Cheddar: A Swift Fully Homomorphic Encryption Library Designed for GPU Architectures

Wonseok Choi*
Seoul National University
Seoul, Republic of Korea
wonseok.choi@snu.ac.kr

Jongmin Kim*
Seoul National University
Seoul, Republic of Korea
jongmin.kim@snu.ac.kr

Jung Ho Ahn
Seoul National University
Seoul, Republic of Korea
gajh@snu.ac.kr

Abstract

Fully homomorphic encryption (FHE) frees cloud computing from privacy concerns by enabling secure computation on encrypted data. However, its substantial computational and memory overhead results in significantly slower performance compared to unencrypted processing. To mitigate this overhead, we present Cheddar, a high-performance FHE library for GPUs, achieving substantial speedups over previous GPU implementations. We systematically enable 32-bit FHE execution, leveraging the 32-bit integer datapath within GPUs. We optimize GPU kernels using efficient low-level primitives and algorithms tailored to specific GPU architectures. Further, we alleviate the memory bandwidth burden by adjusting common FHE operational sequences and extensively applying kernel fusion. Cheddar delivers performance improvements of 2.18–4.45× for representative FHE workloads compared to state-of-the-art GPU implementations.

CCS Concepts: • **Security and privacy → Cryptography**; • **Computer systems organization → Single instruction, multiple data**.

Keywords: Fully homomorphic encryption, Cryptography, GPU, Security

ACM Reference Format:
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 (ASPLOS '26), March 22–26, 2026, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3760250.3762223

1 Introduction

The ubiquitous application of cloud services across increasingly diverse domains has led to a rapid expansion in user data collection, raising serious concerns about security and privacy. Under such circumstances, homomorphic encryption (HE [5]) is gaining attention due to its ability to provide strong cryptographic guarantees for user privacy. HE enables the processing of encrypted data (ciphertexts), allowing clients to offload computations on private data to cloud servers without ever exposing the underlying information.

Fully homomorphic encryption (FHE [26]) is a special subclass of HE that supports a mechanism called bootstrapping (`bts`). Without `bts`, only a fixed number of operations can be performed on a ciphertext; `bts` refreshes the ciphertext to enable practically an infinite number of operations on it.

Despite these strengths, hurdles remain for the real-world use of FHE, as processing encrypted data with FHE is several orders of magnitude slower than unencrypted computation [37]. Fortunately, FHE operations exhibit high degrees of parallelism [46], allowing for significant performance improvements through hardware acceleration using parallel computing. Previous studies have demonstrated remarkable performance improvements on various hardware systems, including CPUs [8, 37, 54], GPUs [23, 25, 36, 54, 64], FPGAs [3, 25, 33, 57, 58], and custom ASICs [42, 43, 46, 59, 60].

This study focuses on GPU acceleration of CKKS [15], a prominent FHE scheme that exhibits the highest throughput among FHE schemes [6]. CKKS is adequate for various practical applications (e.g., machine learning [29, 34, 47–49, 56]) due to its capability to handle real (or complex) numbers with high arithmetic precision. Prior GPU acceleration studies have shown fruitful results. While Lee et al. [47] reported that encrypted convolutional neural network (CNN) inference with the ResNet-20 model (CIFAR-10) took 2271 seconds on a single-core CPU, recent work [23] reduces the time to 5.88 seconds (386× faster) with an NVIDIA A100 GPU.

Despite such impressive results, we identify opportunities for further improvement; adopting *32-bit residue number system (RNS)* is especially promising for GPU architectures, which natively utilize 32-bit datapath for integer operations. To meet cryptographic requirements, popular FHE libraries [4, 11, 18, 28, 52] use 64-bit primes for RNS decomposition (64-bit RNS). On GPUs, this would incur a quadratic growth in computational overhead as GPUs emulate 64-bit integer operations with their 32-bit datapath. Recent studies introduced advanced mechanisms for 32-bit RNS (e.g.,

*Wonseok Choi and Jongmin Kim contributed equally to this work.

ACM ISBN 979-8-4007-2165-6/2026/03
https://doi.org/10.1145/3760250.3762223

rational rescaling [12, 61], see §3), which enable equivalent cryptographic parameter choices with only 32-bit primes.

However, previous 32-bit RNS constructions are not ideal for the use with GPUs. We identify that they either require excessive memory capacity requirements for additional public keys [61] or reduce the parallelizability due to control divergence [12].

To overcome these limitations, we introduce **Cheddar**, a swift GPU library for FHE CKKS, featuring a novel systemized 32-bit RNS construction (25-30 prime system). We develop a method to minimize the number of unique primes used for RNS, which allows using a single highly compatible public key, without control divergences. We also propose an inverted-terminal data layout (§3.3), which augments our 32-bit RNS construction by simplifying contiguous data allocation through appropriate prime ordering.

Cheddar also includes numerous optimizations at various levels. We develop optimized low-level kernels for major polynomial operations supporting our 32-bit RNS construction (§4). We implement the kernels with signed Montgomery reduction [62] to reduce the number of integer operations and devise optimization methods for number-theoretic transform (NTT) and basis conversion (BConv), which can be selectively applied according to the computational and memory characteristics of a specific GPU device.

While our optimized 32-bit kernels effectively reduce the cost of compute-intensive operations, memory-bound automorphism and element-wise operations remain as performance bottlenecks. To address this, we apply extensive kernel fusion techniques, categorized into sequential and parallel fusion (§5). For sequential fusion, we propose the kernel fusion methods that completely eliminate the cost of operations. For parallel fusion, we deeply analyze FHE operational sequences across various scopes to identify common parallel execution patterns. Although our work is not the first to apply kernel fusion to FHE [23, 36], we fully draw on its capability by extensively reordering and splitting the operational sequences, maximizing the fusion opportunities.

Cheddar is a full-fledged FHE library enabling the translation of high-level programmer codes into fully optimized GPU programs. In addition to the aforementioned optimizations, Cheddar features various high-level FHE mechanism implementations, GPU kernel fine-tuning capability, and state-of-the-art FHE algorithms [2, 9, 10, 30, 43].

While we have introduced various distinguished features of Cheddar, the core strength of Cheddar can be summarized rather briefly: **it is simply fast.** We develop highly efficient implementations of practical FHE applications that demonstrate significantly faster execution times than state-of-the-art GPU implementations [18, 23, 25, 36, 54, 64] and even custom FPGA designs [3, 33, 68]. For instance, Cheddar reduces the latency of complex applications like encrypted CNN inference to a practical level on a single GPU.

The main contributions of this work are as follows:

- We propose the 25-30 prime system, a novel 32-bit RNS design with an inverted-terminal data layout, to enable systematic and efficient FHE execution on GPUs.
- We develop highly optimized 32-bit GPU kernels using signed Montgomery reduction and architecture-aware optimizations to enhance computational efficiency and memory usage in core FHE operations.
- We apply kernel fusion extensively, reordering and splitting operational sequences to mitigate memory bandwidth bottleneck.
- Cheddar achieves speedups of 2.18–4.45× over a state-of-the-art GPU library, reducing encrypted ResNet-20 inference latency to 0.72 seconds on a single RTX 5090 GPU.

2 Background

We describe FHE with an emphasis on its computational aspects. Table 1 summarizes key notations and symbols.

2.1 CKKS Fully Homomorphic Encryption (FHE)

FHE supports direct computation on encrypted ciphertexts through cryptographic mechanisms. CKKS [15] supports the encryption of a message $\mathbf{u} \in \mathbb{C}^{N/2}$, a complex vector of length $N/2$. $\mathbf{u}$ is first encoded as $\langle \Delta \cdot \mathbf{u} \rangle \in \mathcal{R}_Q$, a polynomial in an integer ring $\mathcal{R}_Q = \mathbb{Z}_Q[X]/(X^N+1)$ with modulus Q and degree N. **Scale** Δ is multiplied to $\mathbf{u}$ during encoding to cope with errors. Higher Δ leads to higher precision of the outcome [15]. Finally, $\langle \Delta \cdot \mathbf{u} \rangle$ is encrypted into a ciphertext $[\![\langle \Delta \cdot \mathbf{u} \rangle]\!]$ by using a random a, a secret s, and an error e:

$$[\![\langle \Delta \cdot \mathbf{u} \rangle]\!] = (b, a) = (-a \cdot s + \langle \Delta \cdot \mathbf{u} \rangle + e, a) \in \mathcal{R}_Q^2 \qquad (1)$$

A few basic mechanisms of CKKS are introduced. $\odot$ represents element-wise multiplication (mult) and $\lll$ represents cyclic rotation of a vector. An evaluation key (evk) consists of dnum pairs of polynomials in the larger ring $\mathcal{R}_{PQ}$ [30]; i.e., $\text{evk} \in \mathcal{R}_{PQ}^{2\times\text{dnum}}$.

- $\texttt{HAdd}([\![\langle \Delta \cdot \mathbf{u} \rangle]\!], [\![\langle \Delta \cdot \mathbf{v} \rangle]\!]) = [\![\langle \Delta \cdot (\mathbf{u}+\mathbf{v}) \rangle]\!]$.
- $\texttt{PAdd}([\![\langle \Delta \cdot \mathbf{u} \rangle]\!], \langle \Delta \cdot \mathbf{v} \rangle) = [\![\langle \Delta \cdot (\mathbf{u}+\mathbf{v}) \rangle]\!]$.
- $\texttt{HMult}([\![\langle \Delta \cdot \mathbf{u} \rangle]\!], [\![\langle \Delta \cdot \mathbf{v} \rangle]\!], \text{evk}_\odot) = [\![\langle \Delta^2 \cdot (\mathbf{u} \odot \mathbf{v}) \rangle]\!]$.
- $\texttt{PMult}([\![\langle \Delta \cdot \mathbf{u} \rangle]\!], \langle \Delta \cdot \mathbf{v} \rangle) = [\![\langle \Delta^2 \cdot (\mathbf{u} \odot \mathbf{v}) \rangle]\!]$.
- $\texttt{HRot}([\![\langle \Delta \cdot \mathbf{u} \rangle]\!], r, \text{evk}_r) = [\![\langle \Delta \cdot (\mathbf{u} \lll r) \rangle]\!]$.

HMult and HRot need evks for computation, where a separate evk_r is required for every unique HRot distance r.

Residue number system (RNS). To avoid costly big-integer computations required for large modulus ($Q \sim 2^{1200}$) [14] in FHE, RNS is used to handle the large Q efficiently [13]. Q is decomposed into a product of smaller primes that fit within the word size (e.g., 32 or 64 bits): $Q = \prod_{i=0}^{L-1} q_i$, allowing each polynomial in $\mathcal{R}_Q$ to be represented as L residue polynomials:

$$a = (a[0], \cdots, a[L-1]), \text{ where } a[i] = a \bmod q_i.$$

Then polynomial operations are performed limb-wise; e.g.,

$$a \cdot b = (a[0] \cdot b[0] \bmod q_0, \cdots, a[L-1] \cdot b[L-1] \bmod q_{L-1}).$$

Table 1. Notations and symbols.

Notation	Explanation
N	Polynomial ring degree (typically 2^{16}).
$Q \mid q_i \mid \tau_i$	Modulus | main (RNS) primes | terminal (RNS) primes. $Q = \prod_i \tau_i \cdot \prod_i q_i$ for each level (§3).
$P \mid p_i \mid \alpha$	Auxiliary modulus | auxiliary (RNS) primes | the number of p_i's (fixed). $P = \prod_{i=0}^{\alpha-1} p_i$.
$\mathcal{R}_Q$	Cyclotomic polynomial ring $\mathbb{Z}_Q[X]/(X^N + 1)$.
$\langle \Delta \cdot \mathbf{u} \rangle$	Encoding of $\mathbf{u} \in \mathbb{C}^{N/2}$ (not encrypted).
Δ	Scale multiplied to $\mathbf{u}$ during encoding (e.g., 2^{40}).
$[\langle \Delta \cdot \mathbf{u} \rangle]$	Ciphertext ($\in \mathcal{R}_Q^2$) encrypting $\langle \Delta \cdot \mathbf{u} \rangle$.
evk	Evaluation key ($\in \mathcal{R}_{PQ}^{2\times \mathsf{dnum}}$ for dnum [30]).
L_{eff}	Effective level. The level after bootstrapping.

With RNS, a polynomial can be regarded as an $L \times N$ matrix of small integers, each row of which corresponds to a limb. P (used for evks) is handled similarly by setting $P = \prod_{i=0}^{\alpha-1} p_i$.

Rescaling. Multiplicative mechanisms (`HMult` and `PMult`) produce ciphertexts with squared scales (Δ^2). We need to reduce the scale back to Δ for further computation. For that purpose, rescaling, an approximate division by Δ, is performed [15]. For $[\langle \Delta^2 \cdot \mathbf{u} \rangle] \in \mathcal{R}_Q^2$, rescaling reduces the modulus from Q to the next modulus Q/Δ and produces $[\langle \Delta \cdot \mathbf{u} \rangle] \in \mathcal{R}_{Q/\Delta}^2$. For every multiplicative mechanism, this process is repeated with the modulus Q/Δ, Q/Δ^2, and so on.

RNS requires Q/Δ to be also constructed with RNS primes. However, as the RNS primes are discrete, the modulus cannot exactly be reduced to Q/Δ but to $Q' \simeq Q/\Delta$, which results in the scale to be $\Delta' = \Delta^2/(Q/Q')$ [40]. We aim at maintaining $\Delta' \simeq \Delta$, which requires a careful RNS construction (§3).

Level and bootstrapping (`bts`). We associate each modulus with an index called level (e.g., Q_3 for 3 levels), which is equal to the number of rescaling applicable to the current modulus of a ciphertext. At level 0, we have dissipated all the modulus and cannot perform rescaling anymore. `bts` is performed on ciphertexts at level 0 to restore the modulus [14]. `bts` is a complex mechanism including dozens of `HMult` and `HRot` evaluations. The frequency of `bts` is determined by the level after `bts`, which is referred to as **effective level** (L_{eff}) [42]. Maximizing L_{eff} is crucial for FHE performance.

2.2 Ring Polynomial Operations

Only the following four types of polynomial operations are sufficient to comprise CKKS mechanisms [44].

Basis conversion (BConv). BConv is performed to match the modulus between polynomials before computation. For example, we can generate $\hat{a} \in \mathcal{R}_P$ from $a \in \mathcal{R}_Q$ by performing the following for each p_i ($i = 0, 1, \ldots, \alpha - 1$):

$$\hat{a}[i] = \textstyle\sum_{j=0}^{L-1} (a[j] \cdot (Q/q_j)^{-1} \bmod q_j) \cdot (Q/q_j) \bmod p_i. \quad (2)$$

Algorithm 1 Computation of ModSwitch

```
Require: $a[\alpha][N] \in \mathcal{R}_Q$
Ensure: $\hat{a}[L][N] \in \mathcal{R}_P$
1: Initialize $temp[L][N]$ and $\hat{a}[\alpha][N]$ with zero values
2: $temp \leftarrow \mathrm{INTT}(a)$
3: for $j \leftarrow 0$ to $\alpha - 1$ do                          ▷ BConv const mult
4:     $temp[j] \leftarrow temp[j] \cdot (Q/q_j)^{-1} \bmod q_j$
5: for $i \leftarrow 0$ to $\alpha - 1$ do                          ▷ BConv matrix mult
6:     for $j \leftarrow 0$ to $L - 1$ do
7:         $\hat{a}[i] \leftarrow \hat{a}[i] + temp[j] \cdot (Q/q_j) \bmod p_i$
8: $\hat{a} \leftarrow \mathrm{NTT}(\hat{a})$
```

Number-theoretic transform (NTT). Polynomial mult can be expensive as it is equivalent to performing a (negacyclic) convolution between length-N vectors of coefficients, where N is typically as large as 2^{16}. To alleviate its cost, NTT, a Fourier transform variant for $\mathcal{R}_Q$, is utilized. With RNS, NTT can be applied to each limb separately if the q_i's satisfy the following condition:

$$q_i \equiv 1 \pmod{2N}. \quad (3)$$

After NTT is applied to each limb, polynomial mult boils down to element-wise mult between $L \times N$ matrices. Well-known fast Fourier transform (FFT) techniques are used to perform NTT with $O(N \log N)$ complexity for each limb.

Automorphism (ϕ_r). Automorphism is an operation found in `HRot`. For automorphism, we shuffle the columns of a polynomial according to the following one-to-one mapping, which moves the i-th column to the j-th position:

$$j = \phi_r(i) = (((2i + 1) \cdot 5^{-r} \bmod 2N) - 1)/2. \quad (4)$$

We extend the use of the symbol ϕ_r to also represent automorphism on a polynomial a as $\phi_r(a)$.

Element-wise operations. The rest of the operations are all performed in an element-wise manner; e.g., element-wise addition and mult between $L \times N$ matrices (polynomials).

2.3 ModSwitch Routine

Hereafter, we regard that a polynomial is in the NTT-applied form by default. However, prior to performing BConv on a polynomial, it needs to be brought back to the original form by performing inverse NTT (INTT). This creates a common computational routine of INTT → BConv → NTT, which is called ModSwitch [19]. ModSwitch is the most compute-intensive component of CKKS, encompassing nearly all NTT and BConv operations [44].

An exemplar ModSwitch procedure from $\mathcal{R}_Q$ to $\mathcal{R}_P$ is shown in Alg. 1 (ModSwitch can be performed for various input and output moduli pairs, not limited to Q and P). After INTT, BConv is performed in two steps: a per-limb scalar multiplication step (BConv const mult) and a matrix multiplication with precomputed constants (BConv matrix mult). Finally, NTT is performed.

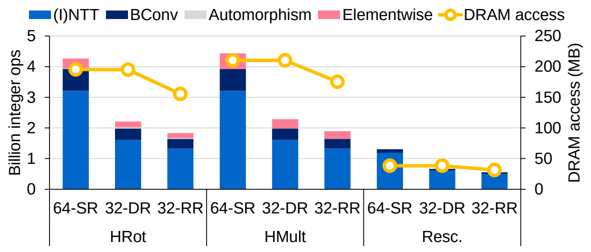


Figure 1. Integer operation counts and DRAM access amounts of `HRot`, `HMult`, and rescaling on an RTX 4090 GPU. We used 11-level parameter sets with different word sizes (64/32) and rescaling methods (SR/DR/RR). $N = 2^{16}$. $\alpha = 6$ (64-bit word) or $\alpha = 12$ (32-bit word) was used.

The mechanisms in §2.1 consist of complex combinations of ModSwitch routine and other operations in §2.2. For example, $\texttt{HRot}([\langle\Delta \cdot \mathbf{u}\rangle], r, \mathsf{evk}_r)$ follows the sequence: ModUp → KeyMult → MAC → automorphism → ModDown. Here, ModUp and ModDown are variants of ModSwitch that change the modulus from Q to PQ and vice versa, respectively. KeyMult is an element-wise operation between the polynomial $a \in \mathcal{R}_{PQ}$ and evk_r. MAC is constant-polynomial mult and add, which is also element-wise operation.

2.4 GPU Execution Model

GPUs are highly parallel computing systems composed of many streaming multiprocessors (SMs), each with numerous int32, fp32, fp64, and tensor cores [22]. Programmers write a parallel function, called a kernel, which is executed by many threads on a GPU. Threads are grouped into thread blocks and further into warps (32 threads), which are scheduled by warp schedulers across the SM sub-partitions.

When a warp is stalled by a memory or data dependency, the scheduler can switch to another ready warp to keep the cores busy. To maintain high utilization, thousands of warps must be launched per kernel. This high degree of thread-level parallelism aligns well with FHE workloads, where independent operations over RNS limbs can be executed in parallel.

3 32-bit Residue Number System (RNS)

As GPUs utilize int32 cores for integer operations, we choose RNS primes smaller than 2^{31}, which is the maximum value enabling easy modular addition with 32-bit integer arithmetic. This 32-bit RNS setting requires up to 2× (linear) more number of limbs to represent a polynomial compared to conventional 64-bit RNS settings used in popular FHE libraries [4, 11, 18, 28, 52]. However, integer operations with larger word sizes require emulation with int32 operations, whose complexity increases quadratically. Overall, as shown in Fig. 1 (64-SR vs. 32-DR), 32-bit RNS requires significantly fewer integer operations for the same mechanism. Also, the linear increase in the number of limbs does not inflate the amount of DRAM access and rather enhances the degree of parallelism, which leads to higher performance due to a more balanced job distribution on GPUs.

3.1 Prior RNS Construction Methods

There are several ways to implement 32-bit RNS based on different rescaling methods. Using an optimized 32-bit RNS construction with a proper rescaling method can further enhance FHE performance by reducing both computation and DRAM access amounts (32-DR vs. 32-RR in Fig. 1). In this section, we use an example of maintaining $\Delta \simeq 2^{40}$, a typical scale value, throughout a sequence of rescaling. We refer to the primes sufficiently close to 2^k as Pr∼k.

Single and double rescaling. To maintain $\Delta \simeq 2^{40}$, the easiest solution is to select the RNS primes comprising Q (q_i's except q_0) as Pr∼40. Then, we can discard one prime for every rescaling, performing an approximate division by $q_i \simeq \Delta$ for each level, which allows us to maintain the scale close to our target 2^{40}. This is called single rescaling (SR).

We may also make the RNS primes satisfy $q_{2i} \cdot q_{2i+1} \simeq 2^{40}$ ($i > 0$) and discard two primes for every rescaling [1, 42], which we call double rescaling (DR). However, double rescaling is not scalable for many levels due to the scarcity of primes that satisfy Eq. 3. We observe that only one prime smaller than 2^{20} satisfies Eq. 3 for $N = 2^{16}$, which means double rescaling can be performed only once for $\Delta \simeq 2^{40}$. Thus, double rescaling is only viable for large scales (e.g., $\Delta \simeq 2^{50}$ in Fig. 2). Both single and double rescaling are not adequate for 32-bit RNS construction with $\Delta \simeq 2^{40}$.

Rational rescaling. BitPacker [61] addresses this problem by adding new RNS primes during rescaling, which is generally referred to as rational rescaling (RR) [12]. An exemplar 32-bit RNS construction with BitPacker is shown in Fig. 2 (RR based on greedy search). First, all the RNS primes are selected as Pr∼30. For the first rescaling, two Pr∼30 (q_5, q_6) are discarded and one Pr∼20 (τ_0) is added, resulting in the scale being adjusted by roughly $2^{-2\cdot 30+20} = 2^{-40}$. For the next rescaling, two Pr∼30 (q_3, q_4) as well as τ_0 are discarded and two new Pr∼20 (τ_1, τ_2) are added ($2^{-2\cdot 30-20+2\cdot 20} = 2^{-40}$). BitPacker greedily searches the most adequate τ_i's, which are referred to as **terminal primes**, for each level to maintain $\Delta \simeq 2^{40}$. RR also allows compact storage of polynomials by composing the modulus with the smallest possible number of RNS primes. From now on, q_i's denote the main Pr∼30 primes and $Q = \prod_i \tau_i \cdot \prod_i q_i$ for each level.

3.2 25-30 Prime System

Evaluation key (evk) compatibility problem. BitPacker keeps adding new primes found by greedy search, which becomes problematic for evk preparation. evks are typically prepared for the top level. When using a top-level evk at lower levels, we may truncate the evk without any computation to adjust its modulus. Fig. 2 shows that single or double

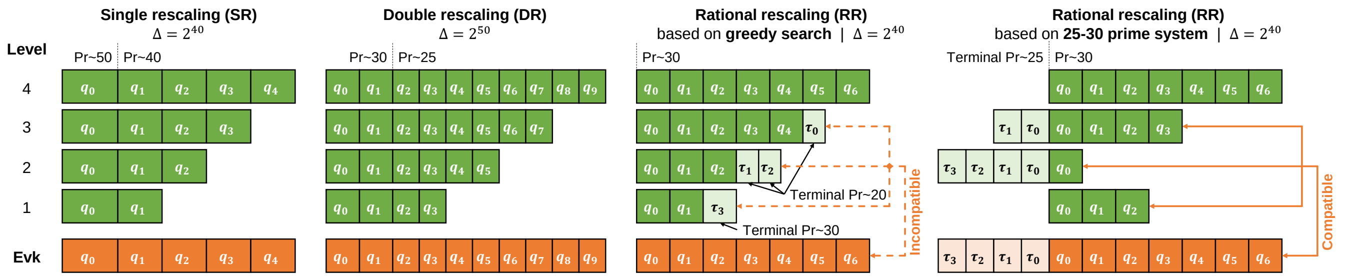


Figure 2. RNS constructions (choice of RNS primes comprising Q) for non-bts levels 1–4 and evk (P-part omitted) based on various rescaling methods. Pr~k represents primes sufficiently close to 2^k.

Table 2. An exemplar 32-bit RNS construction with our 25-30 prime system, where levels 0–4 are non-bts levels ($L_{\text{eff}} = 4$), and the upper levels are reserved for bts.

Level	0	1	2	3	4	5	6	7	8	$\cdots$
# of Pr~25 (τ_i)	2	0	4	2	0	1	2	3	4	$\cdots$
# of Pr~30 (q_i)	0	3	1	4	7	8	9	10	11	$\cdots$
$\log_2 Q$	50	90	130	170	210	265	320	375	430	$\cdots$
$\log_2 \Delta$	-	40	40	40	40	55	55	55	55	$\cdots$

rescaling allows such truncation. However, BitPacker cannot do the same due to the terminal primes, requiring additional measures, such as preparing evks for every possible level. In the typical parameter set, the key size of evk$_\odot$ with BitPacker blows up by approximately five times compared to that of a single top-level key [12].

Grafting [12] addresses this problem by using a power-of-two RNS base (e.g., 2^{30}), which is highly amenable to the truncation, instead of the terminal primes. However, Grafting requires special care of the power-of-two RNS base, which is not adequate for the parallel execution model of GPUs. For example, (I)NTT with an RNS base 2^{30} requires embedding the limb into $\mathcal{R}_{q^*}$ for a large $q^* > N \cdot 2^{60}$, necessitating costly 128-bit execution on GPUs.

25-30 prime system. To overcome the limitations, we introduce 25-30 prime system, our novel 32-bit RNS construction with a systemized rational rescaling approach. Instead of selecting the terminal primes separately for each level, we choose them from a fixed list of Pr~25 (τ_i's). We find a "rescaling cycle" to minimize the number of Pr~25; an exemplar rescaling cycle is shown in Table 2 and Fig. 2. For $\Delta \simeq 2^{40}$, we construct a cycle of 1) discarding three Pr~30 while adding two Pr~25 ($2^{-3\cdot30+2\cdot25} = 2^{-40}$), 2) performing 1) once more at the next level, and 3) discarding four Pr~25 and adding two Pr~30 ($2^{-4\cdot25+2\cdot30} = 2^{-40}$) at the level after next. We can easily find such cycles for typical power-of-2^5 scales (2^{30}–2^{50}) with no more than five Pr~25 used for each case.

The primes used at each level are selected in a fixed order from carefully chosen lists of Pr~30 and Pr~25, which limits the divergence of scales [40] to less than 0.1 bits; the scale stays in between $2^{39.9}$ and $2^{40.1}$ in our $\Delta \simeq 2^{40}$ example. Although our 25-30 prime system does not support an arbitrary scale, the provided choices are fine-grained enough to support the precision required by various data types; otherwise, we can also construct similar prime systems (e.g., 24-30).

We set the modulus for evks as PQ_{max}, where Q_{max} is the product of all the Pr~30 and Pr~25 used (e.g., $Q_{\text{max}} = \prod_{i=0}^{3} \tau_i \cdot \prod_{i=0}^{6} q_i$ in Fig. 2), which makes each evk compatible with all levels. As the rescaling cycle limits the number of terminal primes used, the gap between Q_{max} and the top-level Q value is small. Still, unlike single or double rescaling, we are wasting the modulus budget; security constraints limit PQ_{max} to certain values depending on N (e.g., $PQ_{\text{max}} \leq 2^{881}$ for $N = 2^{15}$) [5]. Therefore, although small, we want to get rid of the gap to allow ciphertexts to use as large modulus as possible, which leads to more available levels.

Double rescaling for bts levels. We remove the modulus gap by selecting primes differently for the upper levels reserved for bts (bts levels). In FHE CKKS, the highest levels are only used during bts. Larger scales, typically around 2^{50}–2^{60} [9], are used at bts levels. We sample the primes for bts levels based on double rescaling, which is adequate for such large scales. While doing so, we ensure that all the RNS primes are eventually included at the final top level. For example, if we are performing bts at levels 5–19 in addition to the construction in Fig. 2, we may force levels 5–8 to use double rescaling with primes $\{\tau_0, q_7\}$, $\{\tau_1, q_8\}$, $\{\tau_2, q_9\}$, $\{\tau_3, q_{10}\}$ and $\Delta \simeq 2^{55}$ (see Table 2). Then, the set of primes used at level 8 or higher will include all the terminal primes. In this way, we can guarantee the top-level ciphertext modulus is equal to PQ_{max}. The actual implementation in Cheddar allows diverse Δ values for bts levels through a complex scale management with non-Pr~30 q_i's and delayed rescaling.

3.3 Inverted-Terminal Data Layout

We prefer to allocate each polynomial in a contiguous and aligned memory address space to simplify index calculation in GPU kernels and to reduce the overhead of memory allocation. The use of rational rescaling complicates this because the limbs corresponding to the same RNS prime can be placed at different memory positions for each level.

Algorithm 2 32-bit signed Montgomery reduction (SMR)

```
Require: $x \in [-q_i \cdot 2^{31}, q_i \cdot 2^{31})$, $q_i < 2^{31}$, $m_i \in [-2^{31}, 2^{31})$,
    $m_i = q_i^{-1} \bmod 2^{32}$          ▷ Constant $m_i$ is precomputed for each $q_i$.
Ensure: $y = x \cdot 2^{-32} \bmod q_i$, $y \in (-q_i, q_i)$
 1: $x = x_{\mathrm{hi}} \cdot 2^{32} + x_{\mathrm{lo}}$, $x_{\mathrm{lo}} \in [0, 2^{32})$          ▷ Bit extraction
 2: $z \leftarrow$ mullo32($x_{\mathrm{lo}}, m_i$)   ▷ 32-bit signed mult (retains lower 32 bits)
 3: $z \leftarrow$ mulhi32($z, q_i$)    ▷ 32-bit signed mult (retains upper 32 bits)
 4: $y \leftarrow x_{\mathrm{hi}} - z$
```

We propose an inverted-terminal data layout to solve this problem. Taking level 3 ($\Delta \simeq 2^{40}$) as an example, at which we utilize four Pr~30 (q_0, q_1, q_2, q_3) and two Pr~25 (τ_0, τ_1), we place the limbs of a polynomial in the order of $[\tau_1, \tau_0, q_0, q_1, q_2, q_3]$ (see Fig. 2). Because we select τ_i's and q_i's in a fixed order (§3.2), we can guarantee that polynomial data are allocated contiguously in memory. The simplified index calculation from our inverter-terminal data layout is especially beneficial for computations between polynomials at different levels, such as in rescaling (see Fig. 4).

4 Operation-Level Optimizations

4.1 32-bit Modular Arithmetic

We implement signed Montgomery reduction (SMR) [62] for GPUs. Alg. 2 shows its computation process. While prior studies utilized classical reduction methods from Barrett [7], Montgomery [51], and Shoup [66], Table 3 shows that SMR is cheaper than the classical methods. SMR includes fewer instructions than Barrett or Montgomery reduction. Shoup reduction requires slightly less computation than SMR but Shoup reduction is restricted to constant mult and requires extra memory access for precomputed constants. Therefore, we use SMR for all modular mult in Cheddar.

Potential drawbacks of using SMR are additional computation requirements to convert to or from the Montgomery form; in the Montgomery form, each data element x of a polynomial is stored as $x \cdot 2^{32} \bmod q_i$. We completely eliminate this overhead by kernel fusion in §5.1.

4.2 Kernel Optimizations

The roofline plot in Fig. 3a summarizes the computational characteristics of polynomial operations comprising `HRot`. Automorphism and element-wise operations involve little amount of computation and are bottlenecked by limited GPU memory (DRAM or L2 cache) bandwidth. Their inherently simple structure hinders further kernel-level optimization. Meanwhile, on RTX 4090, BConv is mostly compute-bound and (I)NTT is mostly memory-bound. We devise optimization methods addressing these characteristics.

BConv matrix mult with lazy reduction. We implement BConv matrix mult in Alg. 1 with a well-known blocked matrix-matrix mult method [39]. However, unlike regular

Table 3. Comparison of modular reduction methods. On recent GPU architectures, the computational cost is in the order of mulwide ≥ mulhi > mullo. We ignore signedness. # of const refers to the number of precomputed constants required per RNS prime for modular reduction.

Reduction method	Computation requirements	# of const	Output range
Barrett	mulhi64 + mullo64 + add64	1	$[0, 2q_i)$
Montgomery	mulwide32† + mullo32 + add64	1	$[0, 2q_i)$
Shoup	2 * mullo32 + add32	Many‡	$[0, 2q_i)$
Signed Mont.	mulhi32 + mullo32 + add32	1	$(-q_i, q_i)$

† mulwide32 multiplies two 32-bit integers and produces a 64-bit result.
‡ Shoup reduction is used only for constant mult, and requires an additional precomputed constant for every unique constant to multiply.

matrix-matrix mult, BConv matrix mult includes modular reductions, which introduce significant compute overhead.

To reduce the number of modular reductions, we delay the reduction, which is a widely used technique (e.g., 100× [36]) referred to as lazy reduction. However, lazy reduction often requires restricting the range of RNS primes (e.g., $q_i \in [0, 2^{30})$ [31]) to exploit extra bits in a word.

We discover that, even with the minimal prime restriction ($q_i \in [0, 2^{31})$) in Cheddar, SMR allows sufficiently lazy reduction for BConv. We prepare $temp[j]$ and $(Q/q_j)^{-1} \bmod p_i$ (see Alg. 1) respectively in the ranges of $(-q_j/2, q_j/2) \in (-2^{30}, 2^{30})$ and $(-p_i/2, p_i/2)$, whose product will be in the range of $(-2^{29} \cdot p_i, 2^{29} \cdot p_i) \in (-2^{60}, 2^{60})$. Thus, we can accumulate up to eight products, which will be in the range of $(-2^{32} \cdot p_i, 2^{32} \cdot p_i)$, without an overflow for a signed 64-bit integer. For more accumulation, we add or subtract $2^{32} \cdot p_i$ to adjust the range to $(-2^{31} \cdot p_i, 2^{31} \cdot p_i)$, after which we can accumulate four more products. We repeat this until we accumulate all the products. Then, we use SMR for the final reduction. For regular unsigned Montgomery reduction, the product will be in a twice wider range $[0, 2^{31} \cdot p_i)$, requiring the range adjustment to be performed twice as much.

(I)NTT with extended on-the-fly twiddle factor generation (EOT). We use the standard radix-2 Cooley-Tukey [17] FFT, which involves $\log_2 N$ stages of butterfly operations, to implement (I)NTT kernels. We partition the stages into a small number of phases, where synchronization between threads are required in between the phases. At each stage, data are multiplied by constants named twiddle factors. As each limb requires a unique set of N twiddle factors, whose total size is the same as the limb itself, twiddle factors become significant sources of memory access.

We can use on-the-fly twiddle factor generation (OT) [45] to reduce the memory burden. OT generates a twiddle factor $\overline{W} = W^k$ during runtime by computing $\overline{W} = \overline{W}_{\mathrm{lo}} \cdot \overline{W}_{\mathrm{hi}}$, where $\overline{W}_{\mathrm{lo}} = W^{\mathrm{lsb}(k)}$, $\overline{W}_{\mathrm{hi}} = W^{k-\mathrm{lsb}(k)}$, and lsb(·) is a function extracting a number of lower bits from an integer.

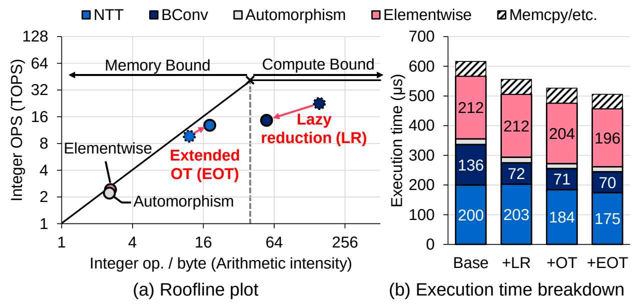


Figure 3. (a) Roofline plot and (b) execution time breakdown of HRot with incremental optimizations—lazy reduction (LR), on-the-fly twiddle factor generation (OT) [45], and extended OT (EOT). HRot is performed for a ciphertext with 48 limbs on an RTX 4090 GPU. $N = 2^{16}$. $\alpha = 12$.

While Kim et al. [45] only used OT for the last stage of each phase, we propose extended OT (EOT), which iteratively generates all the twiddle factors for every stage in a phase. We give an example for processing three stages in a phase. Each thread needs twiddle factors $\{W^{\text{brev}(i)}\}$ for the first stage, $\{W^{\text{brev}(2\cdot i)}, W^{\text{brev}(2\cdot i+1)}\}$ for the second, and $\{W^{\text{brev}(4\cdot i)}, W^{\text{brev}(4\cdot i+1)}, W^{\text{brev}(4\cdot i+2)}, W^{\text{brev}(4\cdot i+3)}\}$ for the last, where brev refers to bit-reversal. We first generate the last four twiddle factors by OT. Then, EOT iteratively generates the twiddle factors for the preceding stages as follows:

$$W^{\text{brev}(2\cdot i)} \leftarrow (W^{\text{brev}(4\cdot i)})^2 \ \&\ W^{\text{brev}(2\cdot i+1)} \leftarrow (W^{\text{brev}(4\cdot i+2)})^2,$$
$$W^{\text{brev}(i)} \leftarrow (W^{\text{brev}(2\cdot i)})^2.$$

Optimization effects. Fig. 3b shows the performance improvements achieved for HRot by applying the optimizations above. Our SMR-based lazy reduction (+LR) eliminates 65% of the integer operations in BConv matrix mult kernels, which leads to a 1.89× speedup of BConv in HRot. Also, EOT reduces the amount of twiddle factor load by 96% and requires less cache space, which also indirectly enhances the performance of other memory-bound kernels. Overall, EOT achieves a 1.16× speedup of NTT for HRot, compared to the 1.10× speedup of regular OT [45].

Fig. 3a shows that the lazy reduction and EOT respectively make BConv and (I)NTT kernels more balanced in terms of arithmetic intensity on RTX 4090. However, different GPU architectures create different trade-offs, possibly demanding alternative approaches to implement the kernels.

Fine-tuning. Thus, we implement the kernels in a highly parameterized manner such that we can easily modify how much job each thread will take charge, how the (I)NTT stages will be partitioned into multiple phases, to what extent EOT will be applied, what block sizes will be used for the BConv matrix mult kernel based on the blocked matrix-matrix mult method [39], and so on. We fine-tune the kernels for each GPU architecture using our software fine-tuner (§6).

5 Operational Sequence Optimization

Our optimized 32-bit implementation effectively accelerates compute-intensive BConv and (I)NTT operations. However, element-wise operations and automorphism, which have low arithmetic intensity and are bound by memory bandwidth, do not get improved simply by operation-level optimizations. They become serious sources of performance bottlenecks (see Fig. 3b) with the aforementioned optimizations reducing the significance of (I)NTT and BConv.

We thoroughly analyze common operational sequences in FHE and find opportunities to reduce the memory overhead induced by these operations. Our main tool is *kernel fusion*. we discovered new fusion opportunities by analyzing the modified operation sequences that arise from rational rescaling with our 25-30 prime system and signed Montgomery reduction. Additionally, we apply fusion across a wider range of degrees rather than just a few basic sequences. Fusion techniques can be categorized into two types: sequential fusion and parallel fusion. To clearly distinguish between these two categories, we use Seq.X and Par.X notations.

Prior GPU implementations of FHE, including 100× [36] and WarpDrive [23], have also utilized kernel fusion but in a limited form. 100× fuses BConv const mult in ModSwitch (lines 3–4 in Alg. 1) with the preceding INTT (Seq.1). Additionally, 100× implements KeyMult (Par.1), and batched PMult (Par.2) in single kernels, respectively. WarpDrive further applied fusion to batching independent polynomial operations (Par.3). Cheddar incorporates all of these existing fusion techniques and extends them.

5.1 Sequential Fusion

We first find opportunities at the level of basic FHE mechanisms, such as HMult, HRot, and rescaling. We discover methods to fuse kernels for element-wise operations or automorphism forming a sequential dataflow in these mechanisms. We use an example of rescaling from level 3 to level 2 when $\Delta = 2^{40}$ (see Fig. 4).

For rescaling, we first multiply $\tau_3\tau_2$, which are primes that are used at level 2 but not at level 3, to the polynomial to temporarily increase the modulus. Then, we want to divide it by $q_1q_2q_3$, but we first need to zero-out the residues corresponding to these primes. Thus, we perform ModSwitch from $q_1q_2q_3$ to $\tau_3\tau_2\tau_1\tau_1q_0$ and subtract the result to the polynomial, which allows us to finally multiply $q_1^{-1}q_2^{-1}q_3^{-1}$ for the division. Fig. 4 shows that this process requires a lot of constant mults. In addition to the aforementioned constants, we also multiply constants for the transitions to/from the Montgomery form $(2^{32}, 2^{-32})$ and the BConv const.

We accelerate this sequence by fusing constant mults into adjacent INTT and BConv kernels (Seq.2). Our approach is distinct from conventional kernel fusion approaches [36] in that the fused INTT or BConv kernels do not have any additional operations due to the precomputation of the constants. INTT kernels already include constant mult by $N^{-1}\,\cdot$

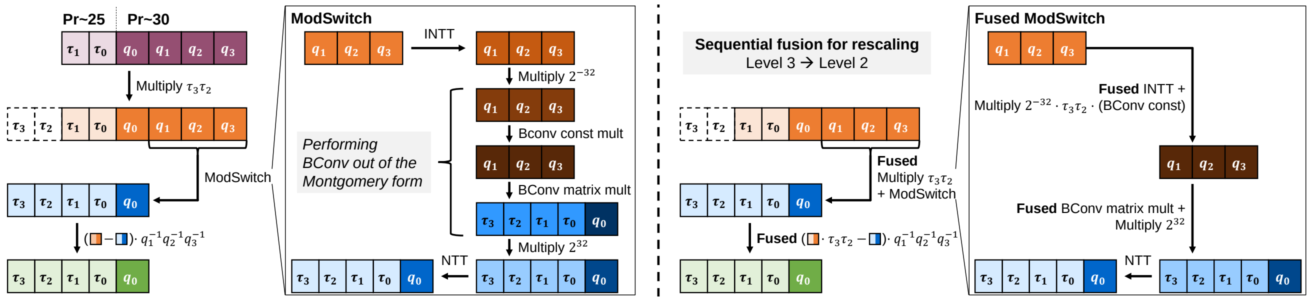


Figure 4. Operational sequence of rescaling from level 3 to level 2 ($\Delta = 2^{40}$) without (left) and with sequential fusion (right). Each block represents a limb of a polynomial. Each arrow represents an operation implemented with one or two GPU kernels.

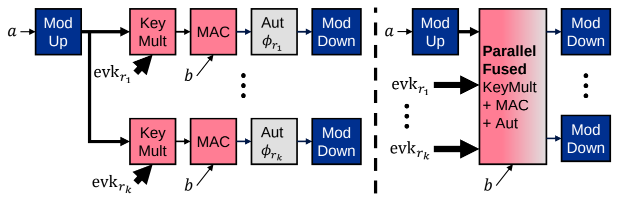


Figure 5. Part of FHE linear transform operational sequence without (left) and with (right) parallel fusion. Blocks represent operations (red: element-wise, gray: automorphism, blue: ModSwitch variant). Arrows show dataflow and arrow thickness represents the size of data.

2^{32} (Montgomery form) at the end; we can instead prepare $N^{-1} \cdot \tau_3\tau_2 \cdot$ (BConv const) to perform the fusion. Also, we can multiply 2^{32} to the BConv matrix in advance. The use of precomputed constants reduces overall computation, notably eliminating the cost of Montgomery conversion entirely.

Also, we split (fiss) and reorder operations [35] to find more fusion opportunities (Seq.3). For example, we split $\tau_3\tau_2$ mult into two: one for $\{\tau_1, \tau_0, q_0\}$-limbs and the other for $\{q_1, q_2, q_3\}$-limbs. We reorder the operational sequence to merge the former with the last kernel of rescaling and the latter with INTT, as illustrated in Fig. 4.

We apply such techniques for sequential fusion to various CKKS mechanisms. For example, `HMult` and `HRot` also include a similar operational sequence (ModDown [30]), where we can perform the same optimization.

5.2 Parallel Fusion

The most basic form of parallel fusion is batching. We apply batching to multiple polynomial-wise operations, similar to Par.3 of WarpDrive. For example, we merge two or three polynomial addition kernels into a single kernel for the `HAdd` mechanism. TensorFHE [25] goes even further by batching multiple mechanism evaluations, which is possible when the server can batch requests from multiple queries. However, as we find this server operation model limited to few use cases, considering the currently high latency of FHE workloads, we focus on the basic single-query model. Also, batching in itself does not reduce memory access without shared data.

Next, we apply parallel fusion for sequences that involve accumulations (Par.4). Applying parallel fusion for such sequences can significantly reduce memory access as we remove the need to store or load temporary accumulation results. Many FHE mechanisms, ranging from basic `HMult`/`HRot` to high-level polynomial evaluations, have such sequences. Especially, by reordering the operational sequences, we discover sequences that accumulate multiple automorphism results, such as $\phi_{r_1}(a_1) + \cdots + \phi_{r_k}(a_k)$. We implement parallel fusion for this sequence by using the inverse of Eq. 4 (ϕ_r^{-1}) to compute this in a single kernel. Each thread can efficiently accumulate multiple automorphism results into a single register as

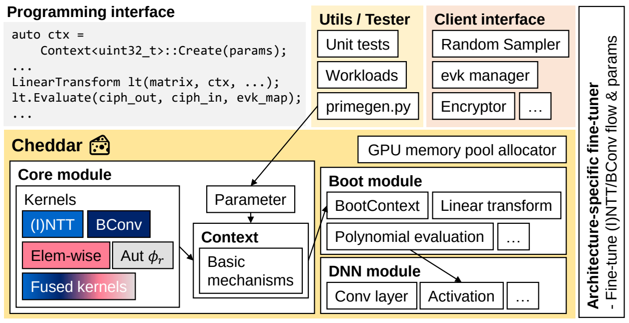
$$a_{\text{res}}[j][i] = a_1[j][\phi_{r_1}^{-1}(i)] + \cdots + a_k[j][\phi_{r_k}^{-1}(i)]. \quad (5)$$

We also fuse sequences that compute multiple accumulation results with the same input, which are prevalent in `bts`.

Finally, we apply parallel fusion for high-level linear transform [9] and other DNN mechanisms [38, 47, 56] (Par.5). Linear transforms require a number of parallel `HRot` operations, and [9] proposed methods to reuse common computational results within these sequences. As an example, we show an optimized linear transform sequence in Fig. 5. In this case, rather than simply batching operations in parallel, we focus on reducing redundant memory access for shared inputs. We merge multiple parallel element-wise (KeyMult and MAC) and automorphism operations into a single kernel, significantly reducing memory access to shared inputs such as ModUp(a) and b.

6 Implementation

Fig. 6 shows the overall software structure of Cheddar. The core module of Cheddar contains the GPU kernels and implementations of the basic mechanisms. `bts` and DNN modules, which are optional, provide implementations of various common high-level mechanisms, such as linear transform, arbitrary polynomial evaluation, and convolutional layers. The aforementioned kernel and operational sequence optimizations are fully applied to each mechanism. The main Cheddar library is implemented in 11,000+ lines of C++/CUDA code.



```
auto ctx =
    Context<uint32_t>::Create(params);
...
LinearTransform lt(matrix, ctx, ...);
lt.Evaluate(ciph_out, ciph_in, evk_map);
...
```

Figure 6. Software implementation of Cheddar.

Clients can easily utilize the library with our high-level C++ programming interface with minimal low-level knowledge.

We provide an architecture-specific fine-tuner to determine the dataflow and parameters for the highly parameterized GPU kernels discussed in §4.2. The fine-tuner performs a comprehensive design space exploration to find the best configuration for each GPU architecture. We also provide various utilities, such as a script (primegen.py) for our 32-bit RNS construction (§3.2), and a test-purpose client interface.

Cheddar incorporates state-of-the-art FHE algorithms. We used fast high-precision bts algorithms [9, 10, 30] and extended the ModDown merging optimization from MAD [2] to support our prime system. Also, we removed a portion of HRot evaluations during bts by the pre-rotation removal technique in ARK [43]. We further devised a method to modify the linear transform matrices in bts to reduce the amount of HConj (mechanism similar to HRot used for conjugation) evaluations. We also applied workload-specific optimizations that have profound performance implications. For example, we merged bts for $[\langle \Delta \cdot \mathbf{u} \rangle]$ and $[\langle \Delta \cdot \mathbf{v} \rangle]$ encrypting real-valued $\mathbf{u}$ and $\mathbf{v}$ into a single bts of $[\langle \Delta \cdot (\mathbf{u} + \sqrt{-1} \cdot \mathbf{v}) \rangle]$, which halves the bts cost for our HELR implementation in §7.

7 Evaluation

7.1 Experimental Setup

By default, we used 128-bit secure parameters with $N = 2^{16}$, $PQ < 2^{1776}$, and $\Delta = 2^{40}$. We set dnum [30] as four. For evaluation, we implemented the following representative FHE CKKS workloads widely used in prior work.

- **bts**: Full-slot bootstrapping of a ciphertext encrypting a length-2^{15} complex vector. $L_{\mathrm{eff}} = 13$ for $\Delta = 2^{40}$. We used higher scales of $\Delta = 2^{55}$–2^{58} during bts.
- **HELR** [29]: Logistic regression binary classification model training. We performed 32 training iterations, each with a 1024-batch of 14×14 grayscale images. We use execution time per iteration (ms/it) for comparison.
- **ResNet** [47]: ResNet-20 CNN inference on a single CIFAR-10 image of size $32 \times 32 \times 3$. Each ReLU uses 14 levels.
- **Sort** [32]: Two-way sorting network with 2^{14} numbers.

We used various GPUs in Table 4 to compare Cheddar with prior state-of-the-art GPU implementations on the same GPU device: V100, A100 40GB, or A100 80GB. A100 80GB has 1.24× higher DRAM bandwidth compared to A100 40GB as well as having larger DRAM capacity. For the comparison with AMD MI100 GPU, we used A100 40GB for Cheddar as it shows similar compute and memory characteristics. We also tested Cheddar on more recent GPU devices, including a server GPU (H100) and a consumer GPU (RTX 5090).

Table 4. Peak 32-bit integer math throughput and DRAM configuration of GPUs based on official specifications. We used the six GPUs listed below for the evaluation of Cheddar.

GPU	Int32 throughput	DRAM (capacity, bandwidth)
V100	14.1 TOPS	HBM2 (16GB, 897GB/s)
A100 40GB	19.5 TOPS	HBM2 (40GB, 1555GB/s)
A100 80GB	19.5 TOPS	HBM2E (80GB, 1935GB/s)
H100	25.6 TOPS	HBM2E (80GB, 2039GB/s)
RTX 4090	41.3 TOPS	GDDR6X (24GB, 1008GB/s)
RTX 5090	104.8 TOPS	GDDR7 (32GB, 1792GB/s)
MI100	23.1 TOPS	HBM2 (32GB, 1229GB/s)

Cheddar also supports 64-bit execution through a polymorphic code design. Further, we can use various RNS constructions using single or double rescaling, other than rational rescaling with our 25-30 prime system. We use those suboptimal configurations of Cheddar for ablation studies.

FHE workloads require large memory capacity to hold various evks, ciphertexts, and other polynomials. Due to the out-of-memory (OoM) errors, we could not run some workloads on V100 or RTX 4090 with relatively small memories.

For aggressive parameter sets with $\Delta \leq 2^{35}$, we use a sparse secret ($h = 256$) [16] for ResNet and HELR, which still ensures 128-bit security.

7.2 FHE Workload Performance

Cheddar achieves speedups ranging from 2.18× to 19.6× for the workloads compared to the state-of-the-art GPU implementations [18, 23, 25, 36, 54, 64]. In particular, Cheddar shows 2.18–4.45× improved performance compared to Warp-Drive [23], which is one of the latest GPU implementations, when comparing the two on the same A100 80GB GPU.

Cheddar also outperforms HEaaN-GPU [18, 54], achieving 6.50× faster ResNet inference time of 1.32 seconds. Meanwhile, HEaaN-GPU also reported an inference time of 1.40 seconds by using a modified FHE CNN implementation based on HyPHEN [41] and AESPA [55], which replaces the 14-level ReLU with a 1-level activation. Such direct workload enhancements can also benefit Cheddar but we did not adopt them for fair comparison with other GPU implementations.

Using Cheddar with more recent NVIDIA GPUs, such as the H100, RTX 4090, and the latest RTX 5090, resulted in even faster execution of the workloads. Compared to the A100 80GB where bts takes 40ms, the H100 and RTX 4090 reduce the time to around 31ms, and this is further accelerated to

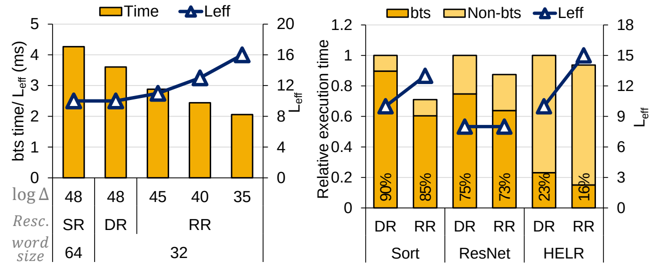
Table 5. Execution time of FHE CKKS workloads using Cheddar compared to 100× [36], TensorFHE [25], HEaaN-GPU [18, 54], WarpDrive [23], GME [64], FAB [3], Poseidon [68], and EFFACT [33]. OoM refers to execution failure due to the out-of-memory error.

Implementation (Hardware)	bts (ms)	HELR† (ms/it)	ResNet (s)
100× (V100)	328	775	-
vs. Cheddar (V100)	4.44×	9.74×	
TensorFHE (A100 40GB)*	250	1007	4.94
vs. Cheddar (A100 40GB)	5.88×	19.6×	3.63×
GME-base (MI100)	413	658	9.99
vs. Cheddar (A100 40GB)	9.72×	12.8×	7.35×
HEaaN-GPU (A100 80GB)	171	-	8.58
vs. Cheddar (A100 80GB)	4.28×		6.50×
WarpDrive (A100 80GB)	121	113	5.88
vs. Cheddar (A100 80GB)	3.03×	2.18×	4.45×
Cheddar (V100)	73.8	79.6	OoM
Cheddar (A100 40GB)	42.5	51.4	1.36
Cheddar (A100 80GB)	40.0	51.9	1.32
Cheddar (H100)	31.2	40.7	1.05
Cheddar (RTX 4090)	31.6	29.9	OoM
Cheddar (RTX 5090)	22.1	25.9	0.72
Cheddar (H100, $\Delta = 2^{35}$)	28.7	39.3	0.82
Cheddar (RTX 5090, $\Delta = 2^{35}$)	19.8	25.5	0.57
FAB (FPGA)	477	103	-
Poseidon (FPGA)	128	72.9	2.66
EFFACT (FPGA)	148	64.6	2.18
GME (Modified MI100)	33.6	54.5	0.98

* TensorFHE batched workload evaluations on multiple input ciphertexts. We divided the execution time with the batch size in favor of TensorFHE.
† HELR includes merged bts. Excluding this optimization shows 57.4 ms/it on A100 80GB, reducing the speedup of Cheddar over WarpDrive from 2.18× to 1.96×.

22.1ms on an RTX 5090; it is faster to perform bts than to receive a new 5MB ciphertext through a 1Gbps network (> 40ms). Further, with an H100 GPU, the encrypted ResNet inference latency is reduced to a practical level of 1.05s, or 0.82s if we opt for a more aggressive parameter choice with $\Delta = 2^{35}$. On an RTX 5090, this latency improves to 0.72s and 0.57s, respectively, because its int32 throughput of 104.8 TOPS is over 4× higher than the H100's 25.6 TOPS, despite its lower DRAM bandwidth (see Table 6).

Cheddar also demonstrates superior performance compared to custom hardware designs based on FPGA [3, 33, 68]. Cheddar proves to be a cost-effective solution for FHE acceleration given that using Cheddar with a single RTX 4090 GPU ($1,600) or RTX 5090 GPU ($2,000) is substantially faster than the FPGA implementations, which used expensive Alveo U280 FPGAs ($6,000). Cheddar also achieves similar performance to GME [64], which added 186mm^2 worth of specialized computational components into an AMD MI100 GPU to turn it into an FHE accelerator.



(a) (bts time)/L_{eff} & L_{eff} **(b)** bts portion & L_{eff}

Figure 7. (a) bts time divided by L_{eff} and (b) relative execution time with bts portion in the workloads depending on the RNS construction using single rescaling (SR), double rescaling (DR), or rational rescaling (RR) on H100. DR uses $\Delta = 2^{48}$ and RR uses $\Delta = 2^{40}$. Higher scales ($\Delta = 2^{55}$–2^{58}) are used for bts levels regardless of the configuration.

Table 6. Execution time (H100) and functionality of HELR, ResNet, and Sort depending on Δ. Higher scales ($\Delta = 2^{55}$–2^{58}) are used for the bts levels regardless of the configuration.

	HELR		ResNet		Sort	
	Time (ms/it)	Acc. (%)	Time (s)	Acc. (%)	Time (s)	Prec.† (bits)
$\Delta = 2^{48}$ (DR)	44.3	96.42	1.20	91.96	14.83	12.84
$\Delta = 2^{40}$ (RR)	40.7	96.37	1.05	91.95	10.20	12.08
$\Delta = 2^{35}$ (RR)	39.3	96.12	0.82	91.88	9.51	11.90
$\Delta = 2^{30}$ (SR)	34.4	52.21	0.68	9.59	8.49	9.17
Unencrypted*	3.5	96.37	0.003	92.52	0.45	∞

† Average $\log_2 \epsilon^{-1}$ (bits). $\epsilon = ||\mathbf{u}^* - \mathbf{u}||_\infty$ for the true sorted vector $\mathbf{u}^*$ and the decrypted FHE Sort evaluation result $\mathbf{u}$.
* Unencrypted execution times are measured on a single-threaded AMD EPYC 7452. Sort is implemented using bitonic sorting network.

7.3 Efficiency of 25-30 Prime System

By using rational rescaling (RR) with our 25-30 prime system, we can utilize a parameter set that provides a fitting precision value tailored to each workload by selecting proper scales (Δ). We can generally get a greater L_{eff} (§2.1) by using smaller scales with rational rescaling, which is not possible for naïve 32-bit execution based on double rescaling (DR). A greater L_{eff} leads to less frequent bts invocations and performance improvements, represented by the reduction in bts time divided by L_{eff} as we use smaller scales (see Fig. 7a). For the workloads, Fig. 7b shows that our default $\Delta = 2^{40}$ (RR) setting increases L_{eff} by up to five levels, reduces the portion of bts in the workloads by 2–8%, and overall results in 1.07–1.41× speedups, compared to our hand-tuned $\Delta = 2^{48}$ (DR) setting.

However, precautions are needed as reducing the scale may compromise the functionality of the workloads. We tested the functionality of HELR, ResNet, and Sort while reducing the scale down to $\Delta = 2^{30}$ (see Table 6). At $\Delta =$

Table 7. Median execution times (RTX 4090) of `HMult`, `HRot`, `HAdd`, and rescaling when using Cheddar or open-source GPU libraries [20, 53, 67]. For the latter using 64-bit RNS, we halved the number of limbs and α for fair comparison.

Library	limbs	α	Execution time (μs)			
			`HMult`	`HRot`	`HAdd`	Rescale
Liberate.FHE [20]	24	6	5989	4554	96	150
HEonGPU [53]	24	6	2427	2279	79	255
Phantom [67]	24	6	958	850	83	114
Cheddar	48	12	533	476	48	68
Liberate.FHE [20]	12	6	2848	2052	22	120
HEonGPU [53]	12	6	825	740	10	113
Phantom [67]	12	6	380	336	18	68
Cheddar	24	12	222	191	11	47

2^{30}, HELR and ResNet stop functioning correctly. At $\Delta = 2^{35}$, although the accuracy values slightly drop, HELR and ResNet are functional. Sort functions correctly with reduced precision for such small scales. Compared to the naïve $\Delta = 2^{48}$ (DR) setting, 1.13–1.56× speedups can be achieved for these workloads with the aggressive $\Delta = 2^{35}$ (RR) setting. We used `dnum` = 6 [30] for Sort with $\Delta = 2^{48}$ and ResNet with $\Delta \geq 2^{40}$ to abide by the 128-bit security requirements.

Finally, Fig. 7a also shows that even for equivalent settings (48-SR vs. 48-DR), 32-bit execution is much more efficient than 64-bit execution, reducing the `bts` time by 1.18×.

7.4 FHE Mechanism Performance

We used three open-source GPU FHE libraries, Phantom [67], HEonGPU [53], and Liberate.FHE [20], to compare the performance of basic mechanisms with Cheddar. By adjusting the number of limbs and α, we made sure all the libraries operate with the same amount of input data, excluding the performance gains stemming from our 25-30 prime system. Table 7 shows the results on an RTX 4090 GPU.

For the 48/24-limb case, Cheddar outperforms the fastest open-source library, Phantom [67], achieving 1.71× faster HMult, 1.76× faster HRot, and 1.64× faster rescaling. Similarly, for the 24/12-limb case, Cheddar achieves 1.73×, 1.76×, and 1.53× improvements, respectively. As the benefits of our 25-30 prime system are excluded, these gains primarily stem from 32-bit execution and the kernel optimizations (§4) that reduce the computational overhead. While our kernel fusion methods also contribute, their impact is limited for these basic mechanisms with straightforward execution sequences. For `HAdd`, which consists solely of element-wise operations, performance is constrained by memory bandwidth, resulting in minimal variation across the libraries.

7.5 Ablation Study

We conducted an ablation study to evaluate the workload-level impact of our proposed kernel optimizations and fusion techniques. Our baseline already incorporates several techniques from prior work, including signed Montgomery reduction (SMR), rational rescaling (RR), and specific fusion methods (Seq.1, Par.1–3).

Table 8. Incremental optimization configurations used in ablation study (Fig. 8). SMR and RR denote signed Montgomery reduction and rational rescaling, while EOT and LR represent extended on-the-fly twiddle factor generation and lazy reduction. Seq.X and Par.X represent sequential and parallel fusion optimizations detailed in §5.

Config.	Optimizations
Base	SMR, RR, Seq.1, Par.1–3
+KerOpt	+EOT, +LR
+SeqFuse	+Seq.2,3
+ParFuse	+Par.4,5

We evaluated the workload performance of Cheddar while incrementally applying three types of optimizations. First, we applied kernel optimizations (+KerOpt), which include the extended on-the-fly twiddle factor generation for NTT and lazy reduction for the BConv kernel. Second, we added sequential fusion (+SeqFuse), which includes the fusion for Montgomery conversion (Seq.2) and rescaling (or ModDown) (Seq.3) in §5.1. Finally, we incorporated parallel fusion (+ParFuse), which includes the fusion for automorphism (Par.4) and linear transforms (Par.5) in §5.2. These optimization configurations are summarized in Table 8.

Fig. 8 shows the performance impact of our kernel optimizations and fusion techniques. Kernel optimizations alone provide modest improvements of 5–7% in average workload performance. These improvements are limited because our kernel optimizations primarily target NTT and BConv operations, while element-wise operations constitute the dominant portion of execution time at the workload level. Element-wise operations account for 53% of `bts` time on an H100 and even 68% on an RTX 4090, which features lower memory bandwidth and higher computational throughput.

The dominance of element-wise operations makes our fusion techniques particularly effective. As sequential fusion reduces the memory accesses for basic mechanisms, including rescaling, `HMult`, and `HRot`, it leads to additional 18–22% average workload performance improvements across the four GPUs. Furthermore, as `bts` includes linear transforms, our parallel fusion optimizations tested here benefit all FHE workloads, leading to additional 12–23% average workload performance improvements.

Overall, our kernel optimizations and fusion techniques achieve average workload speedups of 1.38–1.41× across the four GPUs.

8 Related Work & Discussion

Although abundant GPU studies attempted to accelerate basic CKKS mechanisms [20, 24, 37, 53, 63, 67, 69], only

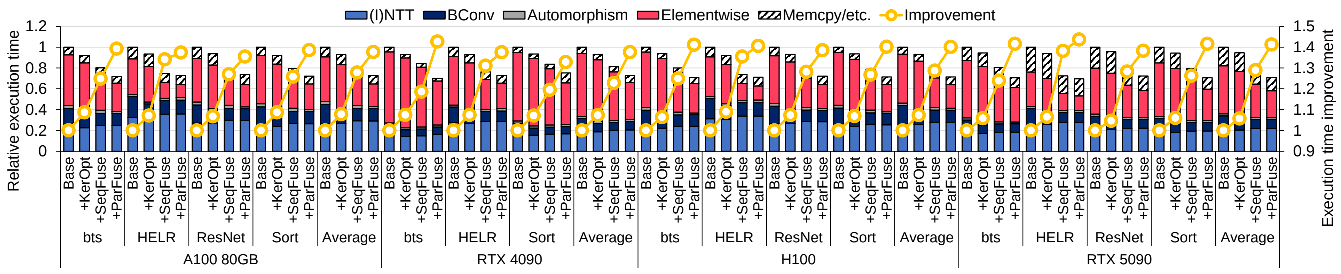


Figure 8. Performance and execution time breakdown of bts, HELR, ResNet, Sort, and their average. The incremental optimization configurations (Base, +KerOpt, +SeqFuse, and +ParFuse) are defined in Table 8.

few [18, 23, 25, 36, 54, 64] fully implemented bts to demonstrate their applicability to FHE CKKS. 100× [36] identified that element-wise operations become the bottleneck in bts and developed basic kernel fusion techniques for them. Park et al. [54] used the HEaaN-GPU library [18] to implement CNN workloads. GME [64] developed a custom hardware solution for FHE on top of the AMD MI100 GPU architecture.

Although Cheddar demonstrates superior performance compared to prior studies, we may adopt orthogonal optimizations from them for further acceleration. For example, we may utilize the tensor cores in recent NVIDIA GPU architectures. TensorFHE [25] utilized the 8-bit integer datapath within the tensor cores to emulate 32-bit integer operations for (I)NTT. WarpDrive [23] improved upon TensorFHE by decomposing the (I)NTT matrix in a more fine-grained manner and utilizing both the int32 cores and the tensor cores.

Other studies [21, 27, 45, 50, 65] also optimized (I)NTT on GPUs. Kim et al. [45] addressed the memory bandwidth bottleneck in (I)NTT through OT (§4.2). Shivdika et al. [65] proposed a modified modular reduction adequate for GPUs. Goey et al. [27] utilized the warp shuffle instructions to exchange the twiddle factors between threads. Durrani et al. [21] used warp shuffle instead for exchanging polynomial data elements, addressing bank conflict issues in shared memory. Meanwhile, Lee et al. [50] implemented the Nussbaumer algorithm on GPUs and compared it with (I)NTT.

Prior studies also accelerated CKKS with custom hardware designs using FPGA [3, 33, 57, 58, 68, 70] or ASIC [42–44, 46, 59, 60]. Apart from ASIC designs, which still require more research and development for realization, FPGA has recently become a popular platform due to its high flexibility and programmability. FAB [3], Poseidon [68], and EF-FACT [33] are notable FPGA implementations that support FHE CKKS with bts. However, due to the low hardware operating frequency and resource limitations, they reported modest performance values compared to Cheddar.

9 Conclusion

To resolve the high computational and memory overhead of FHE, we have developed Cheddar, a swift GPU library for end-to-end execution of FHE workloads. Cheddar features a novel systemized RNS construction method named 25-30 prime system, architecture-aware 32-bit kernel optimizations, and optimized operational sequence implementations based on extensive kernel fusion; these features are provided as a full-fledged GPU library with a high-level interface. Cheddar shows 2.18–4.45× faster workload execution times compared to WarpDrive [23], outperforms custom FPGA designs [3, 33, 68], and achieves 1.53–1.76× speedups over Phantom [67] for basic mechanisms.

Acknowledgments

We appreciate the invaluable feedback from Sangpyo Kim, Jaiyoung Park, Hyesung Ji, and Hyunah Yu. This research was in part supported by Institute of Information & communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) [RS-2021-II211343, RS-2023-00256081, RS-2025-02217656]. Jongmin Kim is with the Interdisciplinary Program in Artificial Intelligence (IPAI), Seoul National University (SNU). Wonseok Choi is with the Department of Intelligence and Information (DII), SNU. Jung Ho Ahn, the corresponding author, is with DII and IPAI, SNU.

References

[1] Rashmi Agrawal, Jung Ho Ahn, Flavio Bergamaschi, Ro Cammarota, Jung Hee Cheon, Fillipe D. M. de Souza, Huijing Gong, Minsik Kang, Duhyeong Kim, Jongmin Kim, Hubert de Lassus, Jai Hyun Park, Michael Steiner, and Wen Wang. 2023. High-Precision RNS-CKKS on Fixed but Smaller Word-Size Architectures: Theory and Application. In *Workshop on Encrypted Computing & Applied Homomorphic Cryptography*. doi:10.1145/3605759.3625257

[2] 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 *MICRO*. doi:10.1145/3613424.3614302

[3] 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 *HPCA*. doi:10.1109/HPCA56546.2023.10070953

[4] 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 *Workshop on Encrypted Computing & Applied Homomorphic Cryptography*. doi:10.1145/3560827.3563379

[5] 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. 2021. Homomorphic Encryption Standard. In *Protecting Privacy through Homomorphic Encryption*. Springer, 31–62. doi:10.1007/978-3-030-77287-1_2

[6] Ahmad Al Badawi and Yuriy Polyakov. 2023. Demystifying Bootstrapping in Fully Homomorphic Encryption. *IACR Cryptology ePrint Archive* 149 (2023). https://eprint.iacr.org/2023/149.

[7] Paul Barrett. 1986. Implementing the Rivest Shamir and Adleman Public Key Encryption Algorithm on a Standard Digital Signal Processor. In *Annual International Conference on the Theory and Application of Cryptographic Techniques*. doi:10.5555/36664.36688

[8] Fabian Boemer, Sejun Kim, Gelila Seifu, Fillipe D. M. de Souza, and Vinodh Gopal. 2021. Intel HEXL: Accelerating Homomorphic Encryption with Intel AVX512-IFMA52. In *Workshop on Encrypted Computing & Applied Homomorphic Cryptography*. doi:10.1145/3474366.3486926

[9] Jean-Philippe Bossuat, Christian Mouchet, Juan Ramón 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*. doi:10.1007/978-3-030-77870-5_21

[10] Jean-Philippe Bossuat, Juan Troncoso-Pastoriza, and Jean-Pierre Hubaux. 2022. Bootstrapping for Approximate Homomorphic Encryption with Negligible Failure-Probability by Using Sparse-Secret Encapsulation. In *Applied Cryptography and Network Security*. doi:10.1007/978-3-031-09234-3_26

[11] Hao Chen, Kim Laine, and Rachel Player. 2017. Simple Encrypted Arithmetic Library - SEAL v2.1. In *Financial Cryptography and Data Security*. doi:10.1007/978-3-319-70278-0_1

[12] 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 *ACM Conference on Computer and Communications Security*.

[13] Jung Hee Cheon, Kyoohyung Han, Andrey Kim, Miran Kim, and Yongsoo Song. 2018. A Full RNS Variant of Approximate Homomorphic Encryption. In *Selected Areas in Cryptography*. doi:10.1007/978-3-030-10970-7_16

[14] Jung Hee Cheon, Kyoohyung Han, Andrey Kim, Miran Kim, and Yongsoo Song. 2018. Bootstrapping for Approximate Homomorphic Encryption. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. doi:10.1007/978-3-319-78381-9_14

[15] Jung Hee Cheon, Andrey Kim, Miran Kim, and Yong Soo Song. 2017. Homomorphic Encryption for Arithmetic of Approximate Numbers. In *International Conference on the Theory and Applications of Cryptology and Information Security*. doi:10.1007/978-3-319-70694-8_15

[16] Jung Hee Cheon, Yongha Son, and Donggeon Yhee. 2022. Practical FHE Parameters against Lattice Attacks. *Journal of the Korean Mathematical Society* 59, 1 (2022), 35–51. doi:10.4134/JKMS.j200650

[17] James W. Cooley and John W. Tukey. 1965. An Algorithm for the Machine Calculation of Complex Fourier Series. *Math. Comp.* 19, 90 (1965), 297–301. doi:10.1090/s0025-5718-1965-0178586-1

[18] CryptoLab. 2023. HEaaN Private AI Homomorphic Encryption Library. https://heaan.it.

[19] Roshan Dathathri, Blagovesta Kostova, Olli Saarikivi, Wei Dai, Kim Laine, and Madan Musuvathi. 2020. EVA: An Encrypted Vector Arithmetic Language and Compiler for Efficient Homomorphic Computation. In *ACM SIGPLAN International Conference on Programming Language Design and Implementation*. doi:10.1145/3385412.3386023

[20] DESILO. 2023. Liberate.FHE: A New FHE Library for Bridging the Gap between Theory and Practice with a Focus on Performance and Accuracy. https://github.com/Desilo/liberate-fhe.

[21] Sultan Durrani, Muhammad Saad Chughtai, Mert Hidayetoglu, Rashid Tahir, Abdul Dakkak, Lawrence Rauchwerger, Fareed Zaffar, and Wen-mei Hwu. 2021. Accelerating Fourier and Number Theoretic Transforms using Tensor Cores and Warp Shuffles. In *PACT*. doi:10.1109/PACT52795.2021.00032

[22] Anne C. Elster and Tor A. Haugdahl. 2022. Nvidia Hopper GPU and Grace CPU Highlights. *Computing in Science & Engineering* 24, 2 (2022), 95–100. doi:10.1109/MCSE.2022.3163817

[23] 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 *HPCA*. doi:10.1109/HPCA61900.2025.00091

[24] Guang Fan, Fangyu Zheng, Lipeng Wan, Lili Gao, Yuan Zhao, Jiankuo Dong, Yixuan Song, Yuewu Wang, and Jingqiang Lin. 2023. Towards Faster Fully Homomorphic Encryption Implementation with Integer and Floating-point Computing Power of GPUs. In *IEEE International Parallel and Distributed Processing Symposium*. doi:10.1109/IPDPS54959.2023.00085

[25] Shengyu Fan, Zhiwei Wang, Weizhi Xu, Rui Hou, Dan Meng, and Mingzhe Zhang. 2023. TensorFHE: Achieving Practical Computation on Encrypted Data Using GPGPU. In *HPCA*. doi:10.1109/HPCA56546.2023.10071017

[26] Craig Gentry. 2009. Fully Homomorphic Encryption Using Ideal Lattices. In *ACM Symposium on Theory of Computing*. doi:10.1145/1536414.1536440

[27] Jia-Zheng Goey, Wai-Kong Lee, Bok-Min Goi, and Wun-She Yap. 2021. Accelerating Number Theoretic Transform in GPU Platform for Fully Homomorphic Encryption. *The Journal of Supercomputing* 77, 2 (2021), 1455–1474. doi:10.1007/s11227-020-03156-7

[28] Shai Halevi and Victor Shoup. 2020. Design and implementation of HElib: a homomorphic encryption library. *IACR Cryptology ePrint Archive* 1481 (2020). https://eprint.iacr.org/2020/1481.

[29] Kyoohyung Han, Seungwan Hong, Jung Hee Cheon, and Daejun Park. 2019. Logistic Regression on Homomorphic Encrypted Data at Scale. In *AAAI Conference on Artificial Intelligence*. doi:10.1609/aaai.v33i01.33019466

[30] Kyoohyung Han and Dohyeong Ki. 2020. Better Bootstrapping for Approximate Homomorphic Encryption. In *Cryptographers' Track at the RSA Conference*. doi:10.1007/978-3-030-40186-3_16

[31] David Harvey. 2014. Faster Arithmetic for Number-Theoretic Transforms. *Journal of Symbolic Computation* 60 (2014), 113–119. doi:10.1016/j.jsc.2013.09.002

[32] Seungwan Hong, Seunghong Kim, Jiheon Choi, Younho Lee, and Jung Hee Cheon. 2021. Efficient Sorting of Homomorphic Encrypted Data With k-Way Sorting Network. *IEEE Transactions on Information Forensics and Security* 16 (2021), 4389–4404. doi:10.1109/TIFS.2021.3106167

[33] Yi Huang, Xinsheng Gong, Xiangyu Kong, Dibei Chen, Jianfeng Zhu, Wenping Zhu, Liangwei Li, Mingyu Gao, Shaojun Wai, Aoyang Zhang, and Leibo Liu. 2025. EFFACT: A Highly Efficient Full-Stack FHE Acceleration Platform. In *HPCA*. doi:10.1109/HPCA61900.2025.00088

[34] Jae Hyung Ju, Jaiyoung Park, Jongmin Kim, Minsik Kang, Donghwan Kim, Jung Hee Cheon, and Jung Ho Ahn. 2024. NeuJeans: Private Neural Network Inference with Joint Optimization of Convolution and FHE Bootstrapping. In *ACM Conference on Computer and Communications Security*. doi:10.1145/3658644.3690375

[35] Wonkyung Jung, Daejin Jung, Byeongho Kim, Sunjung Lee, Wonjong Rhee, and Jung Ho Ahn. 2019. Restructuring Batch Normalization to Accelerate CNN Training. In *Proceedings of Machine Learning and Systems*. https://proceedings.mlsys.org/paper_files/paper/2019/file/9db64c20dee0011899dfdf200e61ef35-Paper.pdf

[36] 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, 4 (2021), 114–148. doi:10.46586/tches.v2021.i4.114-148

[37] Wonkyung Jung, Eojin Lee, Sangpyo Kim, Jongmin Kim, Namhoon Kim, Keewoo Lee, Chohong Min, Jung Hee Cheon, and Jung Ho Ahn. 2021. Accelerating Fully Homomorphic Encryption Through Architecture-Centric Analysis and Optimization. *IEEE Access* 9 (2021), 98772–98789. doi:10.1109/ACCESS.2021.3096189

[38] Chiraag Juvekar, Vinod Vaikuntanathan, and Anantha Chandrakasan. 2018. GAZELLE: A Low Latency Framework for Secure Neural Network Inference. In *USENIX Security Symposium*. https://www.usenix.org/conference/usenixsecurity18/presentation/juvekar

[39] Andrew Kerr, Duane Merrill, Julien Demouth, and John Tran. 2017. CUTLASS: Fast Linear Algebra in CUDA C++. https://developer.nvidia.com/blog/cutlass-linear-algebra-cuda.

[40] Andrey Kim, Antonis Papadimitriou, and Yuriy Polyakov. 2022. Approximate Homomorphic Encryption with Reduced Approximation Error. In *Cryptographers' Track at the RSA Conference*. doi:10.1007/978-3-030-95312-6_6

[41] Donghwan Kim, Jaiyoung Park, Jongmin Kim, Sangpyo Kim, and Jung Ho Ahn. 2024. HyPHEN: A Hybrid Packing Method and Its Optimizations for Homomorphic Encryption-Based Neural Networks. *IEEE Access* 12 (2024), 3024–3038. doi:10.1109/ACCESS.2023.3348170

[42] 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 *ISCA*. doi:10.1145/3579371.3589053

[43] 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 *MICRO*. doi:10.1109/MICRO56248.2022.00086

[44] Jongmin Kim, Sungmin Yun, Hyesung Ji, Wonseok Choi, Sangpyo Kim, and Jung Ho Ahn. 2025. Anaheim: Architecture and Algorithms for Processing Fully Homomorphic Encryption in Memory. In *HPCA*. doi:10.1109/HPCA61900.2025.00089

[45] Sangpyo Kim, Wonkyung Jung, Jaiyoung Park, and Jung Ho Ahn. 2020. Accelerating Number Theoretic Transformations for Bootstrappable Homomorphic Encryption on GPUs. In *IEEE International Symposium on Workload Characterization*. doi:10.1109/IISWC50251.2020.00033

[46] 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 *ISCA*. doi:10.1145/3470496.3527415

[47] 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*. https://proceedings.mlr.press/v162/lee22e.html

[48] 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. doi:10.1109/ACCESS.2022.3159694

[49] Seewoo Lee, Garam Lee, Jung Woo Kim, Junbum Shin, and Mun-Kyu Lee. 2023. HETAL: Efficient Privacy-preserving Transfer Learning with Homomorphic Encryption. In *International Conference on Machine Learning*. https://proceedings.mlr.press/v202/lee23m.html

[50] Wai-Kong Lee, Sedat Akleylek, Denis Chee-Keong Wong, Wun-She Yap, Bok-Min Goi, and Seong-Oun Hwang. 2021. Parallel implementation of Nussbaumer algorithm and number theoretic transform on a GPU platform: application to qTESLA. *The Journal of Supercomputing* 77, 4 (2021), 3289–3314. doi:10.1007/s11227-020-03392-x

[51] Peter L. Montgomery. 1985. Modular Multiplication without Trial Division. *Math. Comp.* 44, 170 (1985), 519–521. doi:10.1090/S0025-5718-1985-0777282-X

[52] Christian Vincent Mouchet, Jean-Philippe Bossuat, Juan Ramón Troncoso-Pastoriza, and Jean-Pierre Hubaux. 2020. Lattigo: A Multiparty Homomorphic Encryption Library in Go. In *Workshop on Encrypted Computing and Applied Homomorphic Cryptography*. doi:10.25835/0072999

[53] Ali Şah Özcan and Erkay Savaş. 2024. HEonGPU: a GPU-based Fully Homomorphic Encryption Library 1.0. *IACR Cryptology ePrint Archive* 1543 (2024). https://eprint.iacr.org/2024/1543.

[54] Jaiyoung Park, Donghwan Kim, Jongmin Kim, Sangpyo Kim, Wonkyung Jung, Jung Hee Cheon, and Jung Ho Ahn. 2023. Toward Practical Privacy-Preserving Convolutional Neural Networks Exploiting Fully Homomorphic Encryption. In *Workshop on Data Integrity and Secure Cloud Computing*. https://dtrilla.github.io/discc-workshop-2023/assets/pdfs/DISCC_2023_paper_4.pdf

[55] Jaiyoung Park, Michael Jaemin Kim, Wonkyung Jung, and Jung Ho Ahn. 2022. AESPA: Accuracy Preserving Low-Degree Polynomial Activation for Fast Private Inference. *arXiv preprint* (2022). doi:10.48550/arXiv.2201.06699

[56] Robert Podschwadt and Daniel Takabi. 2020. Classification of Encrypted Word Embeddings using Recurrent Neural Networks. In *Workshop on Privacy in Natural Language Processing*. https://ceur-ws.org/Vol-2573/PrivateNLP_Paper3.pdf

[57] M. Sadegh Riazi, Kim Laine, Blake Pelton, and Wei Dai. 2020. HEAX: An Architecture for Computing on Encrypted Data. In *ASPLOS*. doi:10.1145/3373376.3378523

[58] 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 *HPCA*. doi:10.1109/HPCA.2019.00052

[59] 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*. doi:10.1145/3466752.3480070

[60] 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 *ISCA*. doi:10.1145/3470496.3527393

[61] Nikola Samardzic and Daniel Sanchez. 2024. BitPacker: Enabling High Arithmetic Efficiency in Fully Homomorphic Encryption Accelerators. In *ASPLOS*. doi:10.1145/3620665.3640397

[62] Gregor Seiler. 2018. Faster AVX2 Optimized NTT Multiplication for Ring-LWE Lattice Cryptography. *IACR Cryptology ePrint Archive* 39 (2018). https://eprint.iacr.org/2018/39.

[63] Shiyu Shen, Hao Yang, Yu Liu, Zhe Liu, and Yunlei Zhao. 2022. CARM: CUDA-accelerated RNS multiplication in word-wise homomorphic encryption schemes for internet of things. *IEEE Trans. Comput.* 72, 7 (2022), 1999–2010. doi:10.1109/TC.2022.3227874

[64] 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 *MICRO*. doi:10.1145/3613424.3614279

[65] Kaustubh Shivdikar, Gilbert Jonatan, Evelio Mora, Neal Livesay, Rashmi Agrawal, Ajay Joshi, José L. Abellán, John Kim, and David Kaeli. 2022. Accelerating Polynomial Multiplication for Homomorphic Encryption on GPUs. In *IEEE International Symposium on Secure and Private Execution Environment Design*. doi:10.1109/SEED55351.2022.00013

[66] Victor Shoup. 2021. NTL: A Library for Doing Number Theory (Version 11.5.1). https://libntl.org.

[67] 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. doi:10.1109/TDSC.2024.3363900

[68] Yinghao Yang, Huaizhi Zhang, Shengyu Fan, Hang Lu, Mingzhe Zhang, and Xiaowei Li. 2023. Poseidon: Practical Homomorphic Encryption Accelerator. In *HPCA*. doi:10.1109/HPCA56546.2023.10070984

[69] Yujia Zhai, Mohannad Ibrahim, Yiqin Qiu, Fabian Boemer, Zizhong Chen, Alexey Titov, and Alexander Lyashevsky. 2022. Accelerating Encrypted Computing on Intel GPUs. In *IEEE International Parallel and Distributed Processing Symposium*. doi:10.1109/IPDPS53621.2022.00074

[70] Yilan Zhu, Xinyao Wang, Lei Ju, and Shanqing Guo. 2023. FxHENN: FPGA-based Acceleration Framework for Homomorphic Encrypted CNN Inference. In *HPCA*. doi:10.1109/HPCA56546.2023.10071133