3. Multiplicative noise growth for a chain of k multiplications under the two encoding families, with starting error ε_0 and alphabet size t. Each step multiplies the accumulator by an operand of fresh-noise budget ε_0.

Encoding	$\|\varepsilon_k\|_\infty$ bound	Max safe chain length
Discrete CKKS	$\varepsilon_0 \cdot t^k$	$O(\log(1/\varepsilon_0)/\log t)$
Block encoding	$k\varepsilon_0(1+O(\varepsilon_0))$	$O(1/(t\varepsilon_0))$

Summary. The discrete CKKS bound is exponential in k and logarithmic in $1/\varepsilon_0$, while the block encoding bound is linear in k and linear in $1/\varepsilon_0$. Under random encryption noise, the empirical prefactor is closer to $O(\sqrt{k} \cdot \varepsilon_0)$, but the worst case remains linear in k. CKKS ciphertext noise from encryption, relinearization, rescaling, and key-switching grows as in standard CKKS. What changes is that the operand-magnitude amplification in multiplication disappears. This behavior is observed experimentally in Subsection 7.4.

3.7 Cleaning

Cleaning is more naturally expressed in the IDCT (or TH) encoding, by evaluating the cleaning map $H : x \mapsto 3x^2 - 2x^3$ [DMPS24], which consumes 2 levels and provides a quadratic reduction in noise. As encoding switches are BLT and consume one level, any encoding E can be cleaned in at most 4 levels

$$\mathsf{E} \xrightarrow{\mathsf{BLT}} \mathsf{IDCT} \xrightarrow{H} \mathsf{IDCT} \xrightarrow{\mathsf{BLT}} \mathsf{E}',$$

where E' can be either the original or a different encoding.

This gives a constant-depth cleaning procedure whose depth is independent of the alphabet size t. The key point is that the potentially t-dependent operation is isolated in the two encoding switches, both of which are BLTs, while the nonlinear cleaning map itself is the fixed polynomial H. This contrasts with previous approaches [CKKL24,BKSS24,AKP25], where the cleaning depth grows logarithmically with t. For WH encoding, the slots are signs, so one can directly clean using the map $H_{\pm} : x \mapsto 1/2(3x - x^3)$ which fixes ± 1 and has zero derivative at both fixed points, saving encoding switches.

3.8 Switching back and to character block encoding

The encoding switches inside the framework are character-to-character switches, realized as BLTs, and do not require leaving the character representation. Should it be necessary to interface with standard discrete CKKS, both directions of switch are available, at very different costs.

From a character block to standard discrete CKKS. For BRU, the map $\mathsf{BRU}_t(m) \mapsto m$ is the Fourier inverse

$$m = \frac{t-1}{2} + \sum_{k=1}^{t-1} \widehat{\mathsf{id}}(k)\zeta_t^{km}, \qquad \widehat{\mathsf{id}}(k) = \frac{1}{t}\sum_{a=0}^{t-1} a\zeta_t^{-ka},$$

which is one BLT. The same applies to L-BRU, WH, and IDCT: each is reduced to standard discrete CKKS by a single BLT, hence one multiplicative level, with depth independent of t and low algorithmic complexity.

From standard discrete CKKS to a character block. The reverse direction is not linear in m. Encoding a scalar $m \in \mathbb{Z}_t$ as $\mathsf{BRU}_t(m) = (\zeta_t^m, \zeta_t^{2m}, \ldots, \zeta_t^{(t-1)m})$ requires computing ζ_t^m as a function of the slot value, which no CKKS BLT can produce on its own. The only homomorphic way to compute ζ_t^m from m is to evaluate the discrete exponential approximately, exactly as the EvalExp step of a CKKS functional bootstrapping [BKSS24, AKP25]. A naive realization is a balanced power-basis tree of depth $\lceil \log_2(t-1) \rceil$, but the structure of EvalExp admits a depth-saving variant detailed in Appendix B. Any other block encoding is reached from BRU by one BLT.

We use the split layout for the switched ciphertext, with each character coordinate held in its own ciphertext. This matches the discrete-CKKS setting where the $N/2$ complex CKKS slots are read as N real-valued slots, with the real and imaginary parts treated separately, so that the bootstrapping output fits the split coordinates directly without further repacking. Neither switch is needed in the rest of the paper, where all constructions remain within the block-encoding framework.

4 CRT Arithmetic

A natural application of character encodings, and in particular BRU and L-BRU encodings, is CRT arithmetic. Let $T = t_1 t_2 \cdots t_n$ with pairwise coprime integers $t_k \geq 2$. By the CRT, there is a ring isomorphism

$$\mathbb{Z}_T \simeq \mathbb{Z}_{t_1} \times \cdots \times \mathbb{Z}_{t_n}$$

and every $m \in \mathbb{Z}_T$ decomposes as

$$m \longleftrightarrow (m_1, \ldots, m_n), \qquad m_k = m \bmod t_k.$$

A message $m \in \mathbb{Z}_T$ can therefore be represented by n BRU blocks. When the moduli t_k are prime, one may instead use n L-BRU blocks. The blocks may be packed contiguously within a single CKKS ciphertext or distributed across multiple ciphertexts. In total, this representation uses $\sum_{k=1}^{n}(t_k - 1)$ slots. For example, one can use the first 44 primes to perform operations on $\mathbb{Z}_T$ with $\log_2 T \approx 257$ using only 3787 slots, with 193 the largest prime.

In the BRU encoding, component-wise CKKS multiplication of the blocks adds residues modulo each t_k, and therefore adds in $\mathbb{Z}_T$ under the CRT isomorphism. In the L-BRU encoding, component-wise CKKS multiplication of the blocks multiplies residues modulo each prime t_k, and therefore multiplies in $\mathbb{Z}_T$.

The same CRT decomposition applies to fixed polynomial maps. Let $P \in \mathbb{Z}_T[X]$ and let $P_k \in \mathbb{Z}_{t_k}[X]$ be obtained by reducing the coefficients of P modulo t_k. For $m \in \mathbb{Z}_T$ with CRT components $(m_1, \ldots, m_n)$, the CRT isomorphism gives

$$\forall k \in [1, n],\ P(m) \bmod t_k = P_k(m_k).$$

Hence, evaluating P over $\mathbb{Z}_T$ reduces to evaluating, independently on each CRT component, the unary map $m_k \mapsto P_k(m_k)$. In either the BRU or L-BRU representation, each such component map is a unary LUT and therefore costs one BLT level, that is, evaluating P costs exactly one level in both encodings.

5 Finite-State Prefix Scans

Several operations on large words with radix digits share the same shape: at each position a small piece of state is updated locally, and that state is propagated through the word. For addition the state is the carry; for subtraction, the borrow; for comparison, whether the more-significant prefix has already decided the outcome. We give one generic construction that turns any such operation into a parallel-prefix scan whose depth is independent of the size of the digit alphabet, and only depends on the number of digits d, generalizing Cha et al. [CPL26].

Let $\mathcal{S}$ be a finite state space, and at each position k let

$$F_k : \mathcal{S} \to \mathcal{S}$$

be the local transition on this state. Given an initial $c_{-1} \in \mathcal{S}$, the state entering position k is

$$c_{k-1} = (F_{k-1} \circ \cdots \circ F_0)(c_{-1}).$$

A parallel-prefix network such as Kogge–Stone computes all of $c_0, \ldots, c_{d-1}$ in $\lceil \log_2 d \rceil$ rounds, provided two adjacent transitions can be composed in low depth.

Composition via indicator tables. We represent a transition $F : \mathcal{S} \to \mathcal{S}$ by its non-reduced IDCT table

$$T_F(c, r) = \delta_{F(c),r}, \qquad c, r \in \mathcal{S},$$

so each row $T_F(c, \cdot)$ is the one-hot encoding of $F(c)$. The non-reduced form is convenient: $|\mathcal{S}|$ does not depend on the word length, and the composition formula becomes a one-hot selection. For two transitions from low bits F_l to high bits F_h, the composed transition table is

$$T_{F_h \circ F_l}(c, r) = \sum_{s \in \mathcal{S}} T_{F_l}(c, s)\, T_{F_h}(s, r).$$

All slot-wise products run in parallel and the sum has plaintext coefficients, so composition costs one multiplicative level. We compose transitions in the order they are encountered by the scan, lower-significance first, so that $F_k \circ \cdots \circ F_0$ matches the digit order.

Bundled summaries. After the scan, each position must emit an output digit or predicate. To avoid a separate state-consumption LUT after the scan, we bundle the local output map $\Phi_k : \mathcal{S} \to \mathcal{Y}$ with the transition F_k into the summary

$$B_k = (F_k, \Phi_k),$$

where $\Phi_k(c)$ is the output at position k when the incoming state is c, stored in any chosen output encoding. Two adjacent summaries combine by

$$B_l \diamond B_h = (F_h \circ F_l,\ \Phi_h \circ F_l).$$

The transition component propagates state through both intervals. The output component stores the right-endpoint output as a function of the state entering the lower interval. Both components are computed by the same one-hot selector $T_{F_l}(c, \cdot)$, so a bundled composition costs the same single multiplicative level as a transition composition alone.

Proposition 2 (Bundled prefix step). *The operation $\diamond$ is associative, and one application of $\diamond$ costs one multiplicative level when transitions are stored as non-reduced IDCT tables and outputs state by state in the output encoding. The proof is given in Appendix C.*

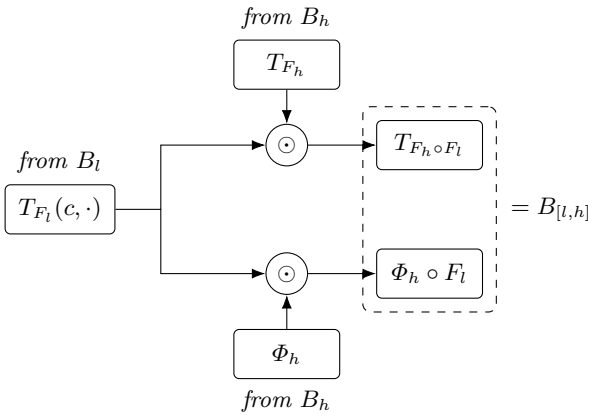

Fig. 2. One $\diamond$ node, expanded. The lower interval contributes its transition table $T_{F_l}(c,\cdot)$, which functions as a one-hot row selector. A single layer of CKKS slot-wise multiplications applies this selector in parallel to T_{F_h} (producing the composed transition $T_{F_h \circ F_l}$) and to Φ_h (producing the right-endpoint output map $\Phi_h \circ F_l$). Both products complete within one multiplicative level. The output map Φ_l is not consumed here: it only participates in bundles whose right endpoint is at position l.

By associativity, every parallel-prefix parenthesization yields the same prefix summary; in particular, $B_0 \diamond \cdots \diamond B_k$ stores

$$\Phi_k \circ F_{k-1} \circ \cdots \circ F_0,$$

and the output at position k is obtained by evaluating this stored table at c_{-1}. No further LUT is needed after the scan.

Theorem 1 (Generic prefix-scan depth). *Let $\mathcal{S}$ be a finite state space whose size is independent of the word length d. Suppose a word operation decomposes as follows.*

- *At each position k, a local extraction of depth D_{set} emits a bundled summary $B_k = (F_k, \Phi_k)$ with $F_k : \mathcal{S} \to \mathcal{S}$ and $\Phi_k : \mathcal{S} \to \mathcal{Y}$.*
- *The state propagates through the word by parallel-prefix composition.*
- *The desired output at position k is $\Phi_k(c_{k-1})$, where c_{k-1} is the state entering at position k.*

Then all output positions are computed in depth

$$D_{\mathrm{set}} + \lceil \log_2 d \rceil,$$

using lookup tables whose sizes depend only on $|\mathcal{S}|$ and on the local output alphabet, not on d.

Proof. The local extraction yields $B_0, \ldots, B_{d-1}$ in depth D_{set}. By Proposition 2, a parallel-prefix network computes all bundled prefixes $B_{[0,k]} = B_0 \diamond \cdots \diamond B_k$ in

$\lceil \log_2 d \rceil$ rounds, each round costing one level. The output at position k is the entry $\Phi_{[0,k]}(c_{-1})$ already stored in $B_{[0,k]}$, so no further lookup is needed. The table sizes are constant in d because $|\mathcal{S}|$ is. □

In the next section, the local summaries B_k are emitted directly by digit-local LUTs, whose outputs are the transition table and state-indexed output values, thereby fusing output preparation into the local extraction stage.

6 Radix Arithmetic

We now apply the prefix scan theory to radix operations. Fix a radix $t \geq 2$ and a word length d. A word modulo $w = t^d$ is represented by d BRU blocks

$$x = \sum_{k=0}^{d-1} x_k t^k \text{ with } x_k \in \mathbb{Z}_t \qquad \big(\mathsf{BRU}_t(x_0), \ldots, \mathsf{BRU}_t(x_{d-1})\big).$$

This is a positional representation, not a CRT decomposition. The radix is therefore not constrained by coprimality conditions. Throughout this section, when a digit is used in a carry, borrow, or order test, it is identified with its canonical representative in $\{0, \ldots, t-1\}$. Word operations are computed modulo w and the output is again given as d BRU blocks.

The main difference with CRT arithmetic is the presence of carries. Addition, subtraction, and comparison have small-state local updates and instantiate the prefix-scan framework of Section 5 with $|\mathcal{S}| \leq 3$.

Specialized packed layout. The packed layout used in this section is wider than the packed layout of Section 3. Each digit occupies $S = t^2$ slots, with the $t-1$ BRU coordinates in the first $t-1$ slots and zero padding in the remaining t^2-t+1 slots. The padding is consumed by the spread and basis-product intermediates of the bivariate LUT during the prefix scan, which use up to t^2 slots per digit, and by the Kogge–Stone rotation headroom, which improves algorithmic efficiency.

A word is then $d_S = d + s_{\max}$ consecutive digit blocks, where $s_{\max} = 2^{\lceil \log_2 d \rceil - 1}$ is the largest Kogge–Stone shift, and the trailing $s_{\max}$ blocks hold $\mathsf{BRU}_t(0)$ so that cyclic shifts within a word do not bring data from a neighboring word into the scan. A single ciphertext at ring dimension N packs $\lfloor N/(2d_S S) \rfloor$ such words, each operated on in parallel by the constructions below.

6.1 Addition

Let $z = x + y \bmod t^d$. The natural application of Theorem 1 takes the carry as state, $\mathcal{S}_{\text{add}} = \{0, 1\}$, and bundles a local summary (F_k, Φ_k) obtained from a bivariate LUT on (x_k, y_k), with $F_k(c) = \lfloor (u_k + c)/t \rfloor$ and $\Phi_k(c) = (u_k + c) \bmod t$, where $u_k = x_k + y_k$. This direct instantiation has depth $3 + \lceil \log_2 d \rceil$. Our implementation uses instead a Brent–Kung G/P adder variant that trades one multiplicative level for substantially smaller plaintext tables, with total depth $4 + \lceil \log_2 d \rceil$ on the packed layout used in the benchmarks of Section 7.3.

The variant separates the per-digit work into two parallel branches. The first branch computes the carry-free sum

$$\mathsf{BRU}_t(u_k \bmod t) = \mathsf{BRU}_t(x_k) \odot \mathsf{BRU}_t(y_k)$$

in a single ciphertext multiplication. The second branch produces two scalar per-digit signals from one bivariate LUT on (x_k, y_k). The *generate* signal G_k is the BRU delta $\mathsf{BRU}_t(1) - \mathsf{BRU}_t(0)$ when $u_k \geq t$ and zero otherwise. The *propagate* signal P_k is the all-ones character mask when $u_k = t-1$ and zero otherwise. Both signals live in the same character block as the digit, so one ciphertext carries the per-digit pair (G_k, P_k) across the word.

A Brent–Kung scan over $\lceil \log_2 d \rceil$ rounds composes the per-digit pairs into a carry prefix. The composition rule is

$$(G, P) \diamond_{\mathrm{BK}} (G', P') = (G + P \cdot G',\ P \cdot P'),$$

and each round costs one ciphertext multiplication, with a saved P-update on the final round. After the scan, the BRU value at digit k encodes the carry entering digit k from the digits below.

A final correction combines the two branches. The carry-free sum is multiplied by a block-zero plaintext mask that zeros out the lowest digit of every batched word, so that no spurious carry enters digit 0, and the result is then multiplied by the carry prefix and added back to the carry-free sum. The complete procedure is Algorithm 2, which uses the CKKS operations of Appendix A.

Lemma 1 (Correctness of PackedAdd). *For inputs $x, y \in \mathbb{Z}_{t^d}$ encoded as d BRU_t digit blocks per word in the layout of Section 6.1, Algorithm 2 returns a ciphertext whose per-digit decoding equals the digits of $(x + y) \bmod w$, $w = t^d$. The proof is given in Appendix D.*

The level count is 3 for the bivariate LUT, $\lceil \log_2 d \rceil$ for the scan, and 1 for the correction multiplication, for a total of $4 + \lceil \log_2 d \rceil$. The final carry $G^{[d-1]}$ is available for non-modular addition as an extra most-significant digit. A worked example for $t = 10$ and $d = 3$ is given in Appendix E.

6.2 Subtraction

Let $z = x - y \bmod t^d$. The state is the borrow, $\mathcal{S}_{\mathrm{sub}} = \{0, 1\}$. Subtraction follows the same Brent–Kung structure as Algorithm 2. The carry-free branch computes the per-digit raw difference

$$\mathsf{BRU}_t(x_k - y_k \bmod t) = \mathsf{BRU}_t(x_k) \odot \overline{\mathsf{BRU}_t(y_k)},$$

using that $\overline{\mathsf{BRU}_t(v)} = \mathsf{BRU}_t(-v \bmod t)$. The generate signal is the BRU delta when $x_k < y_k$ and zero otherwise, and the propagate signal is the all-ones mask when $x_k = y_k$ and zero otherwise. The total depth is $4 + \lceil \log_2 d \rceil$, and the final borrow $G^{[d-1]}$ indicates whether the unsigned subtraction underflowed.

Algorithm 2 Packed radix addition by Brent–Kung G/P prefix scan

procedure $\mathsf{PackedAdd}(\mathsf{ct}_x, \mathsf{ct}_y \in R^2_{Q_\ell})$, where ct_x and ct_y encode each word into d BRU-t digit blocks at slot stride $S = t^\delta$, padded to $d_S = d + s_{\max}$ blocks per word with $s_{\max} = 2^{\lceil \log_2 d \rceil - 1}$ as defined above

$\mathsf{ct}_{\mathrm{rs}} \leftarrow \mathsf{Mul}(\mathsf{ct}_x, \mathsf{ct}_y)$
$\mathsf{ct}_{x'} \leftarrow \mathsf{Rotate}(\mathsf{ct}_x, -S)$, $\mathsf{ct}_{y'} \leftarrow \mathsf{Rotate}(\mathsf{ct}_y, -S)$
$(\mathsf{ct}_G, \mathsf{ct}_P) \leftarrow \mathsf{BivariateLUT}(\mathsf{ct}_{x'}, \mathsf{ct}_{y'};\ f_G, f_P)$
for $r \in [\![0, \lceil \log_2 d \rceil - 1]\!]$ **do**
    $\mathsf{ct}_G^{(s)} \leftarrow \mathsf{Rotate}(\mathsf{ct}_G, -2^r S)$, $\mathsf{ct}_P^{(s)} \leftarrow \mathsf{Rotate}(\mathsf{ct}_P, -2^r S)$
    $\mathsf{ct}_G \leftarrow \mathsf{Add}(\mathsf{ct}_G, \mathsf{Mul}(\mathsf{ct}_P, \mathsf{ct}_G^{(s)}))$
    if $r < \lceil \log_2 d \rceil - 1$ **then**
        $\mathsf{ct}_P \leftarrow \mathsf{Mul}(\mathsf{ct}_P, \mathsf{ct}_P^{(s)})$
$\mathsf{ct}_\delta \leftarrow \mathsf{MulPlain}(\mathsf{ct}_{\mathrm{rs}}, \mathsf{pt}_{\mathrm{mask}})$
$\mathsf{ct}_{\mathrm{corr}} \leftarrow \mathsf{Mul}(\mathsf{ct}_G, \mathsf{ct}_\delta)$
return $\mathsf{Add}(\mathsf{ct}_{\mathrm{rs}}, \mathsf{ct}_{\mathrm{corr}}) \in R^2_{Q_{\ell-4-\lceil \log_2 d \rceil}}$

6.3 Comparison and Equality

For unsigned comparison, the state $\mathcal{S}_{\mathrm{cmp}} = \{\mathsf{eq}, \mathsf{lt}, \mathsf{gt}\}$ records the comparison result of the already-processed more-significant prefix. Once a more-significant unequal digit is found, the state is absorbing:

$$F_k(\mathsf{lt}) = \mathsf{lt}, \quad F_k(\mathsf{gt}) = \mathsf{gt}, \quad F_k(\mathsf{eq}) = \begin{cases} \mathsf{lt} & x_k < y_k, \\ \mathsf{gt} & x_k > y_k, \\ \mathsf{eq} & x_k = y_k. \end{cases}$$

The output map can be $\Phi_k(c) = F_k(c)$ encoded as a non-reduced indicator over $\mathcal{S}_{\mathrm{cmp}}$, or any chosen subset of the predicates $\{x < y, x = y, x > y\}$. A most-significant-first scan with initial state eq yields the comparison in depth $3 + \lceil \log_2 d \rceil$. If only equality is needed, the smaller state $\mathcal{S}_{\mathrm{eq}} = \{0, 1\}$ suffices, with $F_k(e) = e \cdot \delta_{x_k, y_k}$ and 0 absorbing. The scan may proceed in either digit order, and the depth is again $3 + \lceil \log_2 d \rceil$.

7 Experimental Results

The experiments were run on a desktop equipped with an AMD Ryzen 9 9950X and 64GB of RAM, using a single thread. We used Pro7ech's Lattigo [lat24] FHE library for all computation. [4] We use either $\log_2 N = 15$ or $\log_2 N = 16$, depending on the computation depth. The hamming weight of the secret key is $h = 256$, which gives a threshold for the modulus QP of 774 bits for $\log_2 N = 15$ and 1553 bits for $\log_2 N = 16$, according to the most conservative estimations on sparse key security from [Ogi26] for 128 bits of security, using the tables

[4] Our implementation will be made publicly available.

of [Dum26]. CKKS bootstrapping is not needed for any computation presented below, but we systematically choose parameters that leave enough modulus to perform bootstrapping and then re-run the same computation, so that they remain composable. Therefore, $\log_2 QP$ as reported in the tables below is the modulus chain used by each listed operation in isolation, not the parameter ceiling: the underlying CKKS instance is set up to the secure threshold above, so a standard CKKS bootstrapping can raise the chain back to that ceiling and restore the short chain between operations when needed. Functional bootstrapping examples were run with OpenFHE v1.5.1 [BAB+22] on the same hardware. All reported timings are averaged on 10 iterations.

7.1 General Performances

Table 4 reports the cost of the five primitive operations of the framework in the BRU encoding and the packed layout, across alphabet sizes $t = 2^{\log_2 t}$ focused on $\log_2 t \in \{2, 4, 8\}$. The group-law operation is one CKKS multiplication and has constant cost in t. LUT, Bivariate LUT, and 4-variate LUT scale with the alphabet size through the cost of the spread and basis-product steps of Algorithm 1. Full scans are in Appendix F, and the precision behavior across chained operations is discussed in Section 7.4.

Table 4. Primitive operations in the BRU encoding, packed layout, single thread. The group-law operation is one CKKS slot multiplication, which in BRU evaluates a modular addition. Amortized time is the latency divided by the number of packed values, which we maximize for a set t at $\lfloor N/2(t-1) \rfloor$, except for bivariate and 4-variate LUTs at $\lfloor N/2t^2 \rfloor$ and $\lfloor N/2t^4 \rfloor$ respectively to minimize latency.

Operation	$\log_2 t$	$\log_2 N$	$\log_2 QP$	$\log_2 \Delta$	Values	Latency	Amtz. time
Group Law	4	15	155	40	1092	4.9 ms	4 μs
LUT	2	15	155	40	5461	8.7 ms	2 μs
LUT	4	15	155	40	1092	24.2 ms	22 μs
LUT	8	15	155	40	64	122.5 ms	1.9 ms
Bivariate LUT	2	15	235	40	1024	102.1 ms	100 μs
Bivariate LUT	4	15	235	40	64	324.2 ms	5.1 ms
4-variate LUT	2	15	275	40	64	943.5 ms	14.7 ms
Cleaning	2	15	275	40	5461	56.4 ms	10 μs
Cleaning	4	15	275	40	1092	123.6 ms	113 μs
Cleaning	8	15	275	40	64	507.9 ms	7.9 ms

Table 5 fixes $\log_2 t = 4$ and reports LUT and Cleaning across the four block encodings. LUTs have very similar latency across encodings. Cleaning is qualitatively different: BRU and L-BRU require an encoding switch into IDCT to reach the degree-three cleaning polynomial, so they consume four levels at $\log_2 QP = 275$, whereas WH is cleaned directly through the fixed-point polynomial $H_{\pm}$, with IDCT at two levels and $\log_2 QP = 195$.

Table 5. LUT and Cleaning across encodings at $\log_2 t = 4$, packed layout, $\log_2 N = 15$, $\log_2 \Delta = 40$, single thread. The Group Law and multivariate LUTs behave like the BRU column of Table 4 for every encoding and are not repeated. L-BRU requires a prime modulus and falls back to the alphabet of size 13 at the nominal $\log_2 t = 4$, which uses 12 slots per block instead of 15 which increases the number of values treated at once. The comparison is therefore not strictly like-for-like.

Operation	Encoding	$\log_2 t$	$\log_2 QP$	Values	Latency	Amtz. time
LUT	BRU	4	155	1092	24.2 ms	22 μs
LUT	L-BRU	≈ 4	155	1365	20.5 ms	15 μs
LUT	WH	4	155	1092	24.2 ms	22 μs
LUT	IDCT	4	155	1092	19.8 ms	18 μs
Cleaning	BRU	4	275	1092	123.6 ms	113 μs
Cleaning	L-BRU	≈ 4	275	1365	109.0 ms	80 μs
Cleaning	WH	4	195	1092	12.5 ms	11 μs
Cleaning	IDCT	4	195	1092	12.6 ms	12 μs

Table 6 reports the LUT evaluated through the AKP functional bootstrapping [AKP25] in OpenFHE on the same machine, for comparison at the same $\log_2 t \in \{2, 4, 8\}$. The full $\log_2 t$ scan is in Appendix F.

Table 6. LUT through the AKP functional bootstrapping [AKP25] in OpenFHE v1.5.1, single thread. The number of values is the full ring dimension N, as the real and imaginary part of the ciphertext are treated separently.

$\log_2 t$	$\log_2 N$	$\log_2 QP$	$\log_2 \Delta$	Values	Latency	Amtz. time
2	16	1036	35	65536	6.70 s	102 μs
4	16	1234	38	65536	8.51 s	130 μs
8	16	1656	47	65536	17.31 s	264 μs

The two approaches sit on opposite sides of the same tradeoff. Our leveled LUT is two to three orders of magnitude faster per evaluation, since it consumes one multiplicative level rather than a full bootstrapping. The functional bootstrapping has more throughput, beating LUT evaluation on larger alphabets like $\log_2 t = 8$ an order of magnitude. However, in a computation that chains LUTs, our framework avoids paying a bootstrapping at every lookup, at the cost of a smaller SIMD width for improved latency.

7.2 CRT Performances

Table 7 reports CRT arithmetic on $\log_2 T$-bit moduli built from the first primes such that the product exceeds $2^{\log_2 T}$. Addition and subtraction are native in BRU, multiplication is native in L-BRU, and the encoding switch is one BLT per component. Cleaning is the constant-depth procedure of Section 3.7. All operations are exact modulo T.

Table 7. CRT arithmetic, packed layout, single thread. The number of packed values is $\lfloor (N/2)/\sum_i (p_i - 1) \rfloor$ over the component primes, which at $\log_2 T = 64$ are the primes up to 53 ($\sum_i (p_i - 1) = 365$) and at $\log_2 T = 256$ the primes up to 193 ($\sum_i (p_i - 1) = 3787$).

Operation	$\log_2 T$	$\log_2 N$	$\log_2 QP$	$\log_2 \Delta$	Values	Latency	Amtz. time
Add	64	15	155	40	44	4.7 ms	106 μs
Sub	64	15	155	40	44	8.4 ms	190 μs
Mul (L-BRU)	64	15	155	40	44	4.7 ms	106 μs
BRU $\rightarrow$ L-BRU	64	15	155	40	44	48.1 ms	1.1 ms
Cleaning	64	15	275	40	44	224.9 ms	5.1 ms
Add	256	15	155	40	4	4.7 ms	1.2 ms
Sub	256	15	155	40	4	8.4 ms	2.1 ms
Mul (L-BRU)	256	15	155	40	4	4.7 ms	1.2 ms
BRU $\rightarrow$ L-BRU	256	15	155	40	4	103.9 ms	26.0 ms
Cleaning	256	15	275	40	4	449.1 ms	112.3 ms

The closest shared metric with concurrent work is the cost of a 256-bit modular multiplication amortized over packed slots. Reported figures for amortized time are 4.69 s [BK25] under approximate reduction, 62.3 ms [Kim25b] under lazy reduction, 76 ms [GZ26] in the arithmetic mode of the triangle encoding, and 292 ms [CPL26], all using bootstrapping. Our 1.2 ms is depth-one and exact, but it is not on exactly 256 bits and does not include the cost of refreshing the modulus budget or cleaning the value. However, as discussed in Subsection 7.4, cleaning is not needed for very long chains of operations, even 256 bits CRT multiplications. Also the main advantage remains latency, which is *three to four* orders of magnitude lower than previous works [BK25, Kim25b, CPL26, GZ26].

7.3 Radix Performances

Table 8 reports radix arithmetic on $\log_2 w$-bit words, with radix $t = 4$ and word length $d = \log_2 w/2$. Addition and subtraction use the Brent–Kung G/P prefix scan of Algorithm 2. Equality, less-than, and compare instantiate Theorem 1 directly. Cleaning is the constant-depth procedure of Section 3.7.

7.4 Precision under chaining

Figure 3 shows the precision of the encrypted output after k consecutive operations, with a CKKS bootstrapping inserted whenever the modulus chain runs out. The three primitives reported are the group law (one CKKS multiplication), a unary LUT (BLT), and a 256-bit CRT modular multiplication. Each curve averages 3 random seeds at $\log_2 N = 16$ and $\log_2 QP = 1513$. Each iteration applies a fresh random LUT, restricted to bijective f (a uniformly random permutation). A chain of random non-bijective f collapses, within a few iterations, into the short fixed-point cycles of its functional graph and stops being a meaningful test of accumulated noise on diverse plaintexts. For bijective f, the LUT remains non-degenerate at every iteration and the corresponding A_f

Table 8. Radix 4 arithmetic, using the operations of Section 6. The depth scales as $4 + \lceil \log_2 d \rceil$ for addition and subtraction, $3 + \lceil \log_2 d \rceil$ for equality (Eq), less-than (Lt), and compare (Cmp), and 4 for cleaning. The number of packed words is $\lfloor (N/2)/(t^2 \cdot (d + 2^{\lceil \log_2 d \rceil - 1})) \rfloor$, where $t^2 = 16$ slots per digit and the additive headroom isolates words from the cyclic rotations of the scan.

Operation	$\log_2 w$	$\log_2 N$	$\log_2 QP$	$\log_2 \Delta$	Words	Latency	Amtz. time
Add	64	16	480	40	42	2.11 s	50.2 ms
Sub	64	16	480	40	42	2.21 s	52.7 ms
Eq	64	16	440	40	42	1.02 s	24.4 ms
Lt	64	16	440	40	42	1.40 s	33.2 ms
Cmp	64	16	440	40	42	1.78 s	42.5 ms
Clean	64	16	280	40	42	119.9 ms	2.9 ms
Add	256	16	560	40	10	3.16 s	316.1 ms
Sub	256	16	560	40	10	3.29 s	328.5 ms
Eq	256	16	520	40	10	1.57 s	156.5 ms
Lt	256	16	520	40	10	2.21 s	220.6 ms
Cmp	256	16	520	40	10	2.87 s	286.8 ms
Clean	256	16	280	40	10	118.2 ms	11.8 ms

is unitary (Subsection 3.5), so the per-iteration BLT-noise contribution is the same in every direction. Bootstrapping appears as vertical drops in precision, and is executed every 11 operations in every chain. Identical parameters and bootstrapping are used for all configurations.

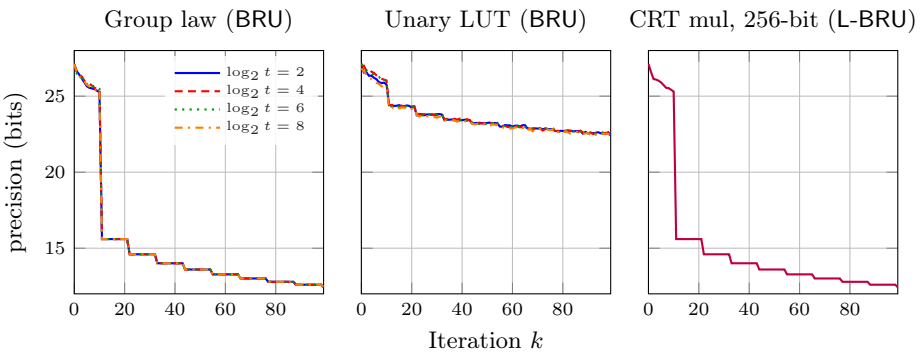

Fig. 3. Precision of the encrypted output after k consecutive operations at $\log_2 N = 16$, averaged over 3 random seeds. A CKKS bootstrapping is performed when the modulus chain is exhausted, every 11 operations in every chain. Bootstrappings appear as the sharp vertical drops. The reported precision is the worst across all slots. The LUT for each iteration is freshly randomly generated.

Two observations follow from the figure. First, the steady-state precision stays well above the correctness threshold for every chain, including after nine bootstrappings and 100 consecutive operations, so the chain length is bounded by the level budget alone rather than by precision. Second, the post-bootstrapping precision is much higher when the chain consists of unary LUTs than when it consists of slot multiplications. For the group law and the CRT multiplication, the precision after the first bootstrapping is approximately 15.6 bits, while for the unary LUT it sits at approximately 24.3 bits across the entire $\log_2 t \in \{2, 4, 6, 8\}$ range, for an identical bootstrapping. After the first bootstrapping, the precision drops sharply, then decays approximately linearly at a slow rate within our 100-iteration window. The linear decay is consistent with the noise-growth analysis of Section 3.6, which predicts a per-multiplication error contribution that adds linearly in the number of operations on unit-circle encodings. After nine bootstrappings, the precision sits at approximately 12.4 bits for the group law and the CRT chain, and at approximately 22.5 bits for the unary LUT chains, still well above the $\log_2(2t)$ correctness threshold for any alphabet up to $t = 256$. On the group-law and CRT chains, whose precision sits at about 12.4 bits after a hundred iterations, one cleaning operation (Subsection 3.7) brings the precision to 15.05 bits. At the empirical degradation rate of 0.040 bits per iteration measured on these chains, this 2.65-bit gain corresponds to roughly 66 iterations of saved degradation, so one cleaning extends the usable chain by approximately that many operations before the precision returns to the same level. The unary LUT chains do not benefit from cleaning, since their plateau is already above the cleaning floor.

Acknowledgments. We thank Aymen Boudguiga, Nathan Fauvelle-Aymar and Renaud Sirdey for reading and providing feedback on an earlier version of this work.

References

AKP25. Andreea Alexandru, Andrey Kim, and Yuriy Polyakov. General functional bootstrapping using CKKS. In Yael Tauman Kalai and Seny F. Kamara, editors, *Advances in Cryptology – CRYPTO 2025, Part III*, volume 16002 of *Lecture Notes in Computer Science*, pages 304–337, Santa Barbara, CA, USA, August 17–21, 2025. Springer, Cham, Switzerland.

BAB+22. Ahmad Al Badawi, Andreea Alexandru, Jack Bates, Flavio Bergamaschi, David Bruce Cousins, Saroja Erabelli, Nicholas Genise, Shai Halevi, Hamish Hunt, Andrey Kim, Yongwoo Lee, Zeyu Liu, Daniele Micciancio, Carlo Pascoe, Yuriy Polyakov, Ian Quah, Saraswathy R.V., Kurt Rohloff, Jonathan Saylor, Dmitriy Suponitsky, Matthew Triplett, Vinod Vaikuntanathan, and Vincent Zucca. OpenFHE: Open-source fully homomorphic encryption library. Cryptology ePrint Archive, Paper 2022/915, 2022. `https://eprint.iacr.org/2022/915`.

BCKS24. Youngjin Bae, Jung Hee Cheon, Jaehyung Kim, and Damien Stehlé. Bootstrapping bits with CKKS. In Marc Joye and Gregor Leander, editors, *Advances in Cryptology – EUROCRYPT 2024, Part II*, volume 14652 of *Lecture*

Notes in Computer Science, pages 94–123, Zurich, Switzerland, May 26–30, 2024. Springer, Cham, Switzerland.

BFG+25. Zvika Brakerski, Offir Friedman, Daniel Golan, Alon Gurny, Dolev Mutzari, and Ohad Sheinfeld. REFHE: Fully homomorphic ALU. Cryptology ePrint Archive, Report 2025/1449, 2025.

BGV12. Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. (Leveled) fully homomorphic encryption without bootstrapping. In Shafi Goldwasser, editor, *ITCS 2012: 3rd Innovations in Theoretical Computer Science*, pages 309–325, Cambridge, MA, USA, January 8–10, 2012. Association for Computing Machinery.

BK25. Dan Boneh and Jaehyung Kim. Homomorphic encryption for large integers from nested residue number systems. In Yael Tauman Kalai and Seny F. Kamara, editors, *Advances in Cryptology – CRYPTO 2025, Part III*, volume 16002 of *Lecture Notes in Computer Science*, pages 338–370, Santa Barbara, CA, USA, August 17–21, 2025. Springer, Cham, Switzerland.

BKSS24. Youngjin Bae, Jaehyung Kim, Damien Stehlé, and Elias Suvanto. Bootstrapping small integers with CKKS. In Kai-Min Chung and Yu Sasaki, editors, *Advances in Cryptology – ASIACRYPT 2024, Part I*, volume 15484 of *Lecture Notes in Computer Science*, pages 330–360, Kolkata, India, December 9–13, 2024. Springer, Singapore, Singapore.

Bra12. Zvika Brakerski. Fully homomorphic encryption without modulus switching from classical GapSVP. In Reihaneh Safavi-Naini and Ran Canetti, editors, *Advances in Cryptology – CRYPTO 2012*, volume 7417 of *Lecture Notes in Computer Science*, pages 868–886, Santa Barbara, CA, USA, August 19–23, 2012. Springer Berlin Heidelberg, Germany.

CGGI16. Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. Faster fully homomorphic encryption: Bootstrapping in less than 0.1 seconds. In Jung Hee Cheon and Tsuyoshi Takagi, editors, *Advances in Cryptology – ASIACRYPT 2016, Part I*, volume 10031 of *Lecture Notes in Computer Science*, pages 3–33, Hanoi, Vietnam, December 4–8, 2016. Springer Berlin Heidelberg, Germany.

CGGI17. Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. Faster packed homomorphic operations and efficient circuit bootstrapping for TFHE. In Tsuyoshi Takagi and Thomas Peyrin, editors, *Advances in Cryptology – ASIACRYPT 2017, Part I*, volume 10624 of *Lecture Notes in Computer Science*, pages 377–408, Hong Kong, China, December 3–7, 2017. Springer, Cham, Switzerland.

CGGI20. Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. TFHE: Fast fully homomorphic encryption over the torus. *Journal of Cryptology*, 33(1):34–91, January 2020.

CHK+19. Jung Hee Cheon, Kyoohyung Han, Andrey Kim, Miran Kim, and Yongsoo Song. A full RNS variant of approximate homomorphic encryption. In Carlos Cid and Michael J.: Jacobson, Jr., editors, *SAC 2018: 25th Annual International Workshop on Selected Areas in Cryptography*, volume 11349 of *Lecture Notes in Computer Science*, pages 347–368, Calgary, AB, Canada, August 15–17, 2019. Springer, Cham, Switzerland.

CHKS25. Jung Hee Cheon, Guillaume Hanrot, Jongmin Kim, and Damien Stehlé. SHIP: A shallow and highly parallelizable CKKS bootstrapping algorithm. In Serge Fehr and Pierre-Alain Fouque, editors, *Advances in Cryptology – EUROCRYPT 2025, Part III*, volume 15603 of *Lecture Notes in Computer*

Science, pages 398–428, Madrid, Spain, May 4–8, 2025. Springer, Cham, Switzerland.

CKKL24. Heewon Chung, Hyojun Kim, Young-Sik Kim, and Yongwoo Lee. Amortized large look-up table evaluation with multivariate polynomials for homomorphic encryption. *IACR Cryptol. ePrint Arch.*, 2024:274, 2024.

CKKS17. Jung Hee Cheon, Andrey Kim, Miran Kim, and Yong Soo Song. Homomorphic encryption for arithmetic of approximate numbers. In Tsuyoshi Takagi and Thomas Peyrin, editors, *Advances in Cryptology – ASIACRYPT 2017, Part I*, volume 10624 of *Lecture Notes in Computer Science*, pages 409–437, Hong Kong, China, December 3–7, 2017. Springer, Cham, Switzerland.

CKSS25. Hyeongmin Choe, Jaehyung Kim, Damien Stehlé, and Elias Suvanto. Leveraging discrete CKKS to bootstrap in high precision. In Chun-Ying Huang, Jyh-Cheng Chen, Shiuh-Pyng Shieh, David Lie, and Véronique Cortier, editors, *ACM CCS 2025: 32nd Conference on Computer and Communications Security*, pages 1083–1097, Taipei, Taiwan, October 13–17, 2025. ACM Press.

CLOT21. Ilaria Chillotti, Damien Ligier, Jean-Baptiste Orfila, and Samuel Tap. Improved programmable bootstrapping with larger precision and efficient arithmetic circuits for TFHE. In Mehdi Tibouchi and Huaxiong Wang, editors, *Advances in Cryptology – ASIACRYPT 2021, Part III*, volume 13092 of *Lecture Notes in Computer Science*, pages 670–699, Singapore, December 6–10, 2021. Springer, Cham, Switzerland.

CPL26. Gyeongwon Cha, Dongjin Park, and Joon-Woo Lee. Improved radix-based approximate homomorphic encryption for large integers via lightweight bootstrapped digit carry. In Joan Daemen and Emmanuel Thomé, editors, *Advances in Cryptology – EUROCRYPT 2026*, pages 243–273, Cham, 2026. Springer Nature Switzerland.

DAP+26. Jules Dumezy, Andreea Alexandru, Yuriy Polyakov, Pierre-Emmanuel Clet, Olive Chakraborty, and Aymen Boudguiga. Evaluating larger lookup tables using ckks. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2026(1):559–591, Jan. 2026.

DM15. Léo Ducas and Daniele Micciancio. FHEW: Bootstrapping homomorphic encryption in less than a second. In Elisabeth Oswald and Marc Fischlin, editors, *Advances in Cryptology – EUROCRYPT 2015, Part I*, volume 9056 of *Lecture Notes in Computer Science*, pages 617–640, Sofia, Bulgaria, April 26–30, 2015. Springer Berlin Heidelberg, Germany.

DMPS24. Nir Drucker, Guy Moshkowich, Tomer Pelleg, and Hayim Shaul. BLEACH: Cleaning errors in discrete computations over CKKS. *Journal of Cryptology*, 37(1):3, January 2024.

Dum26. Jules Dumezy. Sparse key estimate. `https://github.com/jdumezy/sparse-key-estimate`, 2026.

FV12. Junfeng Fan and Frederik Vercauteren. Somewhat practical fully homomorphic encryption. Cryptology ePrint Archive, Report 2012/144, 2012.

Gen09. Craig Gentry. Fully homomorphic encryption using ideal lattices. In Michael Mitzenmacher, editor, *41st Annual ACM Symposium on Theory of Computing*, pages 169–178, Bethesda, MD, USA, May 31 – June 2, 2009. ACM Press.

GZ26. Mingyu Gao and Hongren Zheng. FHE for SIMD arithmetic logic units with amortized $O(1)$ bootstrapping per ciphertext. Cryptology ePrint Archive, Report 2026/233, 2026.

HS18. Shai Halevi and Victor Shoup. Faster homomorphic linear transformations in HElib. In Hovav Shacham and Alexandra Boldyreva, editors, *Advances in Cryptology – CRYPTO 2018, Part I*, volume 10991 of *Lecture Notes in Computer Science*, pages 93–120, Santa Barbara, CA, USA, August 19–23, 2018. Springer, Cham, Switzerland.

Kim25a. Jaehyung Kim. Efficient homomorphic integer computer from CKKS. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2025(4):873–898, 2025.

Kim25b. Jaehyung Kim. Faster homomorphic integer computer. Cryptology ePrint Archive, Report 2025/1440, 2025.

lat24. Pro7ech's lattigo. Online: `https://github.com/Pro7ech/lattigo`, November 2024. EPFL-LDS, Tune Insight SA, Jean-Philippe Bossuat.

LLL+21. Joon-Woo Lee, Eunsang Lee, Yongwoo Lee, Young-Sik Kim, and Jong-Seon No. High-precision bootstrapping of RNS-CKKS homomorphic encryption using optimal minimax polynomial approximation and inverse sine function. In Anne Canteaut and François-Xavier Standaert, editors, *Advances in Cryptology – EUROCRYPT 2021, Part I*, volume 12696 of *Lecture Notes in Computer Science*, pages 618–647, Zagreb, Croatia, October 17–21, 2021. Springer, Cham, Switzerland.

Ogi26. Tabitha Ogilvie. On the concrete hardness gap between MLWE and LWE. Cryptology ePrint Archive, Paper 2026/279, 2026.

SSKM25. Hyewon Sung, Sieun Seo, Taekyung Kim, and Chohong Min. Evalround+ bootstrapping and its rigorous analysis for CKKS scheme. *IEEE Access*, 13:140847–140866, 2025.

A CKKS Operations and Bootstrapping

We collect the homomorphic operations used by the constructions of the paper, together with a short description of CKKS bootstrapping. Throughout, a ciphertext at level ℓ is a pair $\mathsf{ct} = (b, a) \in R_{Q_\ell}^2$ with $b + as \approx m + e \bmod Q_\ell$ for some small error e and some plaintext $m \in R$ with scaling factor Δ. When two ciphertexts have the same level we operate at that level directly, otherwise we first apply $\mathsf{DropLevels}$ to align them.

A.1 Operations

Key switching. For a source key s and a target key s', a key-switching key $\mathsf{swk}_{s\to s'}$ is a gadget-decomposed encryption of s under s'. The operation

$$\mathsf{KeySwitch}(\mathsf{ct}, \mathsf{swk}_{s\to s'})$$

converts an encryption of m under s into an encryption of m under s' at the same level, at the cost of one gadget-product per ciphertext. The relinearization key rlk switches from s^2 to s, the conjugation key cnk switches from $s(X^{-1})$ to s, and the Galois key gk_k switches from $s(X^{5^k})$ to s for rotations.

Linear operations. For $\mathsf{ct}_1 = (b_1, a_1)$ and $\mathsf{ct}_2 = (b_2, a_2)$ at level ℓ with common scaling factor Δ, a plaintext $\mathsf{pt} \in R$, and a complex constant $z \in \mathbb{C}$ encoded as a scaled plaintext $\mathsf{pt}_z = \mathsf{Encode}_\Delta(z)$:

$$\mathsf{Add}(\mathsf{ct}_1, \mathsf{ct}_2) = (b_1 + b_2,\, a_1 + a_2) \bmod Q_\ell,$$

$$\mathsf{AddPlain}(\mathsf{ct}_1, \mathsf{pt}) = (b_1 + \mathsf{pt},\, a_1) \bmod Q_\ell,$$

$$\mathsf{AddConst}(\mathsf{ct}_1, z) = \mathsf{AddPlain}(\mathsf{ct}_1, \mathsf{pt}_z).$$

None of these operations consume a multiplicative level, and they preserve Δ.

Plaintext multiplication. For a plaintext pt with scaling factor Δ,

$$\mathsf{MulPlain}(\mathsf{ct}_1, \mathsf{pt}) = (b_1 \cdot \mathsf{pt},\, a_1 \cdot \mathsf{pt}) \bmod Q_\ell.$$

The output has scaling factor Δ^2 and is brought back close to Δ by a subsequent $\mathsf{Rescale}$, which drops one level. The same procedure with $\mathsf{pt} = \mathsf{Encode}_\Delta(z)$ realizes $\mathsf{MulConst}(\mathsf{ct}_1, z)$.

Ciphertext multiplication and rescaling. For ct_1 and ct_2 at level ℓ with scaling factor Δ, we compute the tensor

$$(d_0, d_1, d_2) = (b_1 b_2,\, a_1 b_2 + a_2 b_1,\, a_1 a_2) \bmod Q_\ell$$

and relinearize using rlk:

$$\mathsf{Mul}(\mathsf{ct}_1, \mathsf{ct}_2) = (d_0, d_1) + \mathsf{KeySwitch}(d_2, \mathsf{rlk}) \bmod Q_\ell.$$

The output has scaling factor Δ^2. The operation $\mathsf{Rescale}(\mathsf{ct})$ returns $\lfloor \mathsf{ct}/q_\ell \rceil$ at level $\ell - 1$ with scaling factor Δ^2/q_ℓ. For $q_\ell \approx \Delta$, this brings the output back to a scaling factor close to Δ.

Rotation and conjugation. A slot rotation by k steps applies the Galois automorphism $X \mapsto X^{5^k}$ to the polynomials of ct_1 and then $\mathsf{KeySwitch}$ with gk_k to restore the secret s. Conjugation applies $X \mapsto X^{-1}$ and then $\mathsf{KeySwitch}$ with cnk. Neither operation consumes a multiplicative level.

A.2 Bootstrapping

CKKS bootstrapping refreshes an exhausted ciphertext to a higher level, so that further multiplications can be performed. For an encryption of m at modulus q_0, it returns an encryption of $\hat{m} \approx m$ at a higher modulus Q_{high}, where Q_{high} is determined by the bootstrapping parameters. The procedure has four steps.

- $\mathsf{ModRaise}$: raise the ciphertext modulus from q_0 to Q_L without changing the polynomial, giving an encryption of $\tilde{m} = m + q_0 I$ for some small $I \in \mathbb{Z}[X]$.
- $\mathsf{CoeffsToSlots}$: homomorphically apply τ^{-1} to place each coefficient $\tilde{m}_i$ of $\tilde{m}$ into its own slot.

- EvalMod: approximate $x \mapsto x \bmod q_0$ on each slot by a polynomial approximation of $x \mapsto (q_0/2\pi)\sin(2\pi x/q_0)$, which equals $x \bmod q_0$ when $q_0 \gg \|m\|_\infty$. After this step each slot holds $\hat{m}_i \approx m_i$.
- SlotsToCoeffs: homomorphically apply τ to return slot data to the coefficient representation, giving an encryption of $\hat{m}$.

The resulting ciphertext has level $L - L_{\mathsf{CoeffsToSlots}} - L_{\mathsf{EvalMod}} - L_{\mathsf{SlotsToCoeffs}}$, where $L_\bullet$ is the number of levels consumed by the corresponding step.

The constructions of this paper do not depend on a specific bootstrapping procedure. Bootstrapping is invoked only to regain multiplicative depth, and any CKKS bootstrapping in the literature can be used. Functional bootstrappings [BKSS24,AKP25] replace the sine by a complex exponential – which shares 2π periodicity – followed by a Hermite interpolation of degree $t-1$. Additionally, it usually is a SlotsToCoeffs first bootstrapping, that is that the order of operation is SlotsToCoeffs $\to$ ModRaise $\to$ CoeffsToSlots $\to$ EvalMod.

B Switching from discrete CKKS to a character block

We detail a depth-saving variant used to switch a standard discrete CKKS ciphertext encoding values in $\mathbb{Z}_t$ slot-wise into split-layout BRU. The input may equivalently be the post-ModRaise ciphertext of a bootstrapping after SlotsToCoeffs, ModRaise, and CoeffsToSlots, whose slots carry $m + q_0 I$ for the overflow polynomial I. The EvalExp stage absorbs that overflow as part of producing ζ_t^m.

A direct realization computes the base coordinate ζ_t^m via the EvalExp stage of a functional bootstrapping and then forms the remaining $t-2$ coordinates by a balanced power-basis tree of $\lceil \log_2(t-1) \rceil$ ciphertext multiplications. The variant exploits the structure of EvalExp. We use the construction of [BKSS24], which evaluates the sine and cosine of $2\pi x/t$ in parallel as two polynomial approximations and recombines them into $\zeta_t^m = \cos(2\pi m/t) + i\sin(2\pi m/t)$. Scaling the input by an integer k before these interpolations produces ζ_t^{km} at the same depth as ζ_t^m. Running K such pre-scaled EvalExp evaluations in parallel on ct therefore returns the first K BRU coordinates $\zeta_t^m, \ldots, \zeta_t^{Km}$ at the depth of a single EvalExp, with no extra multiplicative level.

The remaining coordinates ζ_t^{km} for $k \in [K+1, t-1]$ are written as $k = qK + r$ with $r \in [1, K]$ and $q \geq 1$, and recovered as

$$\zeta_t^{km} = \zeta_t^{rm} \cdot \zeta_t^{qKm}.$$

A balanced power basis of ζ_t^{Km} up to exponent $\kappa = \lceil (t-1)/K \rceil - 1$ is computed in $\lceil \log_2 \kappa \rceil$ levels, after which each remaining coordinate costs one cross-multiplication. The total depth above the functional bootstrapping is $\lceil \log_2 \kappa \rceil + 1$, against $\lceil \log_2(t-1) \rceil$ for the direct power-basis tree. For $t = 256$ and $K = 16$ this is $4 + 1 = 5$ levels instead of 8. The default parallelism we use is $K = 16$.

The complete procedure is Algorithm 3. Any other block encoding is reached from the returned BRU block by one BLT.

Algorithm 3 Switch from standard discrete CKKS to a split-layout BRU_t block

procedure $\mathsf{ToBRU}(\mathsf{ct} \in R^2_{Q_\ell}, K)$ with each slot of ct encoding a value in $\mathbb{Z}_t$ and parallelism budget $1 \le K \le t-1$

for $k \in [\![1, K]\!]$ **in parallel do**
  $\mathsf{ct}^{(k)} \leftarrow \mathsf{EvalExp}_t(\mathsf{MulInt}(\mathsf{ct}, k))$
if $K \ge t-1$ **then**
  return $(\mathsf{ct}^{(1)}, \dots, \mathsf{ct}^{(t-1)})$
$\kappa \leftarrow \lceil (t-1)/K \rceil - 1$
$\pi^{(1)} \leftarrow \mathsf{ct}^{(K)}$
for $q \in [\![2, \kappa]\!]$ in the order of a balanced product tree on $[1, \kappa]$ **do**
  pick $q', q'' \ge 1$ with $q' + q'' = q$
  $\pi^{(q)} \leftarrow \mathsf{Mul}(\pi^{(q')}, \pi^{(q'')})$
for $k \in [\![K+1, t-1]\!]$ **do**
  write $k = qK + r$ with $r \in [1, K]$ and $q \ge 1$
  $\mathsf{ct}^{(k)} \leftarrow \mathsf{Mul}(\mathsf{ct}^{(r)}, \pi^{(q)})$
return $(\mathsf{ct}^{(1)}, \dots, \mathsf{ct}^{(t-1)})$ as the split-layout BRU_t representation

C Proof of Proposition 2

Proof. Let

$$\mathcal{B} = \mathrm{End}(\mathcal{S}) \times \mathcal{Y}^{\mathcal{S}}$$

be the set of bundled summaries. For a finite interval of positions $[a, b]$, define the ideal summary

$$B_{[a,b]} = (F_{[a,b]}, \Phi_{[a,b]})$$

by

$$F_{[a,b]} = F_b \circ F_{b-1} \circ \cdots \circ F_a$$

and

$$\Phi_{[a,b]} = \Phi_b \circ F_{b-1} \circ \cdots \circ F_a,$$

with the convention that $\Phi_{[a,a]} = \Phi_a$. Thus $F_{[a,b]}$ is the state transition through the whole interval, whereas $\Phi_{[a,b]}$ is the output at the right endpoint b as a function of the state entering the interval at a.

We first prove the stronger claim that for every interval $[a, b]$, every parenthesization of

$$B_a \diamond B_{a+1} \diamond \cdots \diamond B_b$$

evaluates to $B_{[a,b]}$. The claim is clear when $a = b$. Assume now that it holds for all intervals of length smaller than $b - a + 1$, and consider an arbitrary parenthesization of the product on $[a, b]$. Its last operation splits the interval at some r with $a \le r < b$, so the product has the form

$$(B_a \diamond \cdots \diamond B_r) \diamond (B_{r+1} \diamond \cdots \diamond B_b)\,.$$

By the induction hypothesis, the two factors are respectively

$$B_{[a,r]} = (F_{[a,r]}, \Phi_{[a,r]}) \qquad \text{and} \qquad B_{[r+1,b]} = (F_{[r+1,b]}, \Phi_{[r+1,b]}).$$

Therefore, by the definition of $\diamond$,

$$B_{[a,r]} \diamond B_{[r+1,b]} = \left(F_{[r+1,b]} \circ F_{[a,r]}, \Phi_{[r+1,b]} \circ F_{[a,r]}\right).$$

The first component is

$$F_b \circ \cdots \circ F_{r+1} \circ F_r \circ \cdots \circ F_a = F_{[a,b]}.$$

The second component is

$$\Phi_b \circ F_{b-1} \circ \cdots \circ F_{r+1} \circ F_r \circ \cdots \circ F_a = \Phi_{[a,b]}.$$

Hence the product equals $B_{[a,b]}$, independently of the split r and independently of the parenthesization. This proves the claim for $[a, b]$ by induction.

Taking $b = a + 2$ gives associativity of $\diamond$. More importantly, the same argument shows that every prefix product

$$B_0 \diamond B_1 \diamond \cdots \diamond B_k$$

is well defined and equals

$$\left(F_k \circ \cdots \circ F_0, \Phi_k \circ F_{k-1} \circ \cdots \circ F_0\right).$$

Thus the output at position k, for initial state c_{-1}, is precisely the stored value

$$\Phi_{[0,k]}(c_{-1}).$$

It remains to justify the level count. Let $B_l = (F_l, \Phi_l)$ and $B_h = (F_h, \Phi_h)$. In non-reduced IDCT table form, the transition component of $B_l \diamond B_h$ is

$$T_{F_h \circ F_l}(c, r) = \sum_{s \in \mathcal{S}} T_{F_l}(c, s) T_{F_h}(s, r).$$

For each output slot j, the output component is

$$(\Phi_h \circ F_l)_j(c) = \sum_{s \in \mathcal{S}} T_{F_l}(c, s) \Phi_{h,j}(s).$$

Both formulas use the same one-hot selector $T_{F_l}(c, \cdot)$. All products appearing in these formulas can be evaluated in one parallel ciphertext-ciphertext multiplication layer. The sums over s are plaintext linear combinations with coefficients in $\{0, 1\}$ and therefore do not add multiplicative depth. Hence one bundled composition costs one multiplicative level. □

D Proof of Lemma 1

We prove that Algorithm 2 computes the digits of $(x + y) \bmod t^d$ when x and y are encoded as d BRU-t digit blocks per word. Let $\omega = \exp(2\pi i/t)$ so that the j-th coordinate of $\mathsf{BRU}_t(m)$ is $\omega^{(j+1)m}$ for $j \in [0, t-2]$. Throughout, equalities between BRU vectors are coordinate-wise, $\odot$ denotes the Hadamard product, and we identify a digit block in a ciphertext slot range with the BRU vector it carries.

Carry-free branch. Since BRU_t is a character on $(\mathbb{Z}_t, +)$, slot-wise multiplication realizes addition modulo t block by block. With $u_k = x_k + y_k$ and $s_k = u_k \bmod t$,

$$\mathsf{BRU}_t(x_k) \odot \mathsf{BRU}_t(y_k) = \mathsf{BRU}_t(s_k).$$

Step 1 of Algorithm 2 therefore produces $\mathsf{ct}_{\mathrm{rs}}$ with $\mathsf{BRU}_t(s_k)$ at digit position k of every batched word.

BRU increment identity. For every $s \in \mathbb{Z}_t$,

$$\mathsf{BRU}_t(s) + \big(\mathsf{BRU}_t(1) - \mathsf{BRU}_t(0)\big) \odot \mathsf{BRU}_t(s) = \mathsf{BRU}_t(s + 1 \bmod t).$$

Indeed, on coordinate j the left-hand side equals $\omega^{(j+1)s}\big(1 + \omega^{j+1} - 1\big) = \omega^{(j+1)(s+1)}$, which is the j-th coordinate of $\mathsf{BRU}_t(s+1 \bmod t)$. Set $\boldsymbol{\delta} = \mathsf{BRU}_t(1) - \mathsf{BRU}_t(0)$. Then for any $c \in \{0, 1\}$,

$$\mathsf{BRU}_t(s) + c \cdot \boldsymbol{\delta} \odot \mathsf{BRU}_t(s) = \mathsf{BRU}_t(s + c \bmod t).$$

Per-digit signals. By construction of the bivariate LUT, the slot range of digit position k in ct_G holds either $\boldsymbol{\delta}$ or the zero vector, and in ct_P either $\mathbf{1}$ or the zero vector, with

$$G_k = g_k \cdot \boldsymbol{\delta}, \qquad P_k = p_k \cdot \mathbf{1}, \qquad g_k = \delta_{u_k \geq t}, \quad p_k = \delta_{u_k = t-1}.$$

The pre-shift by $-S$ on the LUT inputs is the index shift that aligns each carry signal with its consumption point. After the pre-shift, the slot range of digit position k of $\mathsf{ct}_{x'}$ holds $\mathsf{BRU}_t(x_{k-1})$ and likewise for $\mathsf{ct}_{y'}$, so the LUT output at position k encodes (G_{k-1}, P_{k-1}). This is exactly the carry that digit $k-1$ contributes to the carry entering digit k, hence the comment "the LUT emits carry-into directly" in the source code. The Kogge–Stone scan then composes these signals across positions, and after $\lceil \log_2 d \rceil$ rounds the slot range of digit k holds the prefix carry that arrives at digit k from the digits below. The block-zero plaintext mask used later suppresses any spurious carry that this alignment could otherwise inject into digit 0 of each batched word.

Brent–Kung scan. Define the scan composition on pairs of vectors by

$$(G, P) \diamond (G', P') = \big(G + P \odot G',\ P \odot P'\big).$$

The vector substructure of G and P collapses under $\diamond$. Since $P \in \{0 \cdot \mathbf{1}, \mathbf{1}\}$, the product $P \odot G'$ equals G' if $P = \mathbf{1}$ and 0 if $P = 0$, and $P \odot P' \in \{0 \cdot \mathbf{1}, \mathbf{1}\}$. Writing $G_k = g_k \boldsymbol{\delta}$ and $P_k = p_k \mathbf{1}$, the scan reduces to the scalar recurrence

$$(g_k, p_k) \diamond (g_{k'}, p_{k'}) = (g_k + p_k g_{k'},\ p_k p_{k'})$$

on the scalar pair (g_k, p_k).

We verify that this scalar recurrence stays in $\{0, 1\}^2$ and matches the textbook OR-based Brent–Kung carry recurrence. At a single digit, $g_k = 1$ forces

$u_k \geq t$ and $p_k = 1$ forces $u_k = t - 1$, so g_k and p_k are mutually exclusive. Assume by induction that the composed pair (g, p) has mutually exclusive components, and likewise (g', p'). If the new propagate $pp' = 1$ then $p = p' = 1$, hence $g = g' = 0$ by the induction hypothesis, so the new generate $g + pg' = 0$. Thus mutual exclusivity is preserved by composition. In particular $g + pg'$ never exceeds 1, because $g = 1$ implies $p = 0$ and forces the second summand to vanish. The scalar recurrence is therefore boolean and coincides with the standard Brent–Kung carry recurrence.

After $\lceil \log_2 d \rceil$ rounds, the abstract scan computes the prefix carries

$$g^{[k]} = g_k + p_k g_{k-1} + p_k p_{k-1} g_{k-2} + \cdots + p_k p_{k-1} \cdots p_1 g_0,$$

which is the carry leaving digit k, equivalently the carry entering digit $k + 1$. With the pre-shift of the previous paragraph, the ciphertext slot range of digit position k of the scan output holds $g^{[k-1]} \cdot \boldsymbol{\delta}$, where the convention $g^{[-1]} = 0$ accounts for the padding signal $(G_{-1}, P_{-1}) = (0, 0)$ at position 0.

Correction. Let $c_k \in \{0, 1\}$ denote the carry entering digit k, so that $c_0 = 0$ and $c_k = g^{[k-1]}$ for $k \geq 1$. By the pre-shift alignment, the value at position k of the scan output is exactly $c_k \cdot \boldsymbol{\delta}$. Step 9 of Algorithm 2 computes ct_δ, whose value at digit k is $\mathsf{BRU}_t(s_k)$ if $k \geq 1$ and zero if $k = 0$, by virtue of the block-zero plaintext mask. Step 10 then forms $\mathsf{ct}_{\mathrm{corr}}$ whose value at digit k is

$$(c_k \cdot \boldsymbol{\delta}) \odot \mathsf{ct}_\delta|_k = c_k \cdot \boldsymbol{\delta} \odot \mathsf{BRU}_t(s_k) \quad \text{for } k \geq 1$$

and zero for $k = 0$. The final Add on step 11 returns at digit k the value

$$\mathsf{BRU}_t(s_k) + c_k \cdot \boldsymbol{\delta} \odot \mathsf{BRU}_t(s_k) = \mathsf{BRU}_t\big(s_k + c_k \bmod t\big)$$

by the BRU increment identity. Since $s_k + c_k \bmod t$ is precisely the k-th digit of $(x + y) \bmod t^d$ for the carry c_k produced by Brent–Kung, the algorithm is correct. □

E Worked example: 3-digit decimal addition

We trace Algorithm 2 on a small instance of the addition construction of Section 6.1. Take radix $t = 10$, word length $d = 3$, and add $x = 247$ and $y = 158$. The expected sum is $z = 405$, with digits $z_0 = 5$, $z_1 = 0$, $z_2 = 4$.

Per-digit signals. With $u_k = x_k + y_k$, the carry-free digit is $s_k = u_k \bmod 10$, the generate bit is $g_k = \delta_{u_k \geq 10}$, and the propagate bit is $p_k = \delta_{u_k = 9}$.

The raw-sum ciphertext $\mathsf{ct}_{\mathrm{rs}}$ encodes $\mathsf{BRU}_{10}(s_k)$ at digit k, obtained from one ciphertext multiplication of $\mathsf{BRU}_{10}(x)$ and $\mathsf{BRU}_{10}(y)$. The bivariate LUT applied to the pre-shifted inputs emits ct_G and ct_P, where the G_k slot block is the BRU delta $\mathsf{BRU}_{10}(1) - \mathsf{BRU}_{10}(0)$ if $g_k = 1$ and zero otherwise, and the P_k slot block is the all-ones character mask if $p_k = 1$ and zero otherwise.

k	(x_k, y_k)	u_k	s_k	g_k	p_k
0	$(7, 8)$	15	5	1	0
1	$(4, 5)$	9	9	0	1
2	$(2, 1)$	3	3	0	0

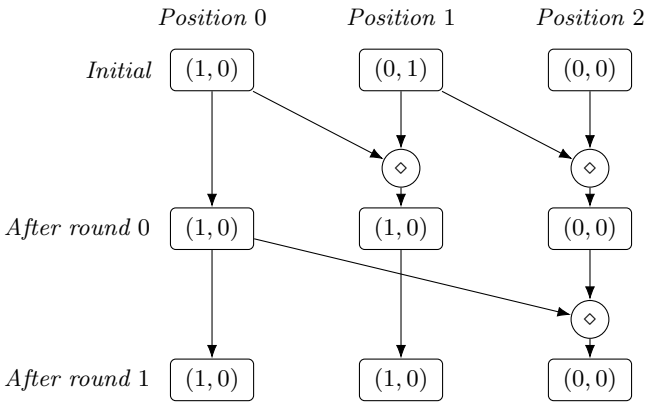

Fig. 4. Brent–Kung scan for $d = 3$ traced on the (g, p) summaries of the worked example. The combine is $(g, p) \diamond (g', p') = (g + p\,g', p\,p')$, drawn as $\diamond$. At round r, position k combines with position $k - 2^r$; positions whose source falls below 0 keep their value, since the zero padding contributes nothing through $\diamond$.

Brent–Kung scan. With $d = 3$ we use $\lceil \log_2 3 \rceil = 2$ rounds. Figure 4 traces the scalar (g_k, p_k) component at each position, since the BRU shape of each signal is identical across rounds.

After two rounds, the g-values are the carry prefix. The carry entering digit 0 is fixed at 0, the carry entering digit 1 is $g^{[0]} = 1$ (produced by $u_0 = 15$), and the carry entering digit 2 is $g^{[1]} = 1$ (propagated from below through $u_1 = 9$). The final $g^{[2]} = 0$ confirms no carry leaves the word.

Correction. The block-zero mask zeros out the lowest-digit block of $\mathsf{ct}_{\mathrm{rs}}$, then the masked ciphertext is multiplied by ct_G and added back. At digit $k \geq 1$, the correction adds $\mathsf{BRU}_{10}(1) - \mathsf{BRU}_{10}(0)$ when $g^{[k-1]} = 1$ and zero otherwise, which is exactly the BRU increment by the incoming carry. Reading the output:

$$z_0 = s_0 = 5, \qquad z_1 = (s_1 + 1) \bmod 10 = 0, \qquad z_2 = s_2 + 1 = 4,$$

so $z = 405$ as expected.

Depth. The bivariate LUT consumes 3 levels. The two scan rounds consume 2 levels. The final correction multiplication consumes 1 level. The total is $4 + \lceil \log_2 3 \rceil = 6$ levels, matching the bound of Section 6.1.

F Detailed benchmarks

We collect the full $\log_2 t$ scans of the primitive operations of Section 7.1 and of the AKP functional bootstrapping comparison. All numbers are single-thread, with a scaling factor $\log_2 \Delta = 40$ and packed layout.

Table 9. Full $\log_2 t$ scan of BRU-encoded operations, packed layout. Following Table 4, the value count is $\lfloor N/2(t-1) \rfloor$ for LUT and Cleaning, $\lfloor N/2t^2 \rfloor$ for the bivariate LUT, and $\lfloor N/2t^4 \rfloor$ for the 4-variate LUT.

Operation	$\log_2 t$	$\log_2 N$	$\log_2 QP$	Values	Latency	Amtz. time
LUT	2	15	155	5461	8.7 ms	2 μs
LUT	3	15	155	2340	15.8 ms	7 μs
LUT	4	15	155	1092	24.2 ms	22 μs
LUT	5	15	155	528	34.9 ms	66 μs
LUT	6	15	155	260	53.5 ms	206 μs
LUT	7	15	155	129	80.1 ms	621 μs
LUT	8	15	155	64	122.5 ms	1.9 ms
LUT	9	15	155	32	203.8 ms	6.4 ms
LUT	10	15	155	16	342.3 ms	21.4 ms
Bivariate LUT	2	15	235	1024	102.1 ms	100 μs
Bivariate LUT	3	15	235	256	170.5 ms	666 μs
Bivariate LUT	4	15	235	64	324.2 ms	5.1 ms
Bivariate LUT	5	15	235	16	725.0 ms	45.3 ms
4-variate LUT	2	15	275	64	943.5 ms	14.7 ms
4-variate LUT	3	15	275	4	5.21 s	1.30 s
Cleaning	2	15	275	5461	56.4 ms	10 μs
Cleaning	3	15	275	2340	87.9 ms	38 μs
Cleaning	4	15	275	1092	123.6 ms	113 μs
Cleaning	5	15	275	528	163.1 ms	309 μs
Cleaning	6	15	275	260	243.4 ms	936 μs
Cleaning	7	15	275	129	338.9 ms	2.6 ms
Cleaning	8	15	275	64	507.9 ms	7.9 ms
Cleaning	9	15	275	32	801.2 ms	25.0 ms
Cleaning	10	15	275	16	1.28 s	80.1 ms

Table 10. Cross-encoding LUT and Cleaning at $\log_2 t \in \{2, 4, 8\}$, packed layout, $\log N = 15$. The BRU rows are in Table 4. L-BRU requires a prime modulus and falls back to alphabets of size 3, 13, and 251 at the nominal $\log_2 t \in \{2, 4, 8\}$, rather than 4, 16, 256. The smaller alphabets use $p - 1$ slots per block, which inflates the packing relative to WH and IDCT at the same nominal $\log_2 t$; the comparison is therefore not strictly like-for-like.

Operation	Encoding	$\log_2 t$	$\log_2 QP$	Values	Latency	Amtz. time
LUT	L-BRU	2	155	8192	7.7 ms	1 μs
LUT	L-BRU	4	155	1365	20.5 ms	15 μs
LUT	L-BRU	8	155	65	124.3 ms	1.9 ms
LUT	WH	2	155	5461	8.8 ms	2 μs
LUT	WH	4	155	1092	24.2 ms	22 μs
LUT	WH	8	155	64	124.8 ms	1.9 ms
LUT	IDCT	2	155	5461	8.6 ms	2 μs
LUT	IDCT	4	155	1092	19.8 ms	18 μs
LUT	IDCT	8	155	64	107.4 ms	1.7 ms
Cleaning	L-BRU	2	275	8192	54.6 ms	7 μs
Cleaning	L-BRU	4	275	1365	109.0 ms	80 μs
Cleaning	L-BRU	8	275	65	522.0 ms	8.0 ms
Cleaning	WH	2	195	5461	12.6 ms	2 μs
Cleaning	WH	4	195	1092	12.5 ms	11 μs
Cleaning	WH	8	195	64	12.5 ms	195 μs
Cleaning	IDCT	2	195	5461	12.5 ms	2 μs
Cleaning	IDCT	4	195	1092	12.6 ms	12 μs
Cleaning	IDCT	8	195	64	12.6 ms	197 μs

Table 11. Full $\log_2 t$ scan of the AKP functional bootstrapping [AKP25] in OpenFHE v1.5.1, single thread. The number of values is the full ring dimension N.

$\log_2 t$	$\log_2 N$	$\log_2 QP$	$\log_2 \Delta$	Values	Latency	Amtz. time
2	16	1036	35	65536	6.70 s	102 μs
3	16	1115	37	65536	7.22 s	110 μs
4	16	1234	38	65536	8.51 s	130 μs
5	16	1393	43	65536	9.40 s	143 μs
6	16	1520	44	65536	10.95 s	167 μs
7	16	1524	46	65536	13.46 s	205 μs
8	16	1656	47	65536	17.31 s	264 μs
9	16	1728	48	65536	22.66 s	346 μs
10	17	2078	53	131072	75.73 s	578 μs

Table 12. Conversion between standard discrete CKKS and the split-layout BRU_t representation, single thread. The forward direction is a single BLT at $\log_2 QP = 155$. The reverse direction is the depth-saving variant of Appendix B and runs in the $\log_2 QP = 770$ chain that accommodates one functional bootstrapping plus the power-basis tree. The level cost of the reverse direction grows from 9 at $\log_2 t = 2$ to 13 at $\log_2 t = 7$ as the power-basis tree widens. Each conversion processes $N/2$ values, since the split layout dedicates one slot to each character coordinate.

Operation	$\log_2 t$	$\log_2 N$	$\log_2 QP$	Levels	Latency	Amtz. time
Split BRU → standard	2	15	155	1	1.4 ms	86 ns
Split BRU → standard	3	15	155	1	2.4 ms	144 ns
Split BRU → standard	4	15	155	1	4.1 ms	252 ns
Split BRU → standard	5	15	155	1	7.5 ms	456 ns
Split BRU → standard	6	15	155	1	13.9 ms	846 ns
Split BRU → standard	7	15	155	1	27.3 ms	1.7 μs
Split BRU → standard	8	15	155	1	54.9 ms	3.3 μs
Standard → split BRU	2	15	770	9	5.16 s	315 μs
Standard → split BRU	3	15	770	9	8.29 s	506 μs
Standard → split BRU	4	15	770	9	14.62 s	892 μs
Standard → split BRU	5	15	770	10	15.61 s	953 μs
Standard → split BRU	6	15	770	12	16.13 s	985 μs
Standard → split BRU	7	15	770	13	17.36 s	1.06 ms