LUTune: Automated Lookup Table Optimization for Efficient TFHE-Based ML Inference

Rostin Shokri and Nektarios Georgios Tsoutsos

University of Delaware
{rostinsh, tsoutsos}@udel.edu

Abstract. TFHE evaluates quantized activation functions by programmable bootstrapping (PBS) using lookup tables (LUTs). The input precision of a LUT and the noise tolerance of its consumers affect PBS latency and evaluation-key size. LUTune rewrites activation LUTs using the outputs permitted by Concrete ML's approximate rounding rule. It folds LUTs to reduce their input precision and decomposes plaintext weights into signed digits to reduce the output-noise variance predicted for dense layers. LUTune checks the circuit failure bound under Concrete's noise model at the selected parameters. On VGG6 it achieves a 5.2× speedup and a 17.2× reduction in evaluation-key size relative to the lowest-latency Concrete ML configuration, using the same frozen weights and quantizers.

1 Introduction

Fully homomorphic encryption (FHE) allows a server to compute on encrypted data [13]. In private inference, a client encrypts its input, the server runs a neural network on the ciphertexts without holding the secret key, and only the client can decrypt the prediction. TFHE [8] evaluates a univariate function represented by a lookup table (LUT) and refreshes the ciphertext in one programmable bootstrap (PBS) [4, 9]. A PBS is efficient only on small integers, because its cost grows quickly with the number of input bits. Concrete ML [40] therefore quantizes a trained network to low-bit integers, so that each activation, merged with its requantization into one table, takes a single PBS, and lets the Concrete compiler [39] choose every cryptographic parameter for a target failure probability [3].

PBS evaluations dominate inference latency and account for much of the evaluation-key size. Increasing LUT input precision typically requires a larger polynomial size N to meet the same failure target (Section 2.1). The baseline CIFAR-10 VGG6 circuit performs 41,088 PBS evaluations of LUTs with 7-bit input precision per image. Its lowest-latency Concrete ML configuration takes 862 s on 64 threads and needs 32.5 GB of evaluation keys, which the client generates and uploads and the server stores for every client. The minimum-key-size configuration needs 5.3 GB and takes 1,445 s.

A quantized activation can be constant across many LUT input indices. Even where its value changes, approximate rounding may permit either of two neighboring entries for a given layer accumulator. Inputs can therefore share an entry if its value is allowed for each of them. Concrete selects parameters for the original LUT input precision (Section 2.2). A subsequent dense layer can also amplify the output noise of its PBS evaluations enough to require additional blind-rotation decomposition levels. Accuracy measurements alone cannot establish that a rewrite respects the original rounding rule on every input where the baseline circuit is defined.

Our baseline is the circuit produced by unmodified Concrete ML for the same frozen weights, quantizers and calibration inputs. LUTune rewrites this circuit before Concrete selects cryptographic parameters (Fig. 1). We compare at the same estimated 128-bit security and target circuit failure probability of 2^{-40} (Section 2.3). Relative to the lowest-latency VGG6 baseline, LUTune achieves a 5.2× speedup and a 17.2× reduction in evaluation-key size. Its keys are also 2.8× smaller than those of the minimum-key-size baseline. Our contributions are as follows:

– **LUT folding with auxiliary lookups.** LUTune reduces LUT input precision by mapping several input indices to one entry whenever they admit a common output. An input-offset LUT remaps constant input intervals. For activations that vary across most of the domain, LUTune decomposes the function into coarse and residual terms (Sections 3.2 to 3.4).

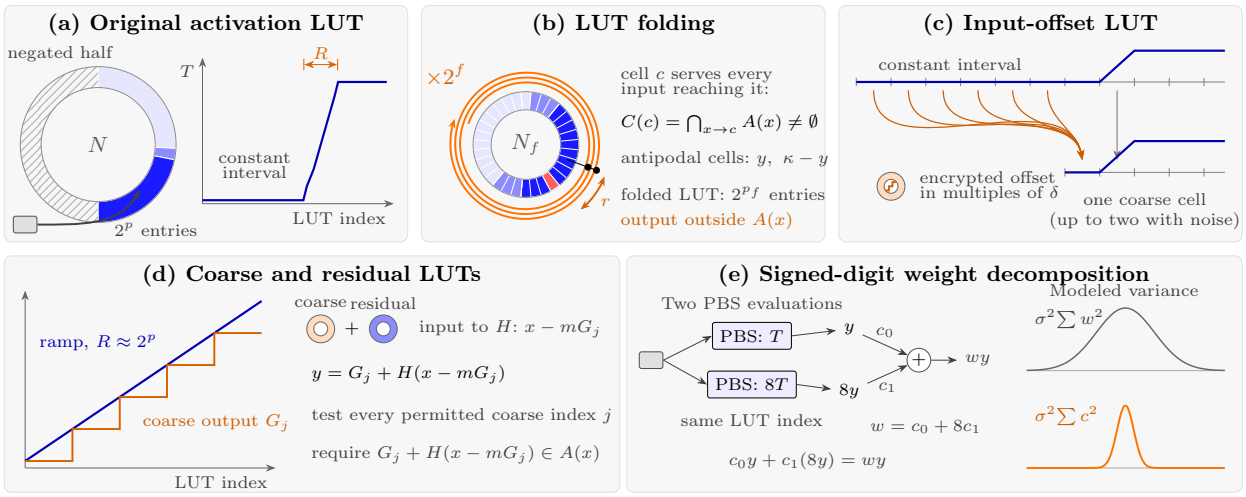

Fig. 1. LUTune overview: (a) an activation LUT with repeated values; (b) folding onto entries allowed for every input reaching them, from polynomial size N to N_f, with margin r to a disallowed output; (c) an encrypted input offset; (d) coarse and residual LUTs; (e) signed-digit weight decomposition, whose PBS evaluations on one ciphertext select the same LUT index. In the VGG6 classifier, (e) lowers the blind-rotation decomposition levels Concrete selects.

- **Signed-digit weight decomposition.** We express plaintext weights in signed digits. Separate PBS evaluations select the same index in scaled activation LUTs. Combining their outputs with the weight digits reconstructs the dense-layer sum and can reduce its modeled output-noise variance (Section 3.5).
- **An automated, semantics-preserving compilation flow.** From bounds on LUT inputs and the allowed rounding outputs, we derive the index displacement each lookup tolerates. We use these margins and the parameters selected by Concrete to bound the probability of circuit failure. LUTune emits standard PBS and linear operations and retains only rewrites that meet the target (Sections 3.1 and 3.6).

Roadmap: Section 2 reviews TFHE, programmable bootstrapping and quantized inference with Concrete ML. Section 3 presents the LUTune compiler, its LUT and dot-product rewrites, and the failure bound. Section 4 compares LUTune with Concrete ML, Section 5 discusses related work, and Section 6 concludes.

2 Background

2.1 TFHE and Programmable Bootstrapping

TFHE [8] encrypts a p-bit message m in a learning with errors (LWE) [29] ciphertext $\mathsf{ct} = (a_1, \ldots, a_n, b) \in \mathbb{Z}_q^{n+1}$ with $b = \sum_{i=1}^{n} a_i s_i + \Delta m + e$, where $\boldsymbol{s} \in \{0,1\}^n$ is the secret key, e is a Gaussian error term, $q = 2^{64}$, and the scaling factor $\Delta = q/2^{p+1}$ keeps one padding bit above the message [3,9]. Decryption rounds $b - \langle \boldsymbol{a}, \boldsymbol{s} \rangle$ to the nearest multiple of Δ and fails once $|e| \geq \Delta/2$. Ciphertext addition and multiplication by a plaintext integer propagate ciphertext noise.

PBS builds on the bootstrapping operation of FHEW [12]. In Concrete, key switching with a key-switching key (KSK) first produces an LWE ciphertext of dimension n. Modulus switching maps its coefficients to $\mathbb{Z}_{2N}$, whose phase determines a rotation index modulo $2N$. Blind rotation applies that index to the generalized LWE (GLWE) accumulator initialized with the LUT's test polynomial $P_L \in \mathbb{Z}_q[X]/(X^N+1)$. The bootstrapping key (BSK) contains GGSW encryptions of the input secret-key bits under a GLWE key of dimension k and polynomial size N [8,9]. Sample extraction returns the selected coefficient in an LWE ciphertext.

A LUT with input precision p has 2^p entries. In the baseline encoding, each entry occupies a cell of $N/2^p$ consecutive rotation indices. The padding bit places valid message centers in the first half of the rotation domain. Since $X^N = -1$, the second half represents the negated entries [9].

Input noise, key-switching noise and modulus-switching error perturb the rotation index. Concrete uses half the cell width, $N/2^{p+1}$ rotation indices, as its failure threshold. For Gaussian displacement with standard deviation σ in rotation indices, the probability of exceeding this threshold is $\operatorname{erfc}\big(N/(2^{p+1}\sqrt{2}\,\sigma)\big)$. A displacement into a neighboring cell can still yield an allowed output. At fixed N, each additional bit of LUT input precision halves the cell width. Blind rotation performs n external products, each costing $O\big((k+1)^2\ell N \log N\big)$ operations with a decomposition of ℓ levels in base B_{PBS}. The bootstrapping key holds $n(k+1)^2\ell N$ coefficients [3]. Both PBS latency and BSK size consequently depend on N and ℓ.

2.2 Quantized Inference with Concrete ML

Concrete ML [40] turns a quantized network into an integer program, which the Concrete compiler [39] lowers to TFHE. The weights stay in plaintext, so a convolution or a dense layer computes each layer accumulator $x = \sum_j w_j y_j + \beta$ with ciphertext additions and plaintext multiplications only. The following activation, together with the requantization of its output to the integer grid of the next layer, forms one univariate function. Concrete represents this operation as a table lookup (TLU) and evaluates each encrypted scalar lookup with one PBS. Accumulators are much wider than the tables a PBS evaluates efficiently, so Concrete ML rounds each layer accumulator to its p most significant bits before the lookup [34]. With rounding step $\delta = 2^t$, the lookup uses the rounded index of x/δ. In the cheaper approximate mode, this index is either $\lfloor x/\delta \rceil$ or $\lfloor x/\delta \rceil + 1$, so a lookup may return either adjacent entry. Concrete ML disables overflow protection, so Concrete's approximate clipping [39] does not apply.

Concrete then selects every cryptographic parameter automatically. Its optimizer [3] minimizes an estimated cost over the LWE dimension, the GLWE parameters (k, N) and the decomposition parameters, at the estimated 128-bit security level for a target circuit failure probability of 2^{-40}. Within a parameter partition, PBS evaluations share parameters chosen for the most demanding operation. Multiple partitions allow different parameters and may require additional evaluation keys and key switches between partitions. We compare both the lowest-latency baseline configuration and the configuration with minimum evaluation-key size.

2.3 Threat Model

We follow the client-server setting of Concrete ML. The client encrypts its input under its secret key and sends the ciphertexts with the evaluation keys, the BSKs and KSKs, so their size sets both the client's upload and the server's storage per client. The server holds the trained model in plaintext, evaluates the network on the encrypted input, and returns encrypted outputs that only the client can decrypt. The server is semi-honest, and security rests on the hardness of LWE and GLWE [5, 25, 29] at the 128-bit security level estimated with the lattice estimator [1]. LUTune uses the same client–server interface and security target. Section 3.6 bounds circuit failure under Concrete's noise model, including incorrect decoding of the final outputs.

3 The LUTune Compiler

LUTune chooses a LUT design for each layer, emits candidate circuits, and lets Concrete select parameters for each partitioning strategy. It checks each candidate against the allowed outputs of the baseline and the target failure probability. It first considers candidates meeting both baseline minima for estimated computational cost and evaluation-key size. If none qualifies, it uses the cost and size of the original circuit, checked with the same bound, as limits. Among eligible candidates it minimizes estimated cost times evaluation-key bytes, breaking ties by estimated cost. Section 4 reports measured server latency separately.

3.1 Allowed Outputs and Input Domain

Consider a lookup with table T whose layer accumulator x is rounded with step δ. Under approximate rounding, Concrete shifts the accumulator by just under half of one integer unit of x, not of δ, before

dropping its low bits [39]. Even at $x = k\delta$, this places the boundary to index $k+1$ less than half a cell away, so Concrete's half-cell tolerance admits both k and $k+1$ (Section 2.2). The set of allowed output values is

$$A(x) = \{T(\lfloor x/\delta \rfloor), T(\lfloor x/\delta \rfloor + 1)\},$$

and $A(x) = \{T(x)\}$ for a lookup without approximate input rounding. We consider inputs on which the baseline circuit is defined, including the requirement that both permitted indices lie within the table. The client checks the legal range of the quantized input before encryption. Starting from this range, we propagate interval bounds through the linear layers, allow each lookup any value in $A(x)$, and intersect the bounds with the baseline encodings. Each LUT returns only its table values, so the ranges reset at every LUT instead of growing through the network. These intervals bound the reachable inputs. If each rewritten lookup returns an allowed value and shared uses see the same choice, those choices describe a permitted baseline execution. For each input in this domain, we bound the probability of a lookup outside its allowed set or incorrect final output decoding by $p_{\text{fail}} = 2^{-40}$.

3.2 LUT Folding

LUT folding multiplies the encrypted layer accumulator by 2^f before PBS. The resulting periodic index mapping reduces LUT input precision from p to $p_f = p - f$. The rotation domain then contains 2^{p_f} cells for the LUT and 2^{p_f} for its negated entries. Several inputs may reach the same cell c after folding, allowing up to half a rounding step of displacement. Cell c must represent an output permitted for every such input, so we require

$$C(c) = \bigcap_{x \text{ reads } c} A(x) \neq \emptyset$$

for every cell reached by a bounded input. Unread cells impose no restriction. For example, if two inputs reaching c allow $\{2,3\}$ and $\{3,4\}$, respectively, then c must return 3. The negacyclic encoding stores $2y - \kappa$ for output y, so the opposite cell returns $\kappa - y$. If the cell opposite c must return 4, that pair fixes $\kappa = 7$. Every pair in the table must admit the same κ. A compatible assignment of output values exists exactly when there is a κ such that

$$C(c) \cap (\kappa - C(c + 2^{p_f})) \neq \emptyset \quad \text{for every cell } c < 2^{p_f}.$$

Any pair whose two cells are both read restricts κ to a sum $a + b$ with $a \in C(c)$ and $b \in C(c + 2^{p_f})$, so the check is exact and takes time polynomial in the number of cells. LUTune also checks that the centered coefficients fit the output encoding. For each layer, it compares feasible designs by summing the estimated PBS costs at their respective input precisions, including the auxiliary lookups. A deeper fold can thus lose to a shallower one with fewer auxiliary lookups.

Let R denote the span of the varying region, measured in LUT input indices from the first change of value to the last, including intervening plateaus. A long increasing ramp can prevent folding. If two pairs of opposite cells admit disjoint sets of sums, no single value of κ satisfies both pairs. This limits the fold even when the constant intervals are long. In Table 1, every layer folded without a coarse and residual decomposition satisfies $2^{p_f} \geq R$. For example, CNV layer 2 has $R = 5$ and its LUT input precision falls from 7 to 4 bits.

3.3 Input-Offset LUTs

After folding, inputs from a constant interval and a varying interval can map to the same entry with incompatible allowed outputs. An auxiliary LUT evaluates a piecewise-constant offset on the encrypted layer accumulator. Each possible offset is a multiple of the public rounding step δ. Adding the encrypted offset before folding maps the constant interval to the coarse cell beside the varying region (Fig. 1c). It can occupy two coarse cells when noise selects a neighboring index. We check the folding condition for every permitted index displacement of the input-offset LUT and extend its endpoint values across the corresponding boundary intervals. This costs one additional PBS evaluation with a smaller polynomial size than the original PBS.

Table 1. LUT design by layer. R is the largest span of the varying region in the layer, and p and p_f are the input precisions in bits of the original and main LUT. Offsets: Section 3.3.

Network	Layer	p	R	p_f	Design
VGG6	1	7	12	5	fold + offset
	2	7	121	5	coarse/residual
	3	7	23	6	fold + offset
	4	7	87	5	coarse/residual + offset
	5	7	38	6	fold + offset
	6	7	27	6	fold + offset
	7	7	25	6	fold + offset
CNV	1	6	6	5	fold + offset
	2	7	5	4	fold + offset
	3	4	7	3	fold
	4	6	5	4	fold + offset
	5	6	17	5	fold + offset
	6	4	7	3	fold
	7	6	6	5	fold + offset
	8	6	16	5	fold + offset
	9	6	9	5	fold + offset
	10	6	42	6	original
	11	6	27	5	fold + offset

3.4 LUT Decomposition into Coarse and Residual Terms

When the activation varies over most input indices, a direct fold may fail. An input-offset LUT changes only the index used by the next lookup. In a coarse and residual decomposition, the first LUT also supplies part of the final value. It reads the high bits of the layer accumulator x and returns an encrypted coarse output G_j. A public integer m shifts the residual input by mG_j. With residual LUT H, the reconstructed output is

$$G_j + H(x - mG_j).$$

For a ramp with slope α measured in output units per unit of x, choosing m near $1/\alpha$ lets a change in the coarse output shift the residual in the opposite direction. Because m is an integer and the quantized ramp has uneven steps, the two changes need not cancel exactly. LUTune therefore checks that the sum belongs to $A(x)$ for every input in the domain and every permitted coarse index j. This decomposition applies to both VGG6 pooling layers (Table 1).

3.5 Signed-Digit Weight Decomposition

LUT folding reduces input precision. PBS cost also depends on the blind-rotation decomposition level count ℓ, which Concrete selects to meet downstream noise constraints. A dense layer computes the layer accumulator $z_k = \sum_i w_{ki} y_i + \beta_k$. Concrete's noise model [39] assigns each bootstrap an output variance σ^2_{out} and propagates the dot-product variance as

$$\sigma^2_{z_k} = \sigma^2_{\text{out}} \sum_i w^2_{ki}. \tag{1}$$

For dense layers with hundreds of inputs, the squared weight norm can reach 10^4 to 10^5. In the VGG6 precision-based partition, the classifier's maximum Euclidean norm over output-neuron weight vectors is 184. Concrete selects six decomposition levels for the BSK also used by three other layers.

LUTune writes each plaintext weight in signed radix-B_w digits as $w_{ki} = \sum_{j=0}^{d-1} c_{kij} B_w^j$, with $|c_{kij}| \le B_w/2$. For $B_w = 2$ it uses the non-adjacent form. Signed digits of small Euclidean weight are known to reduce the noise variance of scalar multiplication in FHE [19]. LUTune obtains the scaled copies $B_w^j y_i$ from the

activation's own PBS, so each carries fresh bootstrap noise. For each activation, d PBS evaluations apply scaled LUTs to the same input ciphertext. When they select the same LUT index, the ideal outputs are $B_w^j y_i$ for a common activation value y_i. The dense layer multiplies output j by c_{kij}. The resulting layer accumulator and its variance are

$$z_k = \sum_{i,j} c_{kij}\big(B_w^j y_i\big) + \beta_k, \quad \sigma_{z_k}^2 = \sigma_{\text{out}}^2 \sum_{i,j} c_{kij}^2. \tag{2}$$

For the VGG6 classifier, $B_w = 8$ gives two digit planes. The maximum Euclidean norm of the stacked digit coefficients for one output neuron is 52, against 184 under (1). Concrete then selects four decomposition levels for the shared BSK.

Approximate input rounding permits either of two adjacent indices. All terms for activation i must use the same permitted value y_i. LUTune repeats the same input ciphertext in one vectorized lookup operation. The copies have identical input encodings and use the same key-switching key and modulus-switching parameters, so all d PBS evaluations use the same rotation index in their respective LUTs. The digits then reconstruct the contribution $w_{ki} y_i$ for every output neuron k. Equation (2) follows Concrete's noise model [3] and the TFHE independence heuristic [8], which treat distinct PBS output errors as independent. Although the d copies share their input ciphertext, rotation index, and bootstrapping key, their scaled test polynomials initialize the accumulator with different bit patterns. Nonlinear gadget decomposition during blind rotation scrambles the resulting GGSW external products, which we empirically verified decorrelates the output errors and matches (2). All our failure bounds use this model.

Multi-output PBS can evaluate several functions of one ciphertext [6, 10]. LUTune uses d standard PBS evaluations, adding $d-1$ per activation. The compiler tries $B_w \in \{2, 4, 8\}$ for the final dense layer alone and for all eligible dense layers, and ranks the candidates as described in Section 3.

3.6 Bounding the Failure Probability

For each PBS evaluation b, we compute a margin r_b in rotation indices. For a main LUT this is the minimum distance to an index yielding a disallowed output, taken across the bounded inputs and every permitted selection of the auxiliary LUTs. For a residual LUT we test the reconstructed sum against $A(x)$. For an auxiliary LUT we bound displacement beyond the selections covered by this check. Concrete models the displacement at the selected parameters as Gaussian with standard deviation σ_b in rotation indices. Its two-sided tail at r_b is $\operatorname{erfc}\big(r_b/(\sqrt{2}\,\sigma_b)\big)$. Summing over all bootstraps and decoded outputs bounds the program's failure probability:

$$P_{\text{fail}} \le \sum_b \operatorname{erfc}\left(\frac{r_b}{\sqrt{2}\,\sigma_b}\right) + \sum_o P_o, \tag{3}$$

where P_o is the modeled probability of incorrect decoding at output o. The union bound allows dependent failure events. When weights are decomposed, we count each scaled LUT evaluation with the original index margin and propagate the modeled variance of (2) to later operations. For a residual LUT, σ_b includes the coarse output's noise variance scaled by m^2. Concrete uses half a cell width as its baseline threshold. LUTune uses the margin from the allowed outputs and keeps a rewrite only if (3) meets the target.

4 Experimental Evaluation

We implement the compiler passes of LUTune in Python on top of Concrete ML 1.9.0 and the Concrete 2.10 compiler it uses [39, 40]. LUTune reads the MLIR program that Concrete ML generates for a trained model, rewrites its lookups, and passes the result to Concrete's unmodified parameter optimizer and runtime. Compiling and checking all candidates takes 1–4 minutes per network and scales linearly with each LUT's input range. All experiments run on an AMD Ryzen Threadripper 7970X (32 cores, 64 threads) and 165 GB of memory under Ubuntu 24, and encrypted runs use 64 threads.

Table 2. Evaluation keys, latency and simulated accuracy at 128-bit security and a circuit failure target of 2^{-40}. TLUs counts the baseline's scalar lookups per inference. †Selected automatically by LUTune.

Network	Baseline TLUs	Configuration	Key size (GB)	Latency (s)	Sim. accuracy (%)
VGG6 (CIFAR-10)	41,088	Concrete ML, lowest latency	32.46	862.2	72.5
		Concrete ML, minimum key size	5.28	1,444.6	72.8
		LUTune, multiple partitions†	1.89	165.4	72.5
		LUTune, single partition	1.33	506.4	72.7
CNV (CIFAR-10)	158,346	Concrete ML, lowest latency	5.05	1,427.5	86.5
		Concrete ML, minimum key size	2.46	3,554.2	86.8
		LUTune, multiple partitions	2.85	335.0	86.4
		LUTune, single partition†	0.84	1,326.8	86.8
LeNet-5 (MNIST)	8,084	Concrete ML, lowest latency	7.93	124.3	98.7
		Concrete ML, minimum key size	3.75	213.8	98.7
		LUTune, multiple partitions	2.51	49.6	98.7
		LUTune, single partition†	1.09	91.1	98.7
Fashion-MNIST CNN	1,176	Concrete ML, lowest latency	0.70	2.67	83.7
		Concrete ML, minimum key size	0.48	3.13	83.6
		LUTune, multiple partitions†	0.59	1.66	83.8
		LUTune, single partition	0.48	3.25	83.8

4.1 Comparison with Concrete ML

We evaluate four quantized networks: VGG6, a VGG-style [33] CIFAR-10 [21] network with 6-bit weights and activations whose layer accumulators are rounded to 7 bits; CNV [35], the CIFAR-10 network of the Concrete ML examples with its public checkpoint and the published 6-bit rounding [40]; LeNet-5 [23] on MNIST, quantized and rounded like VGG6; and a small convolutional network on Fashion-MNIST [37]. Concrete ML and LUTune compile each exported model at the same 128-bit security level and circuit failure target of 2^{-40}. We evaluate the baseline under each parameter-partitioning strategy, then report the configurations with the lowest measured latency and the smallest evaluation-key size. Latency is the server time of one encrypted inference, excluding key generation, encryption and decryption. Key size is the total of all bootstrapping and key-switching keys in Concrete's compilation report, uncompressed. At the selected parameters, we simulated all 10,000 test images per network.

On VGG6 (Table 2), LUTune is 5.2× faster than the lowest-latency baseline with 17.2× smaller keys, at the same simulated accuracy as that baseline. With a single partition, LUTune needs only 1.33 GB, 4.0× less than the minimum-key-size baseline, and still runs 1.7× faster than the lowest-latency baseline. On CNV, LUTune with multiple partitions is 4.3× faster than the lowest-latency baseline and reduces evaluation-key size by a factor of 1.8. Its simulated accuracy is 86.4%, compared with 86.5% for that baseline. With a single partition, LUTune still improves on the lowest baseline latency by a factor of 1.08 and on the minimum baseline key size by a factor of 2.9. On LeNet-5, LUTune with a single partition is 1.36× faster than the lowest-latency baseline, and its keys are 7.3× and 3.4× smaller than those of the two baselines. With multiple partitions, it runs 2.5× faster than the lowest-latency baseline with 3.2× smaller keys. On Fashion-MNIST, LUTune is 1.6× faster than the lowest-latency baseline. Its evaluation keys occupy 0.59 GB, between the two baseline sizes. With a single partition, it matches the minimum-key-size baseline. Within each network, simulated accuracies differ by at most 0.4 points, comparable to the simulator's run-to-run spread.

4.2 VGG6 Ablation

Table 3 separates the rewrites. Without auxiliary LUTs, only the first layer folds, because the long constant intervals of the other activations claim every cell. Folding alone takes 784.1 s. Adding input offsets lets all five activation layers fold and reduces latency to 263.8 s. Coarse and residual LUTs instead handle the pooling

Table 3. VGG6 ablation, with parameters reselected for every row and Concrete ML at lowest latency. (N, ℓ): polynomial size and decomposition levels of the PBS feeding the classifier. Both: input offsets and coarse/residual LUTs; digits: $B_w = 8$.

Configuration	PBS count	Classifier (N, ℓ)	Key size (GB)	Latency (s)
Concrete ML	41,088	$(32768, 10)$	32.46	862.2
Folding	41,088	$(16384, 8)$	6.72	784.1
Folding + offsets	50,944	$(8192, 7)$	6.15	263.8
Folding + coarse/residual	48,768	$(16384, 8)$	9.82	695.1
Folding + both	60,160	$(8192, 6)$	2.44	219.0
Folding + both + digits	60,544	$(8192, 4)$	1.89	165.4

Table 4. VGG6 on CPUs. Ku et al. [22] encrypt 1,000 test images and report no failure bound; Concrete ML (lowest latency) and LUTune simulate all 10,000 test images.

Method	Retrained	Failure target	Latency (s)	Accuracy (%)
Ku et al., $B_g = 2^{27}$	Yes	–	476	75.0
Ku et al., $B_g = 2^{18}$	Yes	–	617	83.0
Concrete ML	No	2^{-40}	862	72.5
LUTune	No	2^{-40}	165	72.5

layers, giving 695.1 s. Enabling both gives 219.0 s. Latency falls although the PBS count rises by 46%: halving N more than halves the cost of a PBS, and fewer decomposition levels reduce it further.

The last row adds the signed-digit decomposition of the classifier weights. Both of its LUTune circuits use the same folded tables and precision-based partitioning. The second digit adds one PBS for each of the 384 classifier inputs. Three earlier layers share the classifier's bootstrapping key, so their PBS evaluations also use four decomposition levels instead of six. Across the network, the additional rewrite reduces measured latency by 24.5% and evaluation-key size by 22.5% relative to the LUT rewrites alone.

4.3 Compiler Rewriting versus Model Retraining

Ku et al. [22] lower the cost of TFHE inference by changing the model instead of the compiler, and our VGG6 follows their VGG6-24 architecture. They evaluate every activation with one bootstrap under a single fixed FHEW/TFHE parameter set, and fine-tune the network for 150 epochs with injected noise so that it tolerates the noise these parameters leave; a smaller gadget decomposition base B_g reduces noise and increases latency. Table 4 compares their CPU results, obtained with OpenFHE on a 32-core Threadripper PRO 5975WX (Zen 3) from a floating-point accuracy of 83.8%, with LUTune's on a 32-core Threadripper 7970X (Zen 4), which start from 76.4% without retraining. With an older CPU generation and a different library, the table gives context rather than a controlled comparison.

5 Related Work

Encrypted neural-network inference. TFHE [8] evaluates an arbitrary univariate function while it bootstraps a ciphertext, which suits quantized networks. FHE-DiNN [4] evaluated discretized networks with sign activations, and Chillotti et al. [9] showed that programmable bootstrapping supports deep networks with ReLU activations. Later work lowers the cost by changing the model. Ku et al. [22] co-design the architecture, quantization and activations with the negacyclic structure of the bootstrap and fine-tune the result, and DCT-CryptoNets [30] computes in the frequency domain to reduce the cost of nonlinear activations, training

each network with quantization-aware training. LUTune keeps the weights and quantizers fixed and rewrites the compiled lookups.

TFHE compilers. Concrete ML [40] compiles quantized PyTorch and scikit-learn models into TFHE programs through the Concrete compiler [39]. Linear layers become integer dot products with plaintext weights, and each activation, fused with its requantization, becomes one table lookup. Rounding accumulators before their lookups shortens the tables [34], and Concrete's optimizer selects every cryptographic parameter for a failure probability of the whole circuit [3]. An optional Concrete pass narrows lookups to value bounds measured on sample inputs, which gives no guarantee for other inputs. LUTune works inside this flow on the program Concrete ML emits. Type systems for FHE programs bound worst-case noise [14]. Our bound uses each baseline lookup's allowed outputs to determine how far its rotation index may move.

Evaluating LUTs with large input precision. Several approaches support functions whose input precision makes a single standard PBS expensive. Bootstrapping variants evaluate several functions in one bootstrap or drop the padding bit [6, 10], or raise the precision a single bootstrap reaches [26]. Others split LUT evaluation at larger input precision into standard bootstraps: tree-based and chained evaluation of digits [16], homomorphic floor and digit decomposition [24, 27], recursive decomposition into negacyclic LUTs of decreasing input precision [17], and decomposition over small fields [2]. Ripple approximates tables by wavelet compression [15]. Hwang et al. [17] start from a specified full-domain function. LUTune chooses an output from each allowed set so that inputs can share a cell while satisfying the negacyclic encoding. Hardware accelerators [18,28,36] reduce the cost of a PBS evaluation and are complementary to reducing its polynomial size. Bipartite and multipartite methods add tabulated corrections to a coarse output value [11, 31, 32]. LUTune's input-offset LUT shifts the encrypted input before folding so a constant interval can share a cell without colliding with the varying region. In LUTune, the encrypted coarse value also shifts the residual LUT's input, and every permitted coarse index is checked. For Boolean gates, Yu et al. merge truth-table entries with equal outputs and use negacyclic test vectors [38]; LUTune merges cells across sets of allowed integer outputs. CompressedLUT shares repeated sub-tables losslessly [20]. ReducedLUT uses inputs absent from training as don't-cares [7]. LUTune enforces the rounding rule on every defined baseline input.

6 Conclusion

LUTune optimizes LUT evaluation in quantized TFHE inference with frozen weights and quantizers. LUT folding, input offsets, and coarse and residual terms reduce the required LUT input precision. Signed-digit weight decomposition can also reduce the blind-rotation decomposition level count selected for a dense layer. On VGG6, LUTune reduces latency by a factor of 5.2 and evaluation-key size by a factor of 17.2 relative to the lowest-latency Concrete ML configuration.

Acknowledgment

The authors would like to acknowledge the support of the National Science Foundation (Award 2239334).

References

1. Martin R. Albrecht, Rachel Player, and Sam Scott. On the concrete hardness of learning with errors. *Journal of Mathematical Cryptology*, 9(3):169–203, 2015.
2. Sonia Belaïd, Nicolas Bon, and Matthieu Rivain. Decomposition of large look-up tables for fast homomorphic evaluation. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2026(3):247–278, 2026.
3. Loris Bergerat, Anas Boudi, Quentin Bourgerie, Ilaria Chillotti, Damien Ligier, Jean-Baptiste Orfila, and Samuel Tap. Parameter optimization and larger precision for (T)FHE. *Journal of Cryptology*, 36(3), 2023.
4. Florian Bourse, Michele Minelli, Matthias Minihold, and Pascal Paillier. Fast homomorphic evaluation of deep discretized neural networks. In *Advances in Cryptology – CRYPTO 2018*, LNCS, pages 483–512, 2018.
5. Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. (leveled) fully homomorphic encryption without bootstrapping. In *Innovations in Theoretical Computer Science (ITCS)*, pages 309–325, 2012.

6. Sergiu Carpov, Malika Izabachène, and Victor Mollimard. New techniques for multi-value input homomorphic evaluation and applications. In *Topics in Cryptology – CT-RSA 2019*, LNCS, pages 106–126, 2019.
7. Oliver Cassidy, Marta Andronic, Samuel Coward, and George A. Constantinides. ReducedLUT: Table decomposition with "don't care" conditions. In *ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA)*, pages 36–42, 2025.
8. Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. TFHE: Fast fully homomorphic encryption over the torus. *Journal of Cryptology*, 33(1):34–91, 2020.
9. Ilaria Chillotti, Marc Joye, and Pascal Paillier. Programmable bootstrapping enables efficient homomorphic inference of deep neural networks. In *Cyber Security Cryptography and Machine Learning (CSCML)*, LNCS, pages 1–19, 2021.
10. Ilaria Chillotti, Damien Ligier, Jean-Baptiste Orfila, and Samuel Tap. Improved programmable bootstrapping with larger precision and efficient arithmetic circuits for TFHE. In *Advances in Cryptology – ASIACRYPT 2021*, LNCS, pages 670–699, 2021.
11. Florent de Dinechin and Arnaud Tisserand. Multipartite table methods. *IEEE Transactions on Computers*, 54(3):319–330, 2005.
12. Léo Ducas and Daniele Micciancio. FHEW: Bootstrapping homomorphic encryption in less than a second. In *Advances in Cryptology – EUROCRYPT 2015*, LNCS, pages 617–640, 2015.
13. Craig Gentry. Fully homomorphic encryption using ideal lattices. In *ACM Symposium on Theory of Computing (STOC)*, pages 169–178, 2009.
14. Tarakaram Gollamudi, Anitha Gollamudi, and Joshua Gancher. ILA: Correctness via type checking for fully homomorphic encryption. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 1454–1468, 2025.
15. Charles Gouert, Mehmet Ugurbil, Dimitris Mouris, Miguel de Vega, and Nektarios G. Tsoutsos. Ripple: Accelerating programmable bootstraps for FHE with wavelet approximations. In *Information Security (ISC)*, LNCS, pages 273–293, 2024.
16. Antonio Guimarães, Edson Borin, and Diego F. Aranha. Revisiting the functional bootstrap in TFHE. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2021(2):229–253, 2021.
17. Intak Hwang, Shinwon Lee, Seonhong Min, and Yongsoo Song. Efficient full domain functional bootstrapping from recursive LUT decomposition. In *Selected Areas in Cryptography – SAC 2025*, LNCS, pages 679–699, 2026.
18. Lei Jiang, Qian Lou, and Nrushad Joshi. MATCHA: A fast and energy-efficient accelerator for fully homomorphic encryption over the torus. In *ACM/IEEE Design Automation Conference (DAC)*, pages 235–240, 2022.
19. Marc Joye. Balanced non-adjacent forms. In *Advances in Cryptology – ASIACRYPT 2021*, volume 13092 of *Lecture Notes in Computer Science*, pages 553–576, 2021.
20. Alireza Khataei and Kia Bazargan. CompressedLUT: An open source tool for lossless compression of lookup tables for function evaluation and beyond. In *ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA)*, pages 2–11, 2024.
21. Alex Krizhevsky. Learning multiple layers of features from tiny images. Technical report, University of Toronto, 2009.
22. Yu-Te Ku, Feng-Hao Liu, Chih-Fan Hsu, Ming-Ching Chang, Shih-Hao Hung, I-Ping Tu, and Wei-Chao Chen. Optimizing encrypted neural networks: Model design, quantization and fine-tuning using FHEW/TFHE. *Proceedings on Privacy Enhancing Technologies*, 2025(4):1075–1091, 2025.
23. Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.
24. Zeyu Liu, Daniele Micciancio, and Yuriy Polyakov. Large-precision homomorphic sign evaluation using FHEW/TFHE bootstrapping. In *Advances in Cryptology – ASIACRYPT 2022*, LNCS, pages 130–160, 2022.
25. Vadim Lyubashevsky, Chris Peikert, and Oded Regev. On ideal lattices and learning with errors over rings. In *Advances in Cryptology – EUROCRYPT 2010*, LNCS, pages 1–23, 2010.
26. Shihe Ma, Tairong Huang, Anyu Wang, Changtong Xu, Tao Wei, and Xiaoyun Wang. Meta-PBS: Compact high-precision programmable bootstrapping. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 2701–2715, 2026.
27. Shihe Ma, Tairong Huang, Anyu Wang, Qixian Zhou, and Xiaoyun Wang. Fast and accurate: Efficient full-domain functional bootstrap and digit decomposition for homomorphic computation. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2024(1):592–616, 2024.
28. Adiwena Putra, Prasetiyo, Yi Chen, John Kim, and Joo-Young Kim. Strix: An end-to-end streaming architecture with two-level ciphertext batching for fully homomorphic encryption with programmable bootstrapping. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1319–1331, 2023.

29. Oded Regev. On lattices, learning with errors, random linear codes, and cryptography. In *ACM Symposium on Theory of Computing (STOC)*, pages 84–93, 2005.
30. Arjun Roy and Kaushik Roy. DCT-CryptoNets: Scaling private inference in the frequency domain. In *International Conference on Learning Representations (ICLR)*, 2025.
31. D. Das Sarma and D. W. Matula. Faithful bipartite ROM reciprocal tables. In *12th IEEE Symposium on Computer Arithmetic (ARITH)*, pages 17–28, 1995.
32. M. J. Schulte and J. E. Stine. Symmetric bipartite tables for accurate function approximation. In *13th IEEE Symposium on Computer Arithmetic (ARITH)*, pages 175–183, 1997.
33. Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. In *International Conference on Learning Representations (ICLR)*, 2015.
34. Andrei Stoian, Jordan Frery, Roman Bredehoft, Luis Montero, Celia Kherfallah, and Benoit Chevallier-Mames. Deep neural networks for encrypted inference with TFHE. In *Cyber Security, Cryptology, and Machine Learning (CSCML)*, LNCS, pages 493–500, 2023.
35. Yaman Umuroglu, Nicholas J. Fraser, Giulio Gambardella, Michaela Blott, Philip Leong, Magnus Jahre, and Kees Vissers. FINN: A framework for fast, scalable binarized neural network inference. In *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*, pages 65–74, 2017.
36. Michiel Van Beirendonck, Jan-Pieter D'Anvers, Furkan Turan, and Ingrid Verbauwhede. FPT: A fixed-point accelerator for torus fully homomorphic encryption. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 741–755, 2023.
37. Han Xiao, Kashif Rasul, and Roland Vollgraf. Fashion-MNIST: a novel image dataset for benchmarking machine learning algorithms, 2017. arXiv:1708.07747.
38. Mingfei Yu, Sergiu Carpov, Alessandro Tempia Calvino, and Giovanni De Micheli. On the synthesis of high-performance homomorphic Boolean circuits. In *Workshop on Encrypted Computing & Applied Homomorphic Cryptography (WAHC)*, pages 51–63, 2024.
39. Zama. Concrete: TFHE compiler that converts Python programs into FHE equivalent. Version 2.10.
40. Zama. Concrete ML: a privacy-preserving machine learning library using fully homomorphic encryption for data scientists, 2022.