CKKS Acceleration for Next-Wave Homomorphic Encryption

Xuan Wang
University of California San Diego
La Jolla, CA, USA
xuw009@ucsd.edu

Sanjay Deshpande
Northwestern University
Evanston, IL, USA
sanjay.deshpande1@northwestern.edu

Jaehyung Kim
Stanford University
Stanford, CA, USA
jaehk@cs.stanford.edu

Jakub Szefer
Northwestern University
Evanston, IL, USA
jakub.szefer@northwestern.edu

Tajana Rosing
University of California San Diego
La Jolla, CA, USA
tajana@ucsd.edu

Patrick Longa
Microsoft Research
Redmond, WA, USA
plonga@microsoft.com

Abstract

Fully Homomorphic Encryption (FHE) is a powerful cryptographic tool that enables computation directly on encrypted data without first decrypting it. It allows privacy-preserving computing where sensitive data is processed in the cloud or in other untrusted environments. Emerging functional bootstrapping and coefficient-based matrix-multiplication techniques offer opportunities to accelerate nonlinear and linear computations, respectively. However, there is still a gap between their theoretical advantages and achievable hardware performance gains. We present GEAR, an algorithm–architecture co-design for efficient FHE acceleration supporting recent CKKS advancements while preserving full conventional CKKS functionality for existing workloads. On the algorithmic side, we propose FusedFBS, a fused functional bootstrapping algorithm that combines coefficient-based matrix multiplication with grafting-enabled functional bootstrapping to enable high-precision, hardware-friendly FHE computation. On the hardware side, we propose the GEAR architecture, with a Modular Systolic Array and a Mixed-Radix Constant-Geometry NTT Unit, to accelerate key kernels. These mechanisms balance efficient execution with flexibility for evolving FHE algorithms, making it a versatile FHE accelerator. We demonstrate FusedFBS through encrypted Deep Learning inference. Across representative workloads, GEAR achieves more than 3000× speedup over the CPU baseline. Compared with state-of-the-art CKKS accelerators, it delivers significant speedup under similar area and power budgets.

1 Introduction

Fully homomorphic encryption (FHE) [47] enables computation directly on encrypted data without decrypting it. A client can therefore offload sensitive workloads to an untrusted cloud while both its inputs and results remain cryptographically confidential. This makes FHE a strong foundation for privacy-preserving services, including deep neural network inference [28, 76, 77], transformer and large language model inference [85, 117], federated learning [87], encrypted database analytics [19, 81, 110, 119], and private information retrieval [11, 82]. Among FHE schemes, CKKS [29] is widely adopted for these workloads because it efficiently supports approximate arithmetic over real and complex numbers.

Despite its compelling promise, FHE remains prohibitively expensive. Computation on encrypted data can be four to five orders of magnitude slower than computation on unencrypted data [71, 72, 98, 99]. Prior work has addressed this overhead through customized CPU implementations [7, 23, 86], GPU libraries [31, 40, 61, 63], and domain-specific FHE accelerators [60, 66–68, 98, 99]. However, bootstrapping [20, 26, 78] remains a dominant performance bottleneck. This operation is responsible for refreshing ciphertexts by reducing the error accumulated during encrypted computation, enabling computations of unbounded depth [20, 26, 78, 99]. Figure 1(a) shows that bootstrapping accounts for over 60% of the total runtime for machine-learning inference on a CPU. Even with dedicated ASIC accelerators, it still accounts for around 59%–85% of the total runtime for ResNet-20 inference [42, 66].

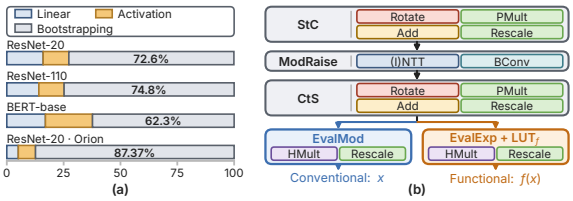


Figure 1: Bootstrapping cost and functionality. (a) Runtime breakdown of FHE applications; (b) Conventional and functional bootstrapping.

Functional bootstrapping (FBS) combines function evaluation with ciphertext refresh, an approach developed initially in the DM/CGGI [30, 70] and subsequently extended to SIMD evaluation using BGV/BFV [79, 80]. Recent works [14, 15, 37] also extend FBS to CKKS for boolean and integer data, known as discrete CKKS, which use functional bootstrapping to evaluate arbitrary lookup table (LUT)-based functional evaluation while retaining SIMD throughput. However, LUT-based evaluation scales poorly to high-precision real-valued nonlinear functions, since LUT cost grows exponentially with input precision. Fourier-extension FBS [18] bridges this gap by evaluating a Fourier-series approximation of the target function during bootstrapping, enabling high-precision numerical function evaluation. In CKKS, this integration can reduce the cost of separate approximation polynomial evaluation, whose successive homomorphic multiplications consume multiplicative depth and may require additional bootstrapping [9, 18]. Conventional CKKS, discrete CKKS, and Fourier-extension FBS differ in ring dimension, ciphertext representation, scaling requirements, and prime moduli. Supporting them efficiently in one accelerator requires **versatility** in the underlying arithmetic.

Alongside FBS-based nonlinear evaluation, FHE also requires efficient linear computation. Prior slot-based packing methods [38, 76, 77] can require hundreds of homomorphic rotations and multiplications with GBs of rotation keys [38] and up to 112 TB of encoded plaintext weights [61, 109] for deep learning inferences.

Recently, matrix-friendly schemes [12, 34, 49, 92] have been proposed to reduce both computational and storage costs. A trending approach is to utilize coefficient-based plaintext–ciphertext matrix multiplication (PCMM) [12], which replaces slot rotations and diagonally encoded weights with a direct modular matrix multiplication between plaintext weights and ciphertext coefficients. By avoiding rotations, PCMM eliminates the associated key switching and rotation keys, reducing both arithmetic and memory costs.

In this paper, we analyze the arithmetic and dataflow requirements of FBS and PCMM to guide the design of *a versatile CKKS accelerator that retains the efficiency of specialized hardware while adapting to different cryptographic parameters and application functionality.* Our investigation highlights several significant challenges in practical implementation. First, with small prime moduli, a fixed-word-length accelerator may not have enough distinct primes to meet FBS scaling requirements. Second, grafting [25] relaxes this constraint by decoupling scaling factors from individual RNS primes, but its non-NTT-friendly arithmetic requires additional transforms on short-word datapaths. Third, new matrix-multiplication dataflows involve GPU-like arithmetic operations that existing accelerators cannot efficiently support. Finally, most CKKS accelerators are designed with NTTUs optimized specifically for $\log N = 16$ and lack efficient support for other ring dimensions. Implementing these capabilities with existing computational approaches would incur substantial hardware costs.

Motivated by these considerations, we present GEAR, a versatile CKKS accelerator developed through algorithm–architecture co-design. We focus on two architectural requirements: 1) *Cryptographic parameter versatility*, which accommodates different ring dimensions, prime moduli, and scaling requirements; 2) *functional versatility*, which requires support for conventional RNS-CKKS, discrete CKKS for exact operations, high-precision functional bootstrapping, and linear algebra across FHE applications. GEAR provides these capabilities through configurable transforms and shared modular arithmetic while retaining conventional CKKS execution.

Overall, this paper makes the following key contributions:

- We characterize the modulus and scaling requirements of representative CKKS FBS methods and develop a 32-bit execution scheme for grafting's non-NTT-friendly arithmetic, reducing transform overhead through computation reuse and rescaling-aware scheduling.
- We develop FusedFBS, which integrates coefficient-based matrix multiplication with grafting-enabled FBS to reduce the computation and storage costs compared to the slot-based implementations. We demonstrate the impact of FusedFBS through encrypted DNN inference development, with an open-source implementation available at https://github.com/gearFusedFBS/GEAR_FusedFBS/.
- We systematically evaluate the compatibility of emerging CKKS algorithmic advances with existing accelerators in terms of functionality, performance, and hardware efficiency, and propose hardware-friendly integration techniques for coefficient-based matrix multiplication and accel erator-oriented, grafting-enabled functional bootstrapping.
- We design GEAR with a modular systolic array shared by base conversion and matrix multiplication, and a mixed-radix constant-geometry NTTU supporting multiple ring dimensions. To the best of our knowledge, GEAR is the first versatile CKKS acceleration framework designed to support emerging CKKS routines beyond conventional RNS-CKKS.
- Our experimental evaluation shows that GEAR outperforms previous work in both area and performance. For instance, GEAR achieves a 3,121× speedup over a CPU baseline and a 1.58× speedup over state-of-the-art CKKS accelerators on representative workloads, while reducing the area overhead by around 30%.

The rest of the paper is organized as follows. Section 2 reviews the necessary background. Section 3 analyzes FBS requirements and develops GEAR's short-word execution techniques for grafting. Section 4 presents FusedFBS, built on PCMM [12] and integrated with grafted FBS. Sections 5 and 6 describe the GEAR architecture and system support, followed by an evaluation in Section 7.

2 FHE Background

2.1 RNS-CKKS Scheme

Encoding and encryption. CKKS [29] is built on the ring learning-with-errors (RLWE) over the polynomial ring $\mathcal{R}_Q = \mathbb{Z}_Q[X]/(X^N + 1)$, where N is the ring dimension and Q is the ciphertext modulus. Encoding packs a vector of up to $N/2$ **cleartext** values, one per *slot*, into a polynomial $\mathbf{m} \in \mathcal{R}_Q$, which is called a **plaintext**. The values are scaled by a factor Δ. CKKS represents and computes on these values approximately, with Δ helping to determine the resulting precision. Encryption samples a uniform random polynomial $\mathbf{a}$ and sets $\mathbf{b} = -\mathbf{as} + \mathbf{m} + \mathbf{e}$, where $\mathbf{s}$ is the secret key and $\mathbf{e}$ is a small error polynomial, producing the **ciphertext** $\mathbf{ct} = (\mathbf{a}, \mathbf{b})$.

RNS and rescaling. Because Q spans hundreds to thousands of bits, it is chosen as a product of word-sized primes $q_0 \cdots q_L$, and each polynomial is stored in residue number system (RNS) form as $L + 1$ *limbs* of N words, with arithmetic performed independently on each limb [27, 46]. After a multiplication, *rescaling* divides by a prime close to Δ (scaling factor) to bring the scale from approximately Δ^2 back to Δ, dropping one limb. A ciphertext at level ℓ thus has $\ell + 1$ limbs remaining. In conventional RNS-CKKS, numerical precision is primarily governed by the scaling factor Δ, requiring rescaled limb primes q_i to be close to Δ to limit rescaling errors.

Operations and key switching. Ciphertext multiplication (HMult) produces a ciphertext under $\mathbf{s}^2$, while rotation (HRot) applies an automorphism that permutes the slots, leaving the result under a rotated secret key. Both must then apply *key switching* to return to $\mathbf{s}$, using a single relinearization key for HMult and a distinct key-switching key (ksk) for each rotation amount. Modern implementations adopt hybrid key switching [52], which splits the input into dnum decomposition digits and extends each digit to auxiliary primes with product P. Evaluation keys are therefore defined modulo PQ. Because polynomial multiplication in R_Q is a negacyclic convolution, ciphertexts are kept in NTT (evaluation) form, where it reduces to element-wise products. However, the basis extension in key switching operates on coefficients, so key switching proceeds through four consecutive kernels: an inverse NTT (iNTT) to coefficient form, base conversion (BConv) to the extended basis, an NTT back to evaluation form, and an element-wise modular inner product with the ksk. Specifically, for a polynomial x in coefficient form over a basis $C = \{q_0, \ldots, q_{k-1}\}$ with product Q_C, BConv computes

each limb of a new basis $\mathcal{B} = \{p_0, \ldots, p_{k'-1}\}$ as [60, 67, 68]

$$[x]_{p_i} = \Big(\sum_{j=0}^{k-1} \big[[x]_{q_j} \cdot \hat{q}_j^{-1} \big]_{q_j} \cdot \hat{q}_j \Big) \bmod p_i, \tag{1}$$

where $\hat{q}_j = Q_C/q_j$. The first step is a limb-wise modular scalar multiplication; the second multiplies a constant $k' \times k$ conversion matrix $[\hat{q}_j]_{p_i}$ by the $k \times N$ matrix of limbs. BConv is thus a modular matrix product of $k\,k'N$ multiply–accumulates and, unlike the other kernels, which process limbs independently, mixes all limbs [67].

Functional Bootstrapping (FBS). FBS extends conventional bootstrapping (**BTS**) by evaluating a target function while restoring ciphertext levels. FBS has recently been adopted in CKKS to combine nonlinear evaluation with ciphertext refresh across thousands of slots [9, 14, 18, 37, 64, 65, 116]. This fusion reduces computational cost and multiplicative-depth requirements, addressing the substantial bootstrapping overhead illustrated in Figure 1(a).

Figure 1(b) illustrates the pipeline shared by conventional bootstrapping and FBS, along with the additional capability that FBS provides. StC (Slots to Coefficients) first moves the slot values into polynomial coefficients. Once the ciphertext reaches the bottom modulus q_0, ModRaise reinterprets it modulo the full modulus Q, changing the underlying plaintext from m to $m + q_0 I$ for a small integer polynomial I. CtS (Coefficients to Slots) then moves these coefficients back into slots, where the subsequent function is evaluated slot-wise [26, 78]. Both are homomorphic linear transforms built from rotations and plaintext multiplications [20, 52]. Conventional bootstrapping then applies EvalMod, a polynomial approximation of modular reduction by q_0, which removes the overflow $q_0 I$ and recovers m. FBS replaces this stage with exponential evaluation (EvalExp)[1] followed by a lookup-table polynomial LUT_f, so that the refreshed ciphertext encrypts $f(x)$ rather than x [9, 15]:

$$\mathsf{ct}_{\text{out}} \leftarrow \mathsf{LUT}_f \circ \mathsf{EvalExp} \circ \mathsf{CtS} \circ \mathsf{ModRaise} \circ \mathsf{StC}(\mathsf{ct}). \tag{2}$$

3 FBS and Word Length for GEAR

To enable **cryptographic parameter versatility**, we first compare prevalent FBS techniques and co-design a 32-bit GEAR grafting schematic built on the CPU grafting [25] algorithm with fixed short-word-length accelerators to enable flexible FHE prime choices.

3.1 Analysis of FBS Methods and Routines

FBS methods differ in evaluation routines and precision requirements, which impose different software costs and hardware demands, although they share a bootstrapping backbone. In this work, we compare methods for general LUTs, high-precision function evaluation, and packed integer arithmetic. GFBS [9] uses Hermite interpolation [10, 105] to evaluate arbitrary LUTs over packed small inputs. For larger bit domains, its digit decomposition introduces dependent bootstraps on the critical path between several ciphertexts. A complementary line targets homomorphic arithmetic logic units (ALUs) for encrypted integer arithmetic and logical operations [21, 45, 64, 65]. For instance, FHIC [65] packs each integer's digits into one ciphertext to compute partial products in parallel. This reduces multiplication latency but accommodates fewer independent integers per ciphertext, while digit reduction still requires sequential bootstraps. FEFBS [18] generalizes this idea through Fourier extension, exploiting function smoothness for high-precision, batched evaluation of a broader class of functions within one bootstrap. Its measured precision and fusion of activation evaluation with refresh make it our representative for DNN inference and FBS.[2]

Table 1: No fixed word length fits every bootstrapping genre.

Scheme	$\log QP$[1]	Prime widths[3]	Word utilization (%)[1] 28-bit	32-bit	36-bit	Prec.[2]
CKKS	1703/1524	35, 40, 56, 60	75.1/74.6	80.6/82.1	71.7/77.0	26.7
[86]	*Scaled-sine EvalMod*					
FEFBS	1462/1394	20, 45, 50, 51, 54	96.7/93.9	84.6/82.2	75.2/73.1	34.7
[18]	*Fourier extension (single-shot)*					
GFBS	1665/1425	47	80.4/82.1	78.8/76.8	70.1/68.2	26.3
[9]	*Hermite LUT + Digit decomposition*					
FHIC	1537/1521	53, 57	94.6/90.5	82.8/84.9	73.6/75.4	20.86
[65]	*Packed digits + Lazy reduction*					

[1] Cells with two values give two parameter sets. The first are original parameters from the corresponding paper or implementation, followed by those generated using the heuristic parameter-selection algorithm of [20, Alg. 8].
[2] Precision is the per-slot error bits, $-\log_2 \epsilon$ against a cleartext reference, averaged over all slots.
[3] Representative retuned prime widths (bits), by stage. CKKS: 35 (StC), 40 (CKKSMult), 60 (EvalMod), 56 (CtS). FEFBS: 20 (post-bootstrap margin), 45 (StC), 54 (LUT and EvalExp polynomial), 50 (EvalExp squaring), 51 (CtS). GFBS: nominally 47 throughout the main chain. FHIC: 53 for intermediate stages and 57 for CtS.

3.2 Challenges of Prime Modulus Selection

FBS chooses its scaling factor (Δ) to fit the precision of the function, so naively its prime modulus changes from one to another (Section 2), while accelerators fix their word length at design time. Table 1 shows four bootstrapping representatives, whose prime widths range from 20 to 60 bits and differ across the stages of one bootstrapping [9, 14, 18, 64]. To accommodate the scaling requirements of different FBS, we first analyze the relationship of RNS primes, the scaling factor, and word length.

First, we observe that **small primes(scales) are needed by existing FBS algorithms but cannot be supported by short-word-length accelerators**. For example, GFBS uses 35- to 38-bit scales for 2- to 4-bit LUTs and 47- to 48-bit scales for 8- to 9-bit LUTs [9]. A 32-bit accelerator builds these scales from a product of primes $\Delta = \prod_{j=1}^{k} q_j$ with each $q_j < 2^{32}$, as in double-prime scaling (DS) [66] and composite scaling [1]. However, we find that since each prime must be NTT-friendly, $q \equiv 1 \pmod{2N}$. At $N = 2^{16}$, the smallest such primes are 786,433 $\approx 2^{19.6}$ and 1,179,649 $\approx 2^{20.2}$, so no product of two distinct NTT-friendly primes falls below $\approx 2^{39.8}$. 33 to 39 bits, including the 35- to 38-bit in GFBS, therefore fit neither one 32-bit prime nor a DS pair. 40 to 49 bits, including the 47-bit GFBS that Table 1 reports, cannot find each prime pairs. Every level of the modulus chain needs its own pair of distinct primes, and at most 14 disjoint pairs have a product within half a bit of 2^{47}.

[1] With the normalized input $x = m/q_0$, EvalExp computes $e^{2\pi i(x+I)} = e^{2\pi i x}$, so the integer overflow I vanishes by periodicity.

[2] For normalized values, 8-, 11-, and 24-bit CKKS precision roughly correspond to INT8/BF16, FP16, and FP32, respectively. Nonlinear evaluation, including polynomial approximations and LUTs, can accumulate and amplify errors across homomorphic operations, requiring substantially higher intermediate precision. For example, FP32-equivalent output accuracy (24 bits) may require over 30 bits of intermediate CKKS precision, i.e., $\log_2(\varepsilon^{-1}) > 30$ or $\varepsilon < 2^{-30}$, to accommodate this precision loss.

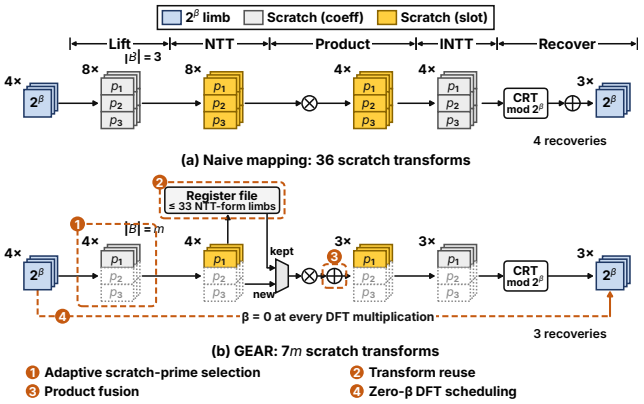


Figure 2: Comparison of GEAR sprout optimization and Direct sprout mapping [25].

Larger scaling factors alleviate these prime-selection constraints but increase the required modulus budget ($\log QP$) without necessarily improving precision. Maintaining 128-bit security may then require a larger ring dimension. Table 1 illustrates this trade-off by comparing bootstrapping performance under the original parameters and those selected using the heuristic of [20]. Meanwhile, DS incurs memory inefficiency due to partially utilized words. We quantify word utilization as the fraction of allocated bits occupied by $\log_2(QP)$ when a b-bit prime takes $\lceil b/w \rceil$ words of width w. As shown in Table 1, for 28-, 32-, and 36-bit words, no single workload fully utilizes all word length, leaving unused bits that still consume SRAM capacity and memory bandwidth. Although BitPacker [100] repacks residues at every rescale, its modulus basis changes with the level, limiting cross-level key reuse and increasing key storage [25].

To support every FBS algorithm on a fixed small word length, GEAR adopts grafting [25] to decouple scale from modulus width. Grafting combines the regular primes q_i with a *sprout*, a power-of-two factor of the modulus, which allows rescaling by amounts that do not correspond to removing a whole prime. A rescale can change both the q_i and the sprout, so Δ can differ from the prime modulus length. The regular primes therefore can stay the same as the hardware word length, which leads to better memory utilization without wasting bits and requires fewer limbs than DS [25, 66].

3.3 32-bit Grafting on GEAR

Grafting [25] is primarily designed for 64-bit CPUs, where its sprout either presents as a pure power of two, such as 2^{60}, or a composite of a smaller power of two and small odd NTT primes at most 2^{19}. For 32-bit words, it samples only one sprout, $2^{15} \cdot (2^{16}+1)$ [25, Example A.3], whose prime $2^{16}+1$ is NTT-friendly only for $N \le 2^{15}$, and $N = 2^{16}$ has no NTT-friendly prime below 20 bits (Section 3.2), which is consistent with our analysis in Section 3.2. GEAR therefore builds on the pure power-of-two sprout, 2^{31} held in a single limb.

As a result, Grafting [25] adds additional computational cost for the polynomial product of the sprout limbs. Figure 2(a) shows the *naive mapping* of the 2^β sprout limbs inside a grafted ciphertext multiplication. Two input ciphertexts of two polynomials each (4×) form four products (⊗) and leave three outputs (3×) with two cross products (⊕). Since the sprout limbs are non-NTT and stay in coefficient form, each of its products is computed on a set B of m *scratch primes*, auxiliary 31-bit NTT-friendly primes. The whole polynomial product requires 5 steps. First, a lift maps both operands onto every prime in B, the NTT units transform them, the modular multipliers then form the product, the (i)NTT units transform it back, and a recovery repacks the m results into the product on the sprout limb, where m is the smallest integer with $31m > 2\beta + \log_2 N + 1$. A *naive mapping* sizes B for the whole sprout, $m = 3$, and lifts, transforms, and recovers each product on its own, which requires 36 NTTs (24 + 12) on one sprout limb. We measured that it gives grafted FEFBS bootstrapping 40% more NTTs than DS, although it needs 11% fewer ordinary limbs.

However, a ciphertext carries only part of the sprout, 2^β with $0 \le \beta \le 31$, which rescaling raises or lowers. Our key observation is that β can be statically determined for the rescaling schedule, during compilation time, fixing the β of every operation. GEAR's compiler can therefore plan the scratch work of every operation, and even schedule for the optimal scenario that at $\beta = 0$, neither products nor key switches need scratch primes.

We therefore further propose four compilation-time **GEAR sprout optimizations** to reduce the overhead. Figure 2(b) shows the optimized execution flow. (1) *Adaptive scratch-prime selection* gives each product only the scratch primes that its own β requires. A product coefficient needs up to $2\beta + \log_2 N + 1$ bits, and a scratch prime holds just under 31. So at $N = 2^{16}$, one scratch prime suffices up to $\beta = 6$, two up to $\beta = 22$, and three above. In a standard 48-bit FEFBS chain, 61% of the products then run on one prime, and scratch NTTs drop by 48%. (2) *Transform reuse* keeps lifted and transformed limbs per homomorphic operation in the scratchpad to reduce NTT computations. Since (3) *Product fusion* exploits the linearity of the NTT: the two cross products of a multiplication, and the per-digit products of a key switch, are summed in NTT form on B, so one inverse transform per scratch prime and one recovery serve the whole sum when it still fits B. Most importantly, (4) *Zero-β DFT scheduling* postpones and rearranges rescaling within CtS and StC, so that $\beta = 0$ at every DFT multiplication, which is the dominant bottleneck for bootstrapping. By retaining sufficient intermediate modulus headroom and restoring the target modulus and scale at each DFT exit, the schedule preserves numerical precision without additional modulus consumption.

Operation Counts. Let n_b and n count the word primes of the input under DS and grafting. A ciphertext multiplication costs four element-wise multiplications and three additions on each limb, so grafting saves $4(n_b - n)$ multiplications and $3(n_b - n)$ additions. In tradeoff, when $\beta > 0$, the sprout limb takes $4m$ lifts, $4m$ NTT and $3m$ (i)NTT transforms, $4m$ multiplications, $m+2$ additions, and three recoveries (Fig. 2(b)). NTTs are the most expensive operations, which is why every sprout optimization targets them. Whereas a direct mapping spends 36 (i)NTTs on this limb, GEAR spends $7m$, a 1.7× reduction at $m = 3$ and a 5× reduction at $m = 1$, and spends none when $\beta = 0$.The sprout's cost depends on m, while grafting's saving grows with removed limbs since key switching and rescaling process fewer limbs, and every ciphertext and key stores fewer.

4 FusedFBS Algorithm

Building on coefficient-based matrix multiplication (PCMM) [12], we develop FusedFBS, which integrates PCMM into grafted FEFBS to fuse linear computation, nonlinear evaluation, and ciphertext

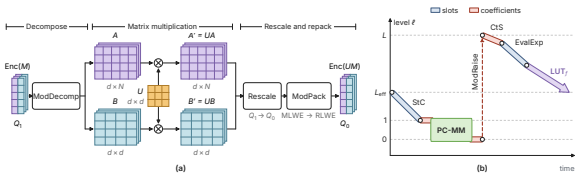


Figure 3: FusedFBS Algorithm. (a) PCMM workflow. (b) FusedFBS details with respect to ciphertext levels.

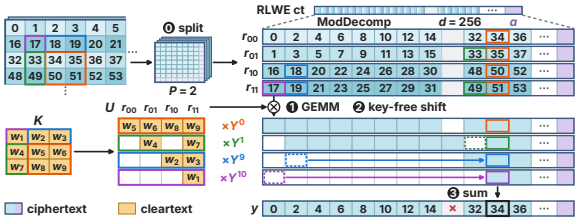


Figure 4: GEAR DNN packing

refresh within a common bootstrapping flow. We further develop an end-to-end encrypted DNN inference using convolution packing for PCMM-based linear layers and grafted FBS for non-linear activation functions.

4.1 Analysis of PCMM Dataflow

To reduce computation and data movement in linear layers due to encoded plaintext and homomorphic rotation, we adopt PCMM [12], which computes modular matrix products directly on ciphertext coefficients without homomorphic rotations. Figure 3(a) shows the workflow of PCMM. For a $d \times d$ tile and ring degree N satisfying $\sqrt{N} \le d \le N$, ModDecomp reorganizes coefficient-encoded inputs into row-oriented Module-LWE (MLWE) ciphertexts without key switching, yielding components $A \in \mathbb{Z}_{Q_1}^{d\times N}$ and $B \in \mathbb{Z}_{Q_1}^{d\times d}$. Given scaled cleartext weights U, each active RNS modulus q_j computes

$$A'_j = UA_j \bmod q_j, \qquad B'_j = UB_j \bmod q_j. \tag{3}$$

The results encrypt UM. Rescaling from Q_1 to Q_0 precedes ModPack, reducing the number of limbs processed by its key switches when repacking the results into ordinary ciphertexts [12].

Ring switching [48], through a key switching process, is utilized to further reduce the matrix multiplication cost, $C_{\text{MM}} = r(d^2N + d^3)$, by executing PCMM in a smaller ring. The inverse switch restores the bootstrapping ring after ModPack. The ring switch trades off against additional key-switching cost. For instance, switching from $N_{\text{BS}} = 2^{16}$ to $N = 2^{13}$ before ModDecomp reduces the computation required for UA and the storage of A and A' by 8× for a fixed tile but requires 2 additional key switchings.

4.2 FBS-Fused Matrix Multiplication: FusedFBS

However, PCMM requires coefficient-wise limbs, while default CKKS HMult and HRot operations all happen in slot-wise domains. We observe that the StC and CtS transforms overhead could offset PCMM savings. Thus, we develop **FusedFBS** that adopts PCMM placement within grafted FBS, as shown in Figure 3(b). Because our workflow already evaluates nonlinear functions inside bootstrapping, it naturally starts from the coefficient-wise domain needed to incorporate the preceding linear computation. Unlike BootNet [115] and functional S2C [112], which fuse linear operations into StC, FusedFBS executes ring switching and the PCMM dataflow after StC, minimizing both the matrix multiplication and the ring switch overhead. The result then returns to the bootstrapping ring before ModRaise. CtS then brings it back to slots, and EvalExp and LUT_f evaluate the subsequent nonlinear function while completing the refresh. To accommodate PCMM with grafting, we set the rescale schedule $\beta = 0$ at the PCMM levels, so ring switching and PCMM run unchanged on regular primes, which are also NTT-friendly in the smaller ring.

FusedFBS, however, requires **different ring dimensions and modular GEMM**, which demands **parameter and functional versatility** that prior CKKS accelerators do not provide. The NTTs in the key switches of ModPack have size N, all others have size N_{BS}, and N varies with ring selection, so GEAR adopts a Mixed-Radix NTT unit that serves every ring dimension (Section 5.4). Additionally, although prior BConv units [60, 66, 67] are implemented with Modular MAC units, they cannot efficiently support the PCMM matrix multiplication. Those BCONV units are sized for inner products of one digit's limbs with a small table of constants, and will need to split each d-term inner product into blocks, each reduced and written back to the scratchpad. GEAR thus builds a Modular Systolic Array (ModSys, Section 5.2) that streams both operands, reduces each inner product once, and also executes BConv.

4.3 FHE Deep Learning with FusedFBS

We use FusedFBS to develop a new encrypted DNN inference pipeline by fusing linear layers and their subsequent activations into a shared bootstrapping flow. This showcases the benefits of exploiting compatibility among emerging FHE advancements.

Linear layers and activations. Slot-based pipelines such as Orion combine rotations and encoded weight diagonals for linear layers with polynomial approximations for activations [38]. In FusedFBS, the linear transformation executes through coefficient-domain PCMM [12], and the following activation is evaluated within the LUT_f stage of FEFBS [18]. The refreshed ciphertext therefore already contains the activated output.

Convolutional packing. Figure 4 illustrates our FusedFBS-based convolution layer layout and computation with a 3×3 convolution. P denotes the product of the remaining spatial strides. We split each input channel into P^2 spatial phases by grouping pixels according to their coordinates modulo P. Each phase occupies a row of d coefficient positions in raster order, with zero guard positions separating raster rows and padding the unused tail. At the PCMM stage, ModDecomp reindexes the ciphertext coefficients into the corresponding MLWE rows. The cleartext matrix U groups the scaled kernel weights by the spatial shift required to align their contributions. A modular GEMM with the ciphertext coefficient matrices produces partial results for each output channel, output phase, and shift group. Instead of relying on HRot as prior slot-based inference, our FusedFBS-based implementation achieves shifting by multiplying a monomial Y^δ, where Y is the subring variable. Here, each ciphertext encrypts its values as the coefficients of a polynomial in the subring variable Y, so multiplying a partial result by the monomial Y^δ before summation shifts it by δ positions under the same key. In the illustrated $P = 2$, $d = 16$ example, the nine kernel taps form four groups with offsets $\delta \in \{0, 1, 4, 5\}$. Since GEAR runs one functional bootstrap per activation ciphertext, its

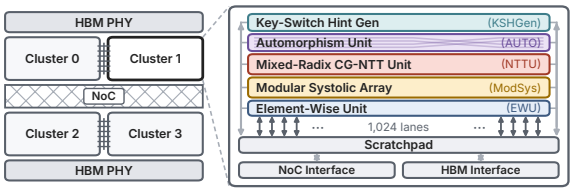


Figure 5: GEAR architecture overview.

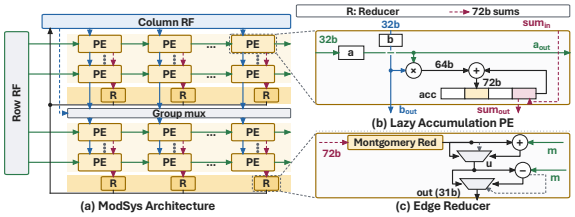


Figure 6: Modular Systolic Microarchitecture and Dataflow.

bootstrap count falls as each ciphertext holds more activations. We dynamically decide between full packing and half packing bootstrapping since GEAR runs one functional bootstrap per activation ciphertext. With half packing, activations occupy only the first half of a ciphertext's coefficients and reduce latency but could require more ciphertexts during the intermediate layers.

5 GEAR Architecture

From the algorithm perspective, we support *Cryptographic parameter versatility* and *Functional Versatility* with **GEAR 32-bit grafted FBS** supporting efficient sprout limb computations, FusedFBS with PCMM, and FusedFBS-enabled DNN inference, which requires flexibility in the ring dimensions, modulus representations, and dataflows. Therefore, from an architecture perspective, we design the GEAR CKKS accelerator based on algorithm-architecture co-design. GEAR provides the required flexibility through a Mixed-Radix NTTU (Section 5.4) and a unified modular systolic array shared by base conversion and matrix operations (Section 5.2).

5.1 GEAR Architecture Overview

Figure 5 shows the four-cluster architecture of GEAR, with 256 lanes per cluster and a NoC for inter-cluster communication. The global data distribution policy between clusters adopts approaches used in SHARP [66], ARK [67], and HAWK [72] to enable full (i)NTT in each individual cluster while efficiently transferring limbs required for base conversion. Specifically, limbs are stored limb-wise in each cluster for (I)NTT and other operations and distributed coefficient-wise for BConv and PCMM [67]. The sprout limbs are also stored like any word limb, in coefficient order (Section 3.3). Each GEAR cluster integrates four main components and has a 32-bit word length, as determined in Section 3.3. The mixed-radix NTT unit (**NTTU**) accelerates (i)NTT, supporting ring dimensions from $\log N = 13$ to $\log N = 16$. A unified modular systolic array (**ModSys**) is specifically designed to support both base conversion and modular matrix operations, enabling efficient execution across diverse FHE workloads while reducing hardware overhead. As in prior accelerators [60, 66, 99], an automorphism unit (**AUTO**) and an element-wise unit (**EWU**), with 4 modular multipliers and 2 modular adders, execute homomorphic rotation and standalone modular multiplications and additions. GEAR also adopts the NTT-friendly Montgomery modular multiplier from [83, 84] to improve area and power efficiency. Specifically, an iterative reduction is implemented, requiring $L = \lceil K/W \rceil$ recursive steps of reduction, where $W = \log_2(2N)$, such that we can accommodate different $\log_2(N)$. However, to support sprout limb operations, we adopts 3 Montgomery multipliers and 1 Barrett multiplier.

5.2 Unified Modular Systolic Array

The Unified Modular Systolic Array (**ModSys**) executes BConv and modular GEMM on shared hardware, removing the need for a dedicated unit for either and keeping the array busy across HE operations. Both are instances of one modular matrix product,

$$c_{j,n} = \left(\sum_{\ell=0}^{K-1} a_{j,\ell}\, b_{\ell,n} \right) \bmod m_j, \tag{4}$$

where K is the dot-product length, which is equivalent to the number of limbs in a single digit, $\alpha = \lceil (\ell+1)/dnum \rceil$, in BConv and the inner matrix dimension in GEMM. Each output $c_{j,n}$ is one K-term inner product reduced modulo its row modulus m_j.

As shown in Figure 6(a), each **ModSys** comprises 24×128 systolic PEs organized into two 12-row reduction groups, each with 128 **Edge Reducer**s. Row operands a propagate horizontally and column operands b propagate vertically, while each PE accumulates its output as partial sums locally over K cycles. Completed sums shift down each group's columns to the **Edge Reducer**s, each shared by 12 PEs, for modular reduction and writeback. The group mux selects whether the lower group receives column inputs forwarded from the upper group or an independent column input from the Column RF. **ModSys** can therefore be split into two 12×128 arrays, which allows the repack of computations with smaller K to reduce functional unit idleness and improve parallelism.

The Row RF and Column RF provide 18 MiB of storage across the four clusters. For each cluster, 1.75 MiB is allocated for the input register files and 2.75 MiB for output register files. The Row RF holds row operands next to the array, so operand loading overlaps computation. The Column RF holds 256 operand banks and 256 result banks per cluster, one per lane, with one read and one write per bank per cycle. Therefore, Column RF supplies two 128-word column inputs and accepts 256 result words per cycle. Thus, it buffers newly produced tiles while streaming preceding tiles to the NTTU through vector chaining [97, 99].

Fig. 6(b) shows the **lazy-accumulation PE**, which defers modular reduction until the inner product is complete. Each PE contains a signed 32×32 integer multiplier and four 72-bit accumulator slots. The multiplier produces a 64-bit product, which is sign-extended and added to the selected accumulator slots. The 72-bit accumulator width accommodates lazy accumulation while avoiding overflow. The accumulator slots also serve directly as output registers: one slot forwards completed sums down the column to the **Edge Reducer** while another accumulates the next inner product. Accumulation and reduction thus overlap rather than run sequentially, enabling a fully pipelined accumulation–reduction flow with minimal head and tail overhead.

The **Edge Reducer** (Fig. 6(c)) performs one modular reduction per completed inner product, allowing 12 PEs to share a reducer.

It uses Montgomery reduction to limit the width of the reduction required. Since $\mathrm{REDC}(S) = SR^{-1} \bmod m$, BConv constants and encoded PCMM weights are pre-scaled by $R \bmod m$ at compile time, yielding outputs in the ordinary residue representation without additional runtime multiplication. A sign correction (add m if negative) followed by a conditional subtraction maps the reducer output from $(-m, 2m)$ to $[0, m)$ as the final **ModSys** output. Modular arithmetic is thus confined to the 256 **Edge Reducer**s, leaving the 3,072 PEs as an ordinary integer GEMM systolic array like those in ML accelerators, only with a wider datapath. Kernels that recent algorithms already reduce to matrix products evaluated with BLAS and other similar routines on CPUs and GPUs [12, 92] therefore map onto **ModSys** well. Thus, we support **versatile scheme functionality**.

5.3 BConv and GEMM Dataflow

Because both BConv and GEMM consume coefficient-domain limbs, they also share the inter-cluster distribution of (i)NTT results from the NTTUs such that the functional unit sharing is efficient. We map both onto **ModSys** with an output-stationary (OS) dataflow.

For **BConv** (Eq. 1), rows correspond to the k' destination limbs p_i, columns to coefficients n, and j spans the $K = \alpha$ source limbs, with $a_{i,j} = [\hat{q}_j]_{p_i}$ drawn from the conversion table in the Row RF, $b_{j,n}$ the n-th coefficient of the scaled source limb $\left[[x]_{q_j} \cdot \hat{q}_j^{-1}\right]_{q_j}$, and $m_i = p_i$. The four clusters partition the coefficient indices, with each cluster receiving all source limbs for its assigned coefficients. For **PCMM GEMM**, rows and columns correspond to those of the output matrix, with $a_{i,j} = A[i][j]$ a cleartext weight, $b_{j,n} = B[j][n]$ a ciphertext coefficient, a common modulus for the job, and K ranging from 64 to 4,608. Larger outputs are tiled across the array's row and column dimensions.

Both BConv and PCMM GEMM will not overflow by deferring reduction, which requires the sum to fit the 72-bit signed accumulator. For the supported BConv length $K \le 15$ and nonnegative operands below 2^{31}, the sum is less than $15 \cdot 2^{62} < 2^{66}$. For PCMM, with signed weights bounded in magnitude by $A_{\max}$ and ciphertext residues by $B_{\max} < 2^{31}$, $KA_{\max}B_{\max} < 2^{71}$ is a sufficient condition.

Conventional input- and weight-stationary (IS and WS) mappings can underutilize the array for BConv's short K unless additional partitioning is provided. For GEMM, IS and WS offer operand reuse but require transferring wide partial sums during accumulation, while OS execution keeps the 72-bit sums local.

5.4 Mixed Radix NTTU

Recent FHE accelerators such as SHARP [66], FAST [41], and Falcon [71] build their NTT units on the ten-step method, which maps the N coefficients onto a $\sqrt{N} \times \sqrt[4]{N} \times \sqrt[4]{N}$ matrix and processes it in lane groups. Each lane group combines butterfly stages with register-based transpose units and dedicated twisting multipliers, and the twiddle factors are constants configured when the computation is initialized. This organization is efficient when every polynomial has the same ring dimension and modulus, but it does not match our workload. Our parameter sets use ring dimensions from 2^{13} to 2^{16}, consecutive polynomials belong to different RNS limbs, and thus different moduli, and NTTs are interleaved with iNTTs. A unit whose matrix shape, transpose wiring, and twiddle constants are fixed to one configuration must idle on other ring dimensions or stall for reconfiguration. Its transpose registers and twisting multipliers also add area and switching activity to every transform. We therefore design the NTTU around three goals: a single datapath that serves every ring dimension at full width, stall-free per-polynomial reconfiguration of size, direction, and modulus, and near-continuous butterfly utilization.

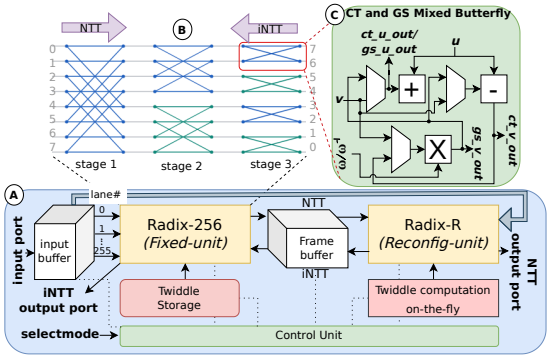


Figure 7: GEAR NTTU. (A) Top-level architecture. *Fixed-unit* is a fixed radix-256 unit, and *Reconfig-unit* is a radix-R unit which can change its entire configuration on-the-fly to $R = 256/128/64/32$. (B) Constant-geometry butterfly network, shown for 8 lanes and 3 stages (each unit has 256 lanes and 8 stages). NTTs traverse the stages from left to right with CT butterflies, and iNTTs from right to left with GS butterflies. Butterflies of the same color share one twiddle factor. (C) CT and GS mixed butterfly.

Fig. 7(A) shows the resulting fully pipelined constant-geometry architecture. To support every ring dimension $N = 2^n$ with $n \in \{13, \ldots, 16\}$, the NTT unit (NTTU) maps the N coefficients onto a two-dimensional $256 \times R$ matrix with $R = N/256 = 2^{n-8}$ and evaluates the transform with the four-step method in two phases: a column-wise 256-point NTT on each of the R columns, followed by row-wise R-point NTTs. Both phases use a radix-256 unit composed of eight fused layers of 128 radix-2 butterflies (Fig. 7(B)). By bypassing its first $s = 16 - n$ layers, the unit operates as 2^s independent radix-R units on the same 256 lanes. The datapath therefore processes 256 coefficients per cycle for every supported N.

To keep the hardware busy, the NTTU dedicates one unit to each phase instead of looping one unit back over both phases. *Fixed-unit* executes the 256-point phase and *Reconfig-unit* the R-point phase, so every polynomial traverses each unit exactly once, and both units can work on different polynomials at the same time. Between the phases, the transpose is absorbed into the addressing of banked frame buffers instead of a dedicated transpose network. Forward transforms flow from *Fixed-unit* to *Reconfig-unit*, whereas inverse transforms flow from *Reconfig-unit* to *Fixed-unit*. Each unit starts its next polynomial immediately after the last beat of the previous one whenever both share the same configuration.

The NTTU requires no separate twisting stage. The twisting factors between the two phases of the four-step method are merged into the twiddle factors of *Reconfig-unit*, and the pre- and post-twist of the negacyclic transform are absorbed by the choice of butterflies: we use Cooley–Tukey (CT) butterflies for the NTT and Gentleman–Sande (GS) butterflies for the iNTT. As shown in Fig. 7(C), each butterfly has a mode bit to switch between CT and GS. As shown in Fig. 7, in GS mode the layers are traversed in reverse order, which exactly undoes the corresponding CT layer. The factors of the inverse twist are thereby absorbed into the inverse twiddle factors, and the scaling by N^{-1} is distributed as one halving per layer,

computed as a shift and a conditional addition of q. Because both modes share the same modular multiplier, adder, and subtraction, the inverse transform requires no additional multipliers.

The twiddle factors follow a hybrid scheme that balances memory against computation: the twiddle factors of *Fixed-unit* are stored, whereas those of *Reconfig-unit* are computed on the fly. *Fixed-unit* applies the same twiddle row to every beat of a polynomial and to every ring dimension, so storing this row costs little memory, while computing it would only reproduce the same values. *Reconfig-unit*, in contrast, requires a different twiddle row in every beat, and storing all of them would take more than an order of magnitude more memory than the rest of the twiddle tables combined. Instead, we factor each twiddle factor of *Reconfig-unit* into a factor that depends only on the position of the butterfly within its radix-R sub-transform and a factor that depends only on the beat. Both factors are kept in small tables, and a row of modular multipliers forms their products for each beat, timed such that each layer's twiddle factors arrive exactly when the beat reaches that layer.

As shown in Fig. 7, *Reconfig-unit* also changes its entire configuration on-the-fly. The transform size, direction, and modulus arrive with the first beat of each polynomial. *Reconfig-unit* then selects the layers to bypass, switches its butterflies between CT and GS mode, fetches the factor row for that size, direction, and modulus, and starts generating the matching twiddle factors.

As described in Section 5.1, GEAR adopts the 32-bit NTT-friendly Montgomery modular multiplier [83, 84], which performs reduction in $L = \lceil K/W \rceil$ steps with $W = \log_2(2N)$. Each reduction step occupies its own pipeline stage, so the multiplier is fully pipelined and accepts a new operand pair every cycle. In total, the NTTU contains 2,303 modular multipliers: 1,024 in each butterfly unit and 255 in the twiddle generator.

5.5 Memory Organization and Sprouts Datapath

GEAR employs a heavily banked memory architecture with 256 MB of on-chip SRAM scratchpad, which connects to HBM for off-chip data loading. The banks serve one read and one write per cycle, and are interleaved so that a cluster's RF exchanges 256 words per cycle with the Mixed-Radix CG-NTTU in each direction while supplying the two 128-word column streams of ModSys and feeding the other functional units, which leads to a total of 32 TB/s on-chip SRAM bandwidth. Because grafting fills every limb with full 32-bit words, each bank is exactly one 32-bit word wide and does not require sub-bank splicing in SHARP [66].

Since FHE programs are static, the compiler allocates the registers and leverages greedy scheduling with Belady's algorithm [17] such that GEAR follows sequential data access and eliminates band conflicts and associated control and addressing. The automorphism access that does not follow sequential access follows a similar shuffle pattern as [66]. Also, we adopt memory optimizations including on-chip generation of each key's uniform half [99] and on-the-fly limb extension [67] to reduce off-chip key and encoded plaintext traffic.

6 Compilation and System Support

Our software stack is built on Lattigo [86] and supports conventional CKKS, FEFBS, grafting, and FusedFBS (Secs. 3 and 4). GEAR compiler is embedded within Lattigo's kernels and records each operation's scope, modulus levels, kernel, and input and output buffers.

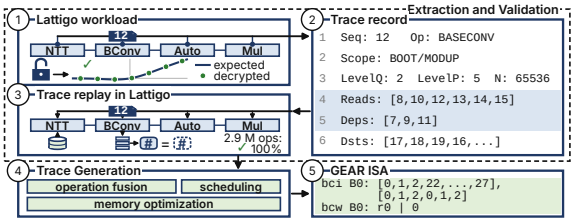


Figure 8: GEAR Compilation Flow.

Figure 8 summarizes the compilation flow for GEAR's instruction generation. A real Lattigo workload (①) is fed for instruction extraction and validation, and it generates the trace records (②). The exact read and write values are also extracted to construct data dependency graphs across consecutive operations. The generated traces are validated against the original application results through replay (③), using operation and scope names to call Lattigo functions, and using the extracted FHE values based on the dependency graph. The traced execution checks that its decrypted outputs are correct. Therefore, we ensure that the exact computations are captured for the original Lattigo workload without missing operations.

The trace generator then maps these operations to the limb-level vector ISA used by GEAR. It fuses operations through a compilation pass, manages the memory using greedy prefetch based on Belady [17, 98], and supports our memory optimizations and reordering [99](④). The compiled traces are then issued as GEAR ISA (⑤) and executed when their dependencies are satisfied. Figure 8 demonstrates a sample trace (②) of a BConv in the ModUp stage. Its 6 input limb buffers were produced by records 7, 9, and 11.

7 Evaluation

7.1 Implementation and Experimental Setup

We implemented GEAR's major hardware components in RTL and synthesized them using the ASAP7 7nm predictive PDK [22, 33, 104]. We modeled the SRAM components and long-distance interconnects using a customized version of FinCACTI [101], calibrated against published 7nm logic and SRAM technology specifications [22, 55, 104, 111]. Processing elements and interconnect networks operate at 1 GHz with fully pipelined execution, while single-ported SRAMs employ 2 GHz double-pumping [44, 66] to support 1R1W per cycle.

We implemented GEAR's FusedFBS and 32-bit grafting in the Lattigo [86] v6 CKKS library, using 128-bit-secure parameters [8]. [3] FusedFBS is implemented in Go, supporting FEFBS, customized grafting, and Fused PCMM, as described in Section 4. For performance evaluation, we built a cycle-accurate in-house simulator of GEAR based on the SST simulator [95], which receives FHE application traces and data-dependency graphs generated from Lattigo programs, as described in Section 6. The hardware architecture is modeled using timing from the synthesized RTL of GEAR.

We evaluate representative FHE workloads using our in-house simulator. We first compare GEAR with a CPU baseline, a 64-core, 128-thread, 5.4GHz AMD Ryzen Threadripper 9985WX with 755 GB

[3] For the sparse-secret encapsulation technique in [20], we use a secret polynomial with Hamming weight $h = 192$ to maintain 128-bit security at a large ciphertext modulus, while temporarily switching to a sparse secret with $\tilde{h} = 32$ during modulus raising. The smaller Hamming weight reduces the modular reduction range, improving bootstrapping precision and efficiency. The ciphertext is subsequently switched back to the original secret for the remaining bootstrapping operations.

Table 2: Area and average power of GEAR.

Component	Area (mm^2)	Power (W)
4×NTTU	14.5	19.6
4×MODSYS	7.5	18.2
4×KSHGen	0.3	0.3
4×EWU	3.2	6.1
4×AUTO	1.2	5.5
Scratchpad Memory	59.3	31.5
NoC	7.2	5.9
HBM	29.6	8.9
Total	**122.8**	**95.7**

of RAM (Table 3). CPU latencies are reported excluding key generation and setup. We also compare GEAR with prior accelerators, particularly Cinnamon [60]. We select Cinnamon because its publicly available simulator[4] enables reproducible and flexible workload-level comparisons. Specifically, we extend our compiler with a Cinnamon backend to map and evaluate FBS-specific workloads, which have not been evaluated by prior FHE accelerators. This approach enables a consistent comparison under identical workload configurations rather than relying on previously reported results using different FHE algorithms and parameter settings. We reconfigured the Cinnamon simulator to 32-bit word length and model our baseline **Cinnamon** with a 256MB register file, 4 clusters, 1 NTT unit, 1 transpose unit, 1 BCU buffer, 2 modular multiplications units, 2 modular addition units, and 1TB/s HBM bandwidth, matching the throughput of the state-of-the-art CKKS accelerator SHARP [66]. FHE instruction traces depend on the FHE implementation, instruction generation, and memory scheduling as discussed in Section 6, whose details are not fully specified in prior work [66, 71, 72]. We observe substantially more memory pressure in traces generated by both our framework and Cinnamon's built-in compiler integrated with SEAL [23] than reported in SHARP [66]. Cinnamon [60] accommodates this demand with 2 TB/s of HBM bandwidth per unit chip, whereas several prior accelerators combine larger on-chip memories with 1 TB/s of HBM bandwidth. We adopt a 256 MB scratchpad in GEAR and baseline **Cinnamon** to reduce memory pressure rather than increase HBM bandwidth. This choice follows the precedent of prior accelerators and limits reliance on HBM's growing demand driven by AI workloads. The workloads we used are as follows:

CKKS BTS: A fully packed bootstrap of a ciphertext with 2^{15} slots, using Lattigo's implementation [86] of the algorithm in [20] with $L_{eff} = 8$.

FEFBS [18]: A functional bootstrap using FEFBS with $\log PQ \leq 1533$. We also use the FEFBS routine as the building block for all our DNN inference workloads.

IntMult: One 16-bit integer multiplication from FHIC [65], which builds on FHIC functional bootstrapping [15, 64]. Each integer is represented as four 4-bit digits, and one multiplication requires four bootstraps.

ResNet20: Inference of one encrypted CIFAR-10 [74] image on ResNet-20 [54] using our proposed FusedFBS (Section 4).

ResNet20 (Orion): ResNet-20 [54] CIFAR-10 [74] inference compiled by Orion[5] [38]. Orion is an automatic compiler that minimizes the number of required bootstrappings and HRot, which evaluates convolutions using the diagonal method, approximates each activation with a polynomial, and inserts conventional CKKS bootstraps.

[4] https://github.com/CMU-CAOS/Cinnamon
[5] https://github.com/baahl-nyu/orion

Table 3: Performance of GEAR, Cinnamon (Cinna) and CPU on full FHE benchmarks.

	Execution time (ms)			GEAR Speedup	
Benchmark	**CPU (s)**	**Cinna**	**GEAR**	**CPU**	**Cinna**
CKKS BTS	11.2	3.8	2.2	5,091×	1.73×
FEFBS	8.5	3.7	2.3	3,696×	1.61×
IntMult	37.4	28.9	19.5	1,918×	1.48×
ResNet20	157.6	84.9	51.6	3,054×	1.65×
ResNet20 (Orion)	357.5	200.0	133.0	2,688×	1.50×
gmean speedup		**1,962×**		**3,121×**	**1.58×**

7.2 Performance and Efficiency

Table 3 compares the performance of GEAR, CPU, and baseline Cinnamon. It shows the execution times and GEAR's speedups over the CPU and Cinnamon. Similar to prior accelerators, GEAR achieves 3,121× geometric mean (gmean) speedup over the CPU implementations, ranging from 1, 918× to 5, 091×, and outperforms Cinnamon across all workloads by 1.58× in gmean and up to 1.73×, including both conventional CKKS BTS and FEFBS, showing that although GEAR targets functional bootstrapping and advanced algorithms, it also efficiently accelerates conventional CKKS.

Table 2 summarizes GEAR's component-level area and power. GEAR occupies 122.8 mm^2 and consumes 95.7 W on average across evaluated workloads, with the proposed NTTU contributing to both area and power savings. In comparison, prior 7 nm FHE accelerators SHARP [66] and Falcon [72] report areas of 178.8 mm^2 and 180.3 mm^2 and power of 98 W and 95.86 W, respectively. Thus, GEAR supports more functionality, such as grafting, FBS, and PC-MM, with less area and comparable power than prior accelerators.

FAST [41] likewise reports results from a predictive TSMC 7-nm PDK at 1 GHz, so we compare their NTTUs directly. In our design, four NTTUs occupy 14.5 mm^2 of logic area and consume 19.6 W, versus 60.88 mm^2 and 142.7 W for FAST. As a FAST NTTU processes two polynomials per cycle in its 36-bit mode, we normalize to throughput: per coefficient processed per cycle, our NTTU requires 2.1× less area and 7.3× less power against FAST's 36-bit mode, and 4.2× and 7.3× less against FAST's 60-bit mode. This gap stems from three architectural choices. First, the NTTU has no transpose network, whereas FAST transposes data in multi-layer quadrant-swap transpose units within every lane group. Second, it has no twisting multipliers, as all twisting factors and the scaling by N^{-1} are merged into the CT/GS butterflies, and the column unit holds one twiddle row per polynomial, reducing toggling. Third, each modular multiplier is cheaper. NTT-friendly primes already satisfy $q \equiv 1 \pmod{2N}$, so $-q^{-1} \equiv -1 \pmod{2^W}$, and each of the $L = \lceil K/W \rceil$ Montgomery steps reduces to a single $W \times (K - W)$-bit multiply-add with no multiplication by q^{-1}. This means two 16×16-bit multiply and add steps on one 32-bit word, whereas each FAST multiplier combines three 36-bit multipliers to support 60-bit moduli. Because the first two choices remove logic that switches every cycle, the power savings exceed the area savings.

7.3 Architectural Analysis

Off-chip Traffic: Figure 9(a) breaks down GEAR's off-chip traffic across all workloads on both DS-based and Grafted execution. The breakdown is by keyswitch hint (Keys), ciphertext inputs, DFT

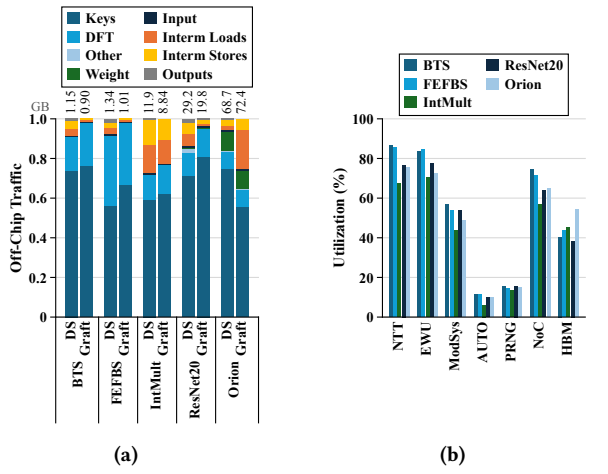


Figure 9: Per-benchmark breakdown of (a) Off-chip data movement, comparing DS and Grafted; and (b) Utilization of key GEAR components.

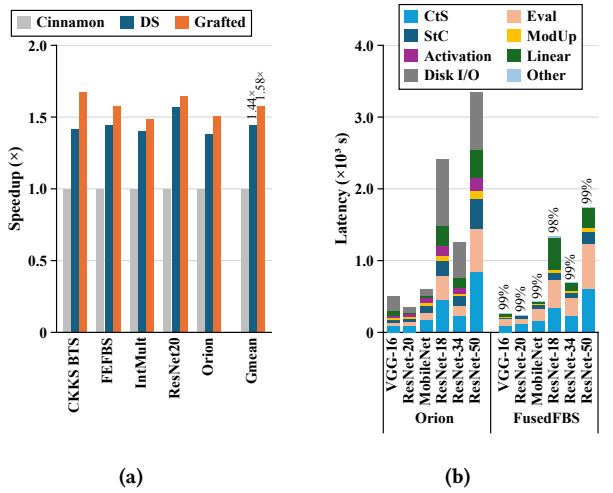


Figure 10: Ablation Study of (a) Double-prime scaling (DS) versus grafting performance; and (b) DNN inference CPU runtime breakdown for Orion-generated model and FusedFBS model.

matrices, encoded plaintext weights, and intermediate loads and stores. We see that for BTS, FEFBS, IntMult, and ResNet20 with FusedFBS, grafting reduces data movement by 22%, 25%, 26%, and 32%, respectively. This is because grafting fully utilizes word-bits and reduces ciphertext and keyswitch hint size, while with OF-limb generation, keyswitch hint dominates off-chip data movement. The Orion-generated ResNet20 remains the only exception, as the generated baby-step/giant-step are not accelerator-friendly and lead to increased scratchpad eviction and intermediate loads and stores. Meanwhile, ResNet20 with FusedFBS loads almost no weights (0.1–0.2%), since GEAR computes its convolutions as cleartext-ciphertext matrix products on ModSys instead of multiplying ciphertexts by encoded weight diagonals.

Component Utilization: Figure 9(b) reports the utilization of GEAR's components. The Mixed-Radix CG-NTTU is the busiest at 67–87% utilization, participates in most of the HE ops, and supports OF-limb generation. ModSys, which performs the base conversions as well as ResNet20's cleartext-ciphertext matrix multiplication, is mostly utilized at over 50%, which is dramatically more than [66]'s 20% because of the functionality sharing. Due to the improved memory management and optimization, together with the 256 MB scratchpad, HBM is busy only 38–54% of the time.

7.4 Ablation Studies

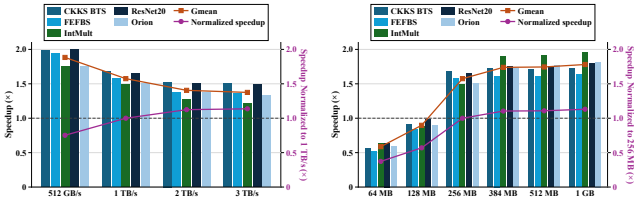
DS versus Grafting: Figure 10(a) evaluates the performance benefit of grafting. We start from the Cinnamon baseline, in which the Cinnamon accelerator executes the FHE operations with the double-prime scaling technique. DS represents the same double-prime scaling program executed on our GEAR architecture. On the same DS program, GEAR is 1.44× faster than Cinnamon on average (1.38–1.57×). This gain comes from GEAR's hardware and scheduling alone. ResNet20 benefits most (1.57×) because GEAR's **ModSys** also computes the cleartext-ciphertext matrix multiplications of its convolutions, which Cinnamon computes on its vector units and base conversion units, which leads to more modular reductions and scratchpad read and writeback. With grafting, GEAR's average speedup rises to 1.58× (1.48–1.73×), and grafting outperforms DS on every benchmark, by 1.10× on average and by 1.05–1.19× per benchmark. While the increased NTT computation for sprouts reduces grafting's performance advantage, a gain remains from higher bit-width utilization and fewer total keys and ciphertext limbs. This demonstrates that GEAR supports versatile FHE moduli and scaling factors with equal or better performance.

FusedFBS Algorithm: Table 4 compares the CPU implementation of GEAR's FusedFBS with Orion [38] across six networks, VGG-16 [103], ResNet-20/34/50 [54] on CIFAR-10 [74], and MobileNet [57] and ResNet-18 [54] on Tiny ImageNet [75]. Orion approximates SiLU with a degree-127 Chebyshev polynomial, whereas GEAR evaluates it as a degree-44 Fourier series inside its functional bootstrapping. We measure GEAR's precision and accuracy from grafted runs and its latency from separate ungrafted runs, since the grafted modulus chains tailored to 32-bit arithmetic underutilize the 64-bit CPU datapath. Precision is defined as $-\log_2$ of the mean absolute difference between the decrypted logits and those of the unencrypted network, and higher values mean better. Following Fhelipe [73], we evaluate enough samples to obtain 95% confidence intervals for accuracy with a half-width below one percentage point [53]. To match Orion's default modulus budget of $\log QP = 1764$ at 128-bit security, GEAR uses a uniform ternary secret, whereas Orion retains a sparse ternary secret. Prior studies [20, 66] report that uniform ternary secrets can increase error growth and degrade ResNet-20 accuracy with polynomial activation approximations. Under this configuration, FusedFBS achieves higher output precision, preserving inference accuracy.

GEAR achieves speedup over Orion on all six networks. Speedup first comes from Orion's disk I/O (Figure 10(b)). Orion stores each layer's encoded weights and rotation keys on an NVMe SSD and loads them for layer evaluation, because keeping them in memory would take more than 400 GB each for ResNet-34 and ResNet-50. These loads, the Disk I/O segments, take 15–39% of Orion's time and amount to over 50% of GEAR speedups. Even without the loads, Orion still takes 1.1–1.5× as long as GEAR, mostly because

Table 4: End-to-end DNN inference performance of Orion and FusedFBS (GEAR) on CPU.

	Prec. (bits)		Acc. (%)		Time (s)		
Model	Orion	GEAR	Orion	GEAR	Orion	GEAR	Speedup
VGG-16	7.2	19.5	92±1	92±1	500	267	1.87×
MobileNet	7.8	19.0	55±1	50±1	607	430	1.41×
ResNet-18	8.1	19.9	57±1	58±1	2415	1344	1.80×
ResNet-20	8.9	20.0	92±1	93±1	358	158	2.27×
ResNet-34	8.1	19.5	N/A	N/A	1248	696	1.79×
ResNet-50	7.3	18.4	N/A	N/A	3338	1744	1.91×



(a) (b)

Figure 11: Workload performance under: (a) different off-chip bandwidths; and (b) different scratchpad capacity.

of the substitution of polynomial-approximation-based activation functions with FEFBS. As shown in Figure 10(b), 98–99% of the runtime and operations are spent during bootstrapping since it contains both linear and non-linear functionalities.

7.5 Sensitivity Study

Figure 11(a) compares the performance of GEAR with off-chip bandwidth from 512 GB/s to 3 TB/s, normalized to the Cinnamon baseline. Doubling bandwidth to 2 TB/s speeds GEAR up by only 1.12×, and provides almost no more speedup beyond 2 TB/s.

Figure 11(b) presents the performance comparison of GEAR with different scratchpad capacities from 64 MB to 1 GB, while Cinnamon keeps 256 MB. A smaller scratchpad hurts every workload. GEAR reaches only 0.57× of its 256 MB performance at 128 MB and 0.37× at 64 MB. This is because the working set no longer fits on-chip, so GEAR must write back values and reload them. On the other hand, scratchpad larger than 256 MB yields diminishing returns, as 384 MB makes GEAR only 1.10× faster than with 256 MB and the speedup almost flattens for 512 MB and 1 GB. IntCKKS Mult and ResNet20 (Orion), which have the smallest speedups at 256 MB, gain the most from a larger scratchpad. At 384 MB, IntCKKS Mult runs four bootstraps, and in each one its CTS keeps nearly the whole 256 MB scratchpad full of the working set values, so GEAR evicts some active intermediate values and kshs that it will be read again within the same bootstrap. In ResNet20 (Orion), the third-stage convolutions likewise keep more baby-step rotations live than 256 MB holds, which causes most of its intermediate loads.

8 Related Work

CPU and GPU acceleration. Years of research have produced mature CPU [7, 23, 51, 86] and GPU [5, 91, 113] libraries that form the software foundation for FHE. GPU work has further refined this foundation through memory-centric kernel fusion [63], 32-bit RNS arithmetic [31], tensor-core acceleration [43, 62], register-resident NTTs [39, 40], and linear-algebra reformulation [106], with further efforts scaling to multi-GPU inference [61] and TPUs [107]. Commodity hardware nonetheless imposes two fundamental limits. First, limited on-chip capacity forces frequent off-chip traffic, and even well-optimized kernels become bound by L2 bandwidth [32]. Second, lacking native modular arithmetic, GPUs emulate it through costly instruction sequences or operand decomposition [35, 40], leaving low-precision tensor throughput underutilized. Proposed GPU extensions [35, 102] reduce these overheads but inherit the underlying execution and memory constraints. At the algorithm level, LEAF [118] combines PCMM with BFV-based FBS in a CPU-based software framework. Our FusedFBS algorithm builds on the similar PCMM primitive but pairs it with grafted CKKS Fourier-extension FBS [18, 25], and our GEAR architecture provides dedicated hardware to execute it.

FPGA/ASIC acceleration. Given the high cost of FHE operations, the community has also turned to dedicated hardware. FPGAs accelerate NTT, key switching, bootstrapping, and matrix operations [2, 4, 93, 94, 96, 108, 114], but face frequency and memory limits. ASICs target bootstrapping throughput [68, 98, 99, 120], smaller working sets [3, 59, 66, 67], arithmetic and dataflow [24, 41, 42, 71, 72, 88], multiple schemes [36, 90, 120], and scale-out [6, 60], with HERACLES realizing SIMD FHE in silicon [50]. Two limitations persist: keys and plaintexts can exceed on-chip SRAM despite generation and reuse [66–68, 99], and fixed unit ratios cause underutilization and spills [24]. Sapphire offers configurable lattice arithmetic [16], FAST runs BConv on a systolic array with output reduction [41], and FHECore shares NTT and BConv hardware [35]. Guided by our versatility and flexibility requirements, GEAR builds on these designs. ModSys shares output-stationary arithmetic and grouped reducers between BConv and PCMM, serving both key switching and linear layers. Extending constant-geometry NTTs that support variable degrees, moduli, and online twiddles such as in [58], our two-unit mixed-radix pipeline replaces the ten-step transpose and twist units of prior accelerators such as [41, 66] with buffer addressing and in-butterfly twiddles.

FBS algorithms and applications. Bootstrapping algorithms have advanced rapidly. Grafting [25] decouples the scaling factor from the moduli using sprouts, and our 32-bit execution trims its auxiliary transforms through reuse, product fusion, and rescale scheduling. MaMBo [12] fuses PCMM with batched bootstrapping; Our FusedFBS algorithm extends this fusion with grafted Fourier-extension FBS [18], ring switching, and convolutional packing, evaluating smooth DNN activations within the bootstrap. RBOOT [116] and BootNet [115] similarly fuse ReLU and convolution–ReLU into bootstrapping, replacing high-degree activation polynomials [38]. Beyond inference, FBS enables database comparison, MIN/MAX, and top-k: 32-bit sign evaluation costs 3.0 ms amortized [9], and sign bootstrapping raises precision from 21 to 32 bits, nearly halves top-k latency, and cuts bitonic sorting time by 24–27% [56]. Digit-wise FBS with CKKS modular reduction yields exact 64-bit integer arithmetic with > 1000× amortized multiplication speedup over TFHE-rs [64, 65], and AES-to-CKKS transciphering uses bit-level FBS to refresh ciphertexts and remove XOR overflow [13, 89], fusing linear layers into bootstrapping [69]. These workloads build on the same FBS and modular-arithmetic primitives that GEAR's versatile design supports.

9 Conclusion

This work presented GEAR, a versatile accelerator designed to efficiently accommodate evolving algorithms, ciphertext representations, and arithmetic requirements through shared, configurable hardware rather than separate algorithm-specific architectures. And FusedFBS, which integrates PCMM with grafted FBS. GEAR and FusedFBS showcase efficient FHE computations with emerging FHE techniques to provide a versatile FHE acceleration design starting point for embracing more next-generation FHE algorithms. Across representative workloads, GEAR achieves 3,121× speedup over a CPU baseline and a 1.58× speedup over state-of-the-art CKKS accelerators, while reducing the area overhead by around 30%. These results show that GEAR can improve FHE performance and efficiency while preserving versatility across FHE parameters and application functionality.

Acknowledgments

The authors used ChatGPT for grammar, clarity, and basic editing of author-written text. The authors also used Anthropic Claude Code for code refactoring, coding assistance, and preparation of experimental scripts. All AI-assisted material was reviewed and approved by the authors, who take full responsibility for this work.

References

[1] Rashmi Agrawal, Jung Ho Ahn, Flavio Bergamaschi, Ro Cammarota, Jung Hee Cheon, Fillipe DM de Souza, Huijing Gong, Minsik Kang, Duhyeong Kim, Jongmin Kim, et al. 2023. High-precision RNS-CKKS on fixed but smaller word-size architectures: theory and application. In *Proceedings of the 11th Workshop on Encrypted Computing & Applied Homomorphic Cryptography*. 23–34.

[2] Rashmi Agrawal, Anantha Chandrakasan, and Ajay Joshi. 2024. Heap: A fully homomorphic encryption accelerator with parallelized bootstrapping. In *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 756–769.

[3] Rashmi Agrawal, Leo de Castro, Chiraag Juvekar, Anantha Chandrakasan, Vinod Vaikuntanathan, and Ajay Joshi. 2023. Mad: Memory-aware design techniques for accelerating fully homomorphic encryption. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*. 685–697.

[4] Rashmi Agrawal, Leo de Castro, Guowei Yang, Chiraag Juvekar, Rabia Yazicigil, Anantha Chandrakasan, Vinod Vaikuntanathan, and Ajay Joshi. 2023. FAB: An FPGA-based accelerator for bootstrappable fully homomorphic encryption. In *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 882–895.

[5] Carlos Agulló-Domingo, Óscar Vera-López, Seyda Guzelhan, Lohit Daksha, Aymane El Jerari, Kaustubh Shivdikar, Rashmi Agrawal, David Kaeli, Ajay Joshi, and José L. Abellán. 2025. FIDESlib: A Fully-Fledged Open-Source FHE Library for Efficient CKKS on GPUs. In *2025 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, Ghent, Belgium. Poster paper.

[6] Aikata Aikata, Ahmet Can Mert, Sunmin Kwon, Maxim Deryabin, and Sujoy Sinha Roy. 2023. REED: Chiplet-based accelerator for fully homomorphic encryption. *arXiv preprint arXiv:2308.02885* (2023).

[7] Ahmad Al Badawi, Jack Bates, Flavio Bergamaschi, David Bruce Cousins, Saroja Erabelli, Nicholas Genise, Shai Halevi, Hamish Hunt, Andrey Kim, Yongwoo Lee, Zeyu Liu, Daniele Micciancio, Ian Quah, Yuriy Polyakov, Saraswathy R.V., Kurt Rohloff, Jonathan Saylor, Dmitriy Suponitsky, Matthew Triplett, Vinod Vaikuntanathan, and Vincent Zucca. 2022. Openfhe: Open-source fully homomorphic encryption library. In *proceedings of the 10th workshop on encrypted computing & applied homomorphic cryptography*. 53–63.

[8] Martin Albrecht, Melissa Chase, Hao Chen, Jintai Ding, Shafi Goldwasser, Sergey Gorbunov, Shai Halevi, Jeffrey Hoffstein, Kim Laine, Kristin Lauter, Satya Lokam, Daniele Micciancio, Dustin Moody, Travis Morrison, Amit Sahai, and Vinod Vaikuntanathan. 2022. Homomorphic encryption standard. In *Protecting privacy through homomorphic encryption*. Springer, 31–62.

[9] Andreea Alexandru, Andrey Kim, and Yuriy Polyakov. 2025. General functional bootstrapping using CKKS. In *Annual International Cryptology Conference*. Springer, 304–337.

[10] Andreea Alexandru, Andrey Kim, Yuriy Polyakov, and Hongren Zheng. 2026. Sparse Hermite Interpolation Method for Discrete-CKKS Functional Bootstrapping. *Cryptology ePrint Archive* (2026).

[11] Sebastian Angel, Hao Chen, Kim Laine, and Srinath Setty. 2018. PIR with compressed queries and amortized query processing. In *2018 IEEE symposium on security and privacy (SP)*. IEEE, 962–979.

[12] Youngjin Bae, Jung Hee Cheon, Guillaume Hanrot, Jai Hyun Park, and Damien Stehlé. 2024. Plaintext-ciphertext matrix multiplication and FHE bootstrapping: fast and fused. In *Annual International Cryptology Conference*. Springer, 387–421.

[13] Youngjin Bae, Jung Hee Cheon, Minsik Kang, and Taeseong Kim. 2025. High-Throughput AES Transciphering using CKKS: Less than 1ms. In *Proceedings of the 13th Workshop on Encrypted Computing & Applied Homomorphic Computing*. 1–12.

[14] Youngjin Bae, Jung Hee Cheon, Jaehyung Kim, and Damien Stehlé. 2024. Bootstrapping bits with CKKS. In *Annual international conference on the theory and applications of cryptographic techniques*. Springer, 94–123.

[15] Youngjin Bae, Jaehyung Kim, Damien Stehlé, and Elias Suvanto. 2024. Bootstrapping small integers with CKKS. In *International Conference on the Theory and Application of Cryptology and Information Security*. Springer, 330–360.

[16] Utsav Banerjee, Tenzin S. Ukyab, and Anantha P. Chandrakasan. 2019. Sapphire: A configurable crypto-processor for post-quantum lattice-based protocols. *arXiv preprint arXiv:1910.07557* (2019).

[17] Laszlo A. Belady. 1966. A study of replacement algorithms for a virtual-storage computer. *IBM Systems journal* 5, 2 (1966), 78–101.

[18] Song Bian, Yunhao Fu, Ruiyu Shen, Haowen Pan, Anyu Wang, and Zhenyu Guan. 2026. High-precision Functional Bootstrapping for CKKS from Fourier Extension. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 274–303.

[19] Song Bian, Haowen Pan, Jiaqi Hu, Zhou Zhang, Yunhao Fu, Jiafeng Hua, Yi Chen, Bo Zhang, Yier Jin, Jin Dong, and Zhenyu Guan. 2025. Engorgio: An Arbitrary-Precision Unbounded-Size Hybrid Encrypted Database via Quantized Fully Homomorphic Encryption. *Cryptology ePrint Archive* (2025).

[20] Jean-Philippe Bossuat, Christian Mouchet, Juan Troncoso-Pastoriza, and Jean-Pierre Hubaux. 2021. Efficient Bootstrapping for Approximate Homomorphic Encryption with Non-sparse Keys. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. 587–617.

[21] Gyeongwon Cha, Dongjin Park, and Joon-Woo Lee. 2026. Improved Radix-Based Approximate Homomorphic Encryption for Large Integers via Lightweight Bootstrapped Digit Carry. In *Advances in Cryptology – EUROCRYPT 2026*, Joan Daemen and Emmanuel Thomé (Eds.). Springer Nature Switzerland, Cham, 243–273.

[22] Jonathan Chang, Yen-Huei Chen, Wei-Min Chan, Sahil Preet Singh, Hank Cheng, Hidehiro Fujiwara, Jih-Yu Lin, Kao-Cheng Lin, John Hung, Robin Lee, Hung-Jen Liao, Jhon-Jhy Liaw, Quincy Li, Chih-Yung Lin, Mu-Chi Chiang, and Shien-Yang Wu. 2017. 12.1 a 7nm 256mb sram in high-k metal-gate finfet technology with write-assist circuitry for low-v min applications. In *2017 IEEE International Solid-State Circuits Conference (ISSCC)*. IEEE, 206–207.

[23] Hao Chen, Kim Laine, and Rachel Player. 2017. Simple encrypted arithmetic library-SEAL v2. 1. In *International conference on financial cryptography and data security*. Springer, 3–18.

[24] Xinhua Chen, Jiangbin Dong, Hongren Zheng, Tian Tang, and Mingyu Gao. 2026. CROPHE: Cross-Operator Dataflow Optimization for Fully Homomorphic Encryption Accelerators. In *2026 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 1–14.

[25] Jung Hee Cheon, Hyeongmin Choe, Minsik Kang, Jaehyung Kim, Seonghak Kim, Johannes Mono, and Taeyeong Noh. 2025. Grafting: decoupled scale factors and modulus in RNS-CKKS. In *Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security*. 1098–1112.

[26] Jung Hee Cheon, Kyoohyung Han, Andrey Kim, Miran Kim, and Yongsoo Song. 2018. Bootstrapping for approximate homomorphic encryption. In *37th Annual International Conference on the Theory and Applications of Cryptographic Techniques, EUROCRYPT 2018*. Springer Verlag, 360–384.

[27] Jung Hee Cheon, Kyoohyung Han, Andrey Kim, Miran Kim, and Yongsoo Song. 2018. A full RNS variant of approximate homomorphic encryption. In *International Conference on Selected Areas in Cryptography*. Springer, 347–368.

[28] Jung Hee Cheon, Minsik Kang, Taeseong Kim, Junyoung Jung, and Yongdong Yeo. 2024. Batch Inference on Deep Convolutional Neural Networks With Fully Homomorphic Encryption Using Channel-By-Channel Convolutions. *IEEE Transactions on Dependable and Secure Computing* (2024).

[29] Jung Hee Cheon, Andrey Kim, Miran Kim, and Yongsoo Song. 2017. Homomorphic encryption for arithmetic of approximate numbers. In *International Conference on the Theory and Application of Cryptology and Information Security*. 409–437.

[30] Ilaria Chillotti, Marc Joye, and Pascal Paillier. 2021. Programmable Bootstrapping Enables Efficient Homomorphic Inference of Deep Neural Networks. In *Cyber Security Cryptography and Machine Learning*, Shlomi Dolev, Oded Margalit, Benny Pinkas, and Alexander Schwarzmann (Eds.). Springer International Publishing, Cham, 1–19.

[31] Wonseok Choi, Jongmin Kim, and Jung Ho Ahn. 2026. Cheddar: A swift fully homomorphic encryption library designed for gpu architectures. In *Proceedings of the 31st ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*. 35–49.

[32] Wonseok Choi, Hyunah Yu, Jongmin Kim, Hyesung Ji, Jaiyoung Park, and Jung Ho Ahn. 2026. Theodosian: A Deep Dive into Memory-Hierarchy-Centric FHE Acceleration. In *2026 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 299–310.

[33] Lawrence T. Clark, Vinay Vashishtha, Lucian Shifren, Aditya Gujja, Saurabh Sinha, Brian Cline, Chandarasekaran Ramamurthy, and Greg Yeric. 2016. ASAP7: A 7-nm finFET predictive process design kit. *Microelectronics Journal* 53 (2016), 105–115.

[34] Eric Crockett, Craig Gentry, Hyojun Kim, Yeongmin Lee, and Yongwoo Lee. 2026. Efficient bootstrapping in fully homomorphic encryption for matrix arithmetic. In *Annual International Cryptology Conference*. Springer, 492–524.

[35] Lohit Daksha, Seyda Guzelhan, Kaustubh Shivdikar, Carlos Agulló Domingo, Óscar Vera Lopez, Gilbert Jonatan, Hubert Dymarkowski, Aymane El Jerari, José Cano, José L. Abellán, John Kim, David Kaeli, and Ajay Joshi. 2026. FHECore: Rethinking GPU Microarchitecture for Fully Homomorphic Encryption. *arXiv preprint arXiv:2602.22229* (2026).

[36] Xianglong Deng, Shengyu Fan, Zhicheng Hu, Zhuoyu Tian, Zihao Yang, Jiangrui Yu, Dingyuan Cao, Dan Meng, Rui Hou, Meng Li, Qian Lou, and Mingzhe Zhang. 2024. Trinity: A general purpose fhe accelerator. In *2024 57th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 338–351.

[37] Nir Drucker, Guy Moshkowich, Tomer Pelleg, and Hayim Shaul. 2024. BLEACH: Cleaning Errors in Discrete Computations Over CKKS: N. Drucker et al. *Journal of Cryptology* 37, 1 (2024), 3.

[38] Austin Ebel, Karthik Garimella, and Brandon Reagen. 2025. Orion: A Fully Homomorphic Encryption Framework for Deep Learning. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. 734–749.

[39] Guang Fan, Yi Chen, Lei Chen, Liang Kong, Chao Niu, Dian Jiao, Yilan Zhu, Geng Yang, Shengyu Fan, Xianglong Deng, Fangyu Zheng, Jian Weng, Meng Li, Yisong Chang, Shoumeng Yan, and Mingzhe Zhang. 2026. HyperDrive: Hierarchical Exploitation of Memory Efficiency for GPU-Based FHE Acceleration. In *2026 ACM/IEEE 53rd Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 1955–1969.

[40] Guang Fan, Mingzhe Zhang, Fangyu Zheng, Shengyu Fan, Tian Zhou, Xianglong Deng, Wenxu Tang, Liang Kong, Yixuan Song, and Shoumeng Yan. 2025. Warpdrive: Gpu-based fully homomorphic encryption acceleration leveraging tensor and cuda cores. In *2025 IEEE international symposium on high performance computer architecture (HPCA)*. IEEE, 1187–1200.

[41] Shengyu Fan, Xianglong Deng, Liang Kong, Guiming Shi, Guang Fan, Dan Meng, Rui Hou, and Mingzhe Zhang. 2025. FAST: An FHE Accelerator for Scalable-parallelism with Tunable-bit. In *Proceedings of the 52nd Annual International Symposium on Computer Architecture*. 92–106.

[42] Shengyu Fan, Xianglong Deng, Zhuoyu Tian, Zhicheng Hu, Liang Chang, Rui Hou, Dan Meng, and Mingzhe Zhang. 2024. TaiYI: A high-performance CKKS accelerator for practical fully homomorphic encryption. *arXiv preprint arXiv:2403.10188* (2024).

[43] Shengyu Fan, Zhiwei Wang, Weizhi Xu, Rui Hou, Dan Meng, and Mingzhe Zhang. 2023. Tensorfhe: Achieving practical computation on encrypted data using gpgpu. In *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 922–934.

[44] Hidehiro Fujiwara, Yi-Hsin Nien, Chih-Yu Lin, Hsien-Yu Pan, Hao-Wen Hsu, Shin-Rung Wu, Yao-Yi Liu, Yen-Huei Chen, Hung-Jen Liao, and Jonathan Chang. 2021. 24.4 A 5nm 5.7 GHz@ 1.0 V and 1.3 GHz@ 0.5 V 4kb standard-cell-based two-port register file with a 16T bitcell with no half-selection issue. In *2021 IEEE International Solid-State Circuits Conference (ISSCC)*, Vol. 64. IEEE, 1–3.

[45] Mingyu Gao and Hongren Zheng. 2026. FHE for SIMD Arithmetic Logic Units with Amortized O(1) Bootstrapping per Ciphertext. In *Advances in Cryptology – CRYPTO 2026*, Nadia Heninger and Mike Rosulek (Eds.). Springer Nature Switzerland, Cham, 299–330.

[46] Harvey L Garner. 1959. The residue number system. In *Papers presented at the March 3-5, 1959, western joint computer conference*. 146–153.

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

[48] Craig Gentry, Shai Halevi, Chris Peikert, and Nigel P Smart. 2012. Ring switching in BGV-style homomorphic encryption. In *International conference on security and cryptography for networks*. Springer, 19–37.

[49] Craig Gentry and Yongwoo Lee. 2026. Fully homomorphic encryption for matrix arithmetic. In *Annual International Cryptology Conference*. Springer, 363–395.

[50] Anupam Golder, Raghavan Kumar, Sachin Taneja, Kylan Race, Paolo A. Aseron, James Greensky, Wen Wang, Huijing Gong, Lalith Kethareswaran, Vikram B. Suresh, Adish Pradeep Vartak, AppaRao Challagundla, Jeremy Casas, Poornima Lalwaney, Duhyeong Kim, Christopher N. Gutierrez, Ernesto Zamora Ramos, Wonhee Cho, Jose M. Rojas Chaves, Michael Steiner, Dan Lake, Nataraj Yennampelli, Karthik Nivarthi, Kamalakanth Bijinapally, Bala Prasad Talamala, Sravanth Valluri, Vasantha Srirambhatla, Chris Wilkerson, Rosario Cammarota, and Sanu K. Mathew. 2026. HERACLES: 8192-Way SIMD Programmable Scalable Fully-Homomorphic Encryption SoC for Privacy-Preserving Cloud Computing in Intel 3 CMOS. In *2026 IEEE International Solid-State Circuits Conference (ISSCC)*, Vol. 69. 424–426. doi:10.1109/ISSCC49663.2026.11409291

[51] Shai Halevi and Victor Shoup. 2014. Algorithms in helib. In *Annual Cryptology Conference*. Springer, 554–571.

[52] Kyoohyung Han and Dohyeong Ki. 2020. Better bootstrapping for approximate homomorphic encryption. In *Cryptographers' Track at the RSA Conference*. Springer, 364–390.

[53] James A Hanley and Abby Lippman-Hand. 1983. If nothing goes wrong, is everything all right?: interpreting zero numerators. *Jama* 249, 13 (1983), 1743–1745.

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

[55] Bernd Hoefflinger. 2020. Irds—international roadmap for devices and systems, rebooting computing, s3s. *NANO-CHIPS 2030: On-Chip AI for an Efficient Data-Driven World* (2020), 9–17.

[56] Deokhwa Hong, Heesoo Lee, Young-Sik Kim, and Yongwoo Lee. 2026. A White-Box Bootstrapping Approach for High Precision Comparison Over Homomorphic Encryption. *Cryptology ePrint Archive* (2026).

[57] Andrew G Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. 2017. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861* (2017).

[58] Jingwen Huang, Chiayi Kuo, Sihuang Liu, and Tao Su. 2024. An Area-Efficient and Configurable Number Theoretic Transform Accelerator for Homomorphic Encryption. *Electronics* 13, 17 (2024), 3382. doi:10.3390/electronics13173382

[59] Yi Huang, Xinsheng Gong, Xiangyu Kong, Dibei Chen, Jianfeng Zhu, Wenping Zhu, Liangwei Li, Mingyu Gao, Shaojun Wei, Aoyang Zhang, and Leibo Liu. 2025. EFFACT: A Highly Efficient Full-Stack FHE Acceleration Platform. In *2025 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 1143–1157.

[60] Siddharth Jayashankar, Edward Chen, Tom Tang, Wenting Zheng, and Dimitrios Skarlatos. 2025. Cinnamon: A Framework for Scale-Out Encrypted AI. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*. 133–150.

[61] Siddharth Jayashankar, Joshua Kim, Michael B Sullivan, Wenting Zheng, and Dimitrios Skarlatos. 2026. Cerium: A Multi-GPU Framework for Terabyte-Scale Encrypted Inference. In *Proceedings of the ACM SIGOPS 32nd Symposium on Operating Systems Principles*. 193–210.

[62] Dian Jiao, Xianglong Deng, Zhiwei Wang, Shengyu Fan, Yi Chen, Dan Meng, Rui Hou, and Mingzhe Zhang. 2025. Neo: Towards Efficient Fully Homomorphic Encryption Acceleration using Tensor Core. In *Proceedings of the 52nd Annual International Symposium on Computer Architecture*. 107–121.

[63] Wonkyung Jung, Sangpyo Kim, Jung Ho Ahn, Jung Hee Cheon, and Younho Lee. 2021. Over 100x faster bootstrapping in fully homomorphic encryption through memory-centric optimization with GPUs. *IACR Transactions on Cryptographic Hardware and Embedded Systems* (2021), 114–148.

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

[65] Jaehyung Kim. 2026. Faster homomorphic integer computer. *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2026, 3 (2026), 1128–1151.

[66] Jongmin Kim, Sangpyo Kim, Jaewan Choi, Jaiyoung Park, Donghwan Kim, and Jung Ho Ahn. 2023. SHARP: A short-word hierarchical accelerator for robust and practical fully homomorphic encryption. In *Proceedings of the 50th Annual International Symposium on Computer Architecture*. 1–15.

[67] Jongmin Kim, Gwangho Lee, Sangpyo Kim, Gina Sohn, Minsoo Rhu, John Kim, and Jung Ho Ahn. 2022. Ark: Fully homomorphic encryption accelerator with runtime data generation and inter-operation key reuse. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 1237–1254.

[68] Sangpyo Kim, Jongmin Kim, Michael Jaemin Kim, Wonkyung Jung, John Kim, Minsoo Rhu, and Jung Ho Ahn. 2022. Bts: An accelerator for bootstrappable fully homomorphic encryption. In *Proceedings of the 49th annual international symposium on computer architecture*. 711–725.

[69] Taeseong Kim, Jonghoo Lee, Taeyeong Noh, Jung Hee Cheon, and Guillaume Hanrot. 2026. Latency-Aware, High-Throughput Homomorphic AES Evaluation with CKKS. *Cryptology ePrint Archive* (2026).

[70] Kamil Kluczniak and Leonard Schild. 2022. FDFB: Full Domain Functional Bootstrapping Towards Practical Fully Homomorphic Encryption. *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2023, 1 (Nov. 2022), 501–537. doi:10.46586/tches.v2023.i1.501-537

[71] Liang Kong, Xianglong Deng, Guang Fan, Shengyu Fan, Lei Chen, Yilan Zhu, Geng Yang, Yisong Chang, Shoumeng Yan, and Mingzhe Zhang. 2026. Falcon: Algorithm-Hardware Co-Design for Efficient Fully Homomorphic Encryption Accelerator. In *Proceedings of the 31st ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. 715–731.

[72] Liang Kong, Shengyu Fan, Xianglong Deng, Lei Chen, Guang Fan, Guiming Shi, Yilan Zhu, Geng Yang, Shoumeng Yan, and Mingzhe Zhang. 2025. HAWK: Fully Homomorphic Encryption Accelerator with Fixed-Word Key Decomposition Switching. In *Proceedings of the 58th IEEE/ACM International Symposium on Microarchitecture*. 1719–1734.

[73] Aleksandar Krastev, Nikola Samardzic, Simon Langowski, Srinivas Devadas, and Daniel Sanchez. 2024. A tensor compiler with automatic data packing for simple and efficient fully homomorphic encryption. *Proceedings of the ACM on Programming Languages* 8, PLDI (2024), 126–150.

[74] Alex Krizhevsky, Geoffrey Hinton, et al. 2009. Learning multiple layers of features from tiny images. (2009).

[75] Yann Le, Xuan Yang, et al. 2015. Tiny imagenet visual recognition challenge. *CS 231N* 7, 7 (2015), 3.

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

[77] Joon-Woo Lee, HyungChul Kang, Yongwoo Lee, Woosuk Choi, Jieun Eom, Maxim Deryabin, Eunsang Lee, Junghyun Lee, Donghoon Yoo, Young-Sik Kim, and Jong-Seon No. 2022. Privacy-preserving machine learning with fully homomorphic encryption for deep neural network. *iEEE Access* 10 (2022), 30039–30054.

[78] Joon-Woo Lee, Eunsang Lee, Yongwoo Lee, Young-Sik Kim, and Jong-Seon No. 2021. High-precision bootstrapping of RNS-CKKS homomorphic encryption using optimal minimax polynomial approximation and inverse sine function. In *Annual International Conference on the Theory and Applications of Cryptographic*

Techniques. Springer, 618–647.

[79] Zeyu Liu and Yunhao Wang. 2023. Amortized Functional Bootstrapping in Less than 7 ms, with $\tilde{O}(1)$ Polynomial Multiplications. In *Advances in Cryptology – ASIACRYPT 2023*, Jian Guo and Ron Steinfeld (Eds.). Springer Nature Singapore, Singapore, 101–132.

[80] Zeyu Liu and Yunhao Wang. 2025. Relaxed Functional Bootstrapping: A New Perspective on BGV/BFV Bootstrapping. In *Advances in Cryptology – ASIACRYPT 2024*, Kai-Min Chung and Yu Sasaki (Eds.). Springer Nature Singapore, Singapore, 208–240.

[81] Federico Mazzone, Maarten Everts, Florian Hahn, and Andreas Peter. 2025. Efficient Ranking, Order Statistics, and Sorting under {CKKS}. In *34th USENIX Security Symposium (USENIX Security 25)*. 8541–8558.

[82] Samir Jordan Menon and David J Wu. 2022. Spiral: Fast, high-rate single-server PIR via FHE composition. In *2022 IEEE symposium on security and privacy (SP)*. IEEE, 930–947.

[83] Ahmet Can Mert, Emre Karabulut, Erdinç Öztürk, Erkay Savaş, and Aydin Aysu. 2020. An extensive study of flexible design methods for the number theoretic transform. *IEEE Trans. Comput.* 71, 11 (2020), 2829–2843.

[84] Ahmet Can Mert, Erdinç Öztürk, and Erkay Savaş. 2019. Design and implementation of a fast and scalable NTT-based polynomial multiplier architecture. In *2019 22nd Euromicro Conference on Digital System Design (DSD)*. IEEE, 253–260.

[85] Jungho Moon, Dongwoo Yoo, Xiaoqian Jiang, and Miran Kim. 2025. THOR: Secure transformer inference with homomorphic encryption. In *Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security*. 3765–3779.

[86] Christian Vincent Mouchet, Jean-Philippe Bossuat, Juan Ramón Troncoso-Pastoriza, and Jean-Pierre Hubaux. 2020. Lattigo: A multiparty homomorphic encryption library in go. In *Proceedings of the 8th Workshop on Encrypted Computing and Applied Homomorphic Cryptography*. 64–70.

[87] Yujin Nam, Abhishek Moitra, Yeshwanth Venkatesha, Xiaofan Yu, Gabrielle De Micheli, Xuan Wang, Minxuan Zhou, Augusto Vega, Priyadarshini Panda, and Tajana Rosing. 2025. Rhychee-FL: Robust and Efficient Hyperdimensional Federated Learning with Homomorphic Encryption. In *2025 Design, Automation & Test in Europe Conference (DATE)*. IEEE, 1–7.

[88] Negar Neda, Austin Ebel, Benedict Reynwar, and Brandon Reagen. 2024. CiFlow: Dataflow analysis and optimization of key switching for homomorphic encryption. In *2024 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 61–72.

[89] Chao Niu, Zhicong Huang, Zhaomin Yang, Yi Chen, Liang Kong, Cheng Hong, and Tao Wei. 2025. Xboot: Free-xor gates for ckks with applications to transciphering. *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2025, 4 (2025), 118–144.

[90] Qingyun Niu, Lutan Zhao, Ming Cai, Kai Li, Dan Meng, and Rui Hou. 2026. UniFHE: Faster Accelerator for FHE with Diverse Algebraic Structure and Balanced Memory System. In *2026 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 1–14.

[91] Ali Şah Özcan and Erkay Savaş. 2024. Heongpu: a gpu-based fully homomorphic encryption library 1.0. *Cryptology ePrint Archive* (2024).

[92] Jai Hyun Park. 2025. Ciphertext-ciphertext matrix multiplication: Fast for large matrices. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 153–180.

[93] Xuanle Ren, Zhaohui Chen, Zhen Gu, Yanheng Lu, Ruiguang Zhong, Wen-Jie Lu, Jiansong Zhang, Yichi Zhang, Hanghang Wu, Xiaofu Zheng, Heng Liu, Tingqiang Chu, Cheng Hong, Changzheng Wei, Dimin Niu, and Yuan Xie. 2023. CHAM: A customized homomorphic encryption accelerator for fast matrix-vector product. In *2023 60th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 1–6.

[94] M. Sadegh Riazi, Kim Laine, Blake Pelton, and Wei Dai. 2020. HEAX: An architecture for computing on encrypted data. In *Proceedings of the twenty-fifth international conference on architectural support for programming languages and operating systems*. 1295–1309.

[95] Arun F Rodrigues, K Scott Hemmert, Brian W Barrett, Chad Kersey, Ron Oldfield, Marlo Weston, Rolf Risen, Jeanine Cook, Paul Rosenfeld, Elliot Cooper-Balis, et al. 2011. The structural simulation toolkit. *ACM SIGMETRICS Performance Evaluation Review* 38, 4 (2011), 37–42.

[96] Sujoy Sinha Roy, Furkan Turan, Kimmo Järvinen, Frederik Vercauteren, and Ingrid Verbauwhede. 2019. FPGA-based high-performance parallel architecture for homomorphic computing on encrypted data. In *2019 IEEE International symposium on high performance computer architecture (HPCA)*. IEEE, 387–398.

[97] Richard M Russell. 1978. The CRAY-1 computer system. *Commun. ACM* 21, 1 (1978), 63–72.

[98] Nikola Samardzic, Axel Feldmann, Aleksandar Krastev, Srinivas Devadas, Ronald Dreslinski, Christopher Peikert, and Daniel Sanchez. 2021. F1: A fast and programmable accelerator for fully homomorphic encryption. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*. 238–252.

[99] Nikola Samardzic, Axel Feldmann, Aleksandar Krastev, Nathan Manohar, Nicholas Genise, Srinivas Devadas, Karim Eldefrawy, Chris Peikert, and Daniel Sanchez. 2022. Craterlake: a hardware accelerator for efficient unbounded computation on encrypted data. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*. 173–187.

[100] Nikola Samardzic and Daniel Sanchez. 2024. Bitpacker: Enabling high arithmetic efficiency in fully homomorphic encryption accelerators. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. 137–150.

[101] Alireza Shafaei, Yanzhi Wang, Xue Lin, and Massoud Pedram. 2014. FinCACTI: Architectural analysis and modeling of caches with deeply-scaled FinFET devices. In *2014 IEEE Computer Society Annual Symposium on VLSI*. IEEE, 290–295.

[102] Kaustubh Shivdikar, Yuhui Bao, Rashmi Agrawal, Michael Shen, Gilbert Jonatan, Evelio Mora, Alexander Ingare, Neal Livesay, José L. Abellán, John Kim, Ajay Joshi, and David Kaeli. 2023. Gme: Gpu-based microarchitectural extensions to accelerate homomorphic encryption. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*. 670–684.

[103] Karen Simonyan and Andrew Zisserman. 2014. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556* (2014).

[104] Taejoong Song, Jonghoon Jung, Woojin Rim, Hoonki Kim, Yongho Kim, Changnam Park, Jeongho Do, Sunghyun Park, Sungwee Cho, Hyuntaek Jung, Bongjae Kwon, Hyun-Su Choi, JaeSeung Choi, and Jong Shik Yoon. 2018. A 7nm FinFET SRAM using EUV lithography with dual write-driver-assist circuitry for low-voltage applications. In *2018 IEEE International Solid-State Circuits Conference-(ISSCC)*. IEEE, 198–200.

[105] A Spitzbart. 1960. A generalization of Hermite's interpolation formula. *The American Mathematical Monthly* 67, 1 (1960), 42–46.

[106] Yintai Sun, Shengyu Fan, Zhenhua Yin, Xinkai Song, Xing Hu, Zidong Du, Qi Guo, Weizhi Xu, Rui Hou, Dan Meng, Song Bian, and Mingzhe Zhang. 2026. TensorFHE+: Fully Homomorphic Encryption Acceleration Based on Linear Algebra. *IEEE Trans. Comput.* 75, 2 (2026), 612–627.

[107] Jianming Tong, Tianhao Huang, Jingtian Dang, Leo de Castro, Anirudh Itagi, Anupam Golder, Asra Ali, Jeremy Kun, Jevin Jiang, Arvind, G. Edward Suh, and Tushar Krishna. 2026. Leveraging ASIC AI Chips for Homomorphic Encryption. In *2026 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 1–18.

[108] Furkan Turan, Sujoy Sinha Roy, and Ingrid Verbauwhede. 2020. HEAWS: An accelerator for homomorphic encryption on the Amazon AWS FPGA. *IEEE Trans. Comput.* 69, 8 (2020), 1185–1196.

[109] Xuan Wang, Tianqi Zhang, Gabrielle De Micheli, Yujin Nam, Keming Fan, Augusto Vega, Minxuan Zhou, and Tajana Rosing. 2026. Rosemary: Efficient and Adaptable In-Storage Acceleration for Fully Homomorphic Encryption. *arXiv preprint arXiv:2610.38262* (2026).

[110] Xuan Wang, Minxuan Zhou, Gabrielle De Micheli, Yujin Nam, Sumukh Pinge, Augusto Vega, and Tajana Rosing. 2025. PATHE: A Privacy-Preserving Database Pattern Search Platform with Homomorphic Encryption. In *IEEE/ACM International Conference on Computer-Aided Design*.

[111] Shien-Yang Wu, C.Y. Lin, M.C. Chiang, J.J. Liaw, J.Y. Cheng, S.H. Yang, C.H. Tsai, P.N. Chen, T. Miyashita, C.H. Chang, V.S. Chang, K.H. Pan, J.H. Chen, Y.S. Mor, K.T. Lai, C.S. Liang, H.F. Chen, S.Y. Chang, C.J. Lin, C.H. Hsieh, R.F. Tsui, C.H. Yao, C.C. Chen, R. Chen, C.H. Lee, H.J. Lin, C.W. Chang, K.W. Chen, M.H. Tsai, K.S. Chen, Y. Ku, and S.M. Jang. 2016. A 7nm CMOS platform technology featuring 4 th generation FinFET transistors with a 0.027 um 2 high density 6-T SRAM cell for mobile SoC applications. In *2016 IEEE International Electron Devices Meeting (IEDM)*. IEEE, 2–6.

[112] Pan Xiao, Rending Ouyang, Heng Zhang, Jiawen Zhang, and Jian Liu. 2026. Bootstrapping is All You Need: Secure Transformer Inference via Improved CKKS Functional Bootstrapping. Cryptology ePrint Archive, Paper 2026/1255. https://eprint.iacr.org/2026/1255

[113] Hao Yang, Shiyu Shen, Wangchen Dai, Lu Zhou, Zhe Liu, and Yunlei Zhao. 2024. Phantom: A cuda-accelerated word-wise homomorphic encryption library. *IEEE Transactions on Dependable and Secure Computing* 21, 5 (2024), 4895–4906.

[114] Yinghao Yang, Huaizhi Zhang, Shengyu Fan, Hang Lu, Mingzhe Zhang, and Xiaowei Li. 2023. Poseidon: Practical homomorphic encryption accelerator. In *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 870–881.

[115] Zhaomin Yang, Chao Niu, Cheng Hong, and Tao Wei. 2026. BootNet: Homomorphic CNN Inference with Convolution and ReLU Fused in Bootstrapping. *Cryptology ePrint Archive* (2026).

[116] Zhaomin Yang, Chao Niu, Benqiang Wei, Zhicong Huang, Cheng Hong, and Tao Wei. 2026. {RBOOT}: Accelerating Homomorphic Neural Network Inference by Fusing {ReLU} within Bootstrapping. In *35th USENIX Security Symposium (USENIX Security 26)*. 121–140.

[117] Jiawen Zhang, Xinpeng Yang, Lipeng He, Kejia Chen, Wen-jie Lu, Yinghao Wang, Xiaoyang Hou, Jian Liu, Kui Ren, and Xiaohu Yang. 2024. Secure transformer inference made non-interactive. *Cryptology ePrint Archive* (2024).

[118] Linru Zhang, Xiangning Wang, Xianhui Lu, Huaxiong Wang, and Kwok-Yan Lam. 2026. LEAF: A Low-Latency Evaluation Architecture for Feedforward Block in Privacy-Preserving Transformer Inference. *IEEE Transactions on Dependable and Secure Computing* 23, 6 (2026), 12086–12103. doi:10.1109/TDSC.

2026.3718538

[119] Zhou Zhang, Song Bian, Zian Zhao, Ran Mao, Haoyi Zhou, Jiafeng Hua, Yier Jin, and Zhenyu Guan. 2024. ArcEDB: an arbitrary-precision encrypted database via (amortized) modular homomorphic encryption. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*. 4613–4627.

[120] Minxuan Zhou, Yujin Nam, Xuan Wang, Youhak Lee, Chris Wilkerson, Raghavan Kumar, Sachin Taneja, Sanu Mathew, Rosario Cammarota, and Tajana Rosing. 2024. UFC: A unified accelerator for fully homomorphic encryption. In *2024 57th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 352–365.