Developing Accelerators for Homomorphic Encryption

Ahmet Vedat Kurt[1], Borhan Javadian[1], and Efe İzbudak[2]

[1]Sabancı University
[2]Middle East Technical University

August 2025

Abstract

Homomorphic Encryption (HE) enables computation on encrypted data but is hampered by significant performance overheads. This work builds necessary utilities for accelerating HE operations on Graphics Processing Units (GPUs). We present two main contributions. First, we develop a pure Python implementation of the TFHE scheme to serve as a high-level prototyping sandbox, enabling rapid verification of complex functionalities like scheme switching before low-level implementation. Second, we conduct a comprehensive CPU benchmark of encrypted matrix multiplication algorithms using Microsoft SEAL. This benchmark provides a data-driven roadmap for identifying algorithms worthy of GPU implementation and establishes a baseline for quantifying performance gains.

ahmet.kurt@sabanciuniv.edu, borhan.javadian@sabanciuniv.edu, efe.izbudak@metu.edu.tr

2020 Mathematics Subject Classification. Primary 94A60, 68W10; Secondary 11T71, 68P25.

Key words and phrases. Homomorphic Encryption, GPU Acceleration, Lattice-Based Cryptography, TFHE, CKKS.

Contents

1 Introduction

A public-key encryption scheme $\mathcal{E} = (\text{KeyGen}, \text{Encrypt}, \text{Decrypt})$ is said to be **Fully Homomorphic (FHE)** if it is augmented with a fourth efficient algorithm, Evaluate. For any function f expressible as a circuit C, any public key pk, and any set of ciphertexts $c_i = \text{Encrypt}(pk, m_i)$, the Evaluate algorithm can compute a ciphertext $c_{res} = \text{Evaluate}(pk, C, c_1, \ldots, c_k)$ such that the following correctness equation holds:

$$\text{Decrypt}(sk, c_{res}) = C(m_1, \ldots, m_k)$$

The homomorphic workflow, depicted in Figure 1, illustrates this process, enabling applications such as privacy-preserving genomic analysis on a third-party cloud or secure fraud detection by an external service, all without exposing the raw, sensitive information. The first plausible construction of such a scheme was proposed by Gentry [Gen09], initiating a line of research that has since evolved through several distinct generations.

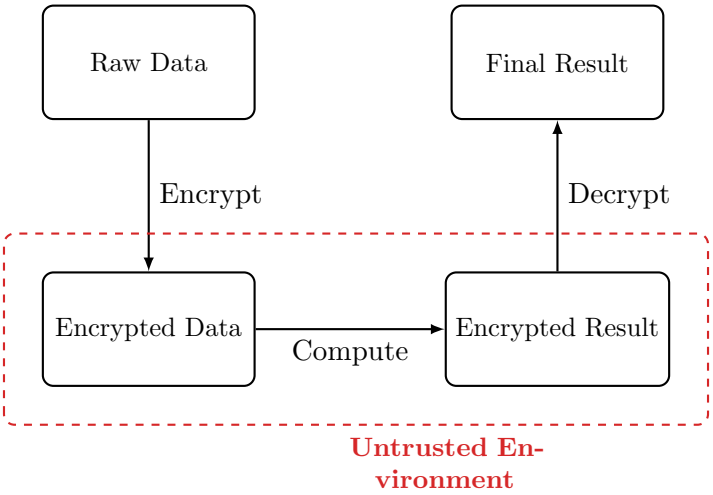

Figure 1: The Homomorphic Encryption Workflow.

The **first generation** of FHE was founded on encryption over ideal lattices. The central challenge in any practical HE scheme is the management of cryptographic "noise". A scheme that can only support a limited depth of operations before the noise corrupts the message is termed Somewhat Homomorphic (SHE). Gentry's primary contribution was bootstrapping: a procedure to transform any SHE scheme into an FHE scheme, provided the SHE scheme is capable of homomorphically evaluating its own decryption circuit. This allows a noisy ciphertext to be "refreshed" by homomorphically decrypting it, thereby resetting its noise level and enabling computations of arbitrary depth. To provide an intuitive model for this functionality, Gentry proposed a conceptual analogy of a jewelry store owner who secures raw precious materials in a locked, transparent glovebox. A worker can then use the gloves to assemble the materials into a finished piece without having direct physical access to them, returning the locked box to the owner who is the sole holder of the key [Gen10].

The **second generation** of FHE schemes (c. 2011-2012), including BGV, BFV, and GSW, shifted the cryptographic foundation to the conjectured hardness of the Learning With Errors (LWE) and Ring-LWE (RLWE) problems. These schemes introduced more efficient noise management techniques that avoided the costly bootstrapping procedure for circuits of a predetermined depth. The core of an LWE-based encryption of a message μ is

a ciphertext $(\mathbf{a}, b) \in \mathbb{Z}_q^n \times \mathbb{Z}_q$, where $b = \langle \mathbf{a}, \mathbf{s} \rangle + e + \Delta\mu$. Here, $\mathbf{s}$ is the secret key, e is a small noise term, and $\Delta = q/t$ is a scaling factor that maps a plaintext in $\mathbb{Z}_t$ to a higher-order part of the ciphertext space $\mathbb{Z}_q$. Noise growth from multiplication is managed primarily through modulus switching, an operation that transforms a ciphertext modulo q into a ciphertext encrypting the same message modulo a smaller $p < q$. This operation scales down the magnitude of both the message and the noise, controlling noise accumulation at the cost of consuming the modulus chain.

A **third generation** of schemes specialized their constructions to achieve greater efficiency for specific use cases. The Cheon-Kim-Kim-Song (CKKS) scheme [CKKS17] is optimized for approximate arithmetic on real or complex numbers. It encodes a vector $\mathbf{z} \in \mathbb{C}^{N/2}$ into a polynomial in the ring $\mathcal{R} = \mathbb{Z}[X]/\langle X^{2N} + 1 \rangle$ via the canonical embedding. Its core innovation is to treat the cryptographic noise as an intrinsic part of the computation's approximation error. Noise and the plaintext scaling factor are managed via a rescaling operation, $c \mapsto \lfloor c/p \rceil \pmod{q/p}$, which is analogous to truncating least-significant bits in fixed-point arithmetic. In contrast, the TFHE scheme [CGGI20] is optimized for bit-wise operations over the real torus $\mathbb{T} = \mathbb{R}/\mathbb{Z}$. Its principal advantage is an exceptionally fast bootstrapping procedure that can be programmed to evaluate an arbitrary univariate function f on the plaintext during the noise-clearing operation itself, with no marginal cost beyond a standard refresh.

Despite these algorithmic advances, the practical application of HE remains impeded by several fundamental drawbacks:

Despite the major improvements in homomorphic encryption primitives, the practical adoption is still limited by:

- **Performance Overhead:** Homomorphic operations are orders of magnitude slower than their plaintext counterparts. This latency is a direct consequence of the large parameters required for security and the complexity of the underlying polynomial arithmetic.
- **Noise Growth:** Every operation introduces a small amount of cryptographic noise into the ciphertext. The accumulation of this noise limits the computational depth, as exceeding a critical threshold renders the ciphertext indecipherable.
- **Ciphertext Expansion:** Ciphertexts are substantially larger than the plaintexts they represent, leading to significant storage and communication overhead. An encryption of a single bit, for instance, may expand to several kilobytes.
- **Implementation Complexity:** The design and secure parameterization of HE-based systems demand specialized expertise in cryptography and algebra to prevent subtle errors that could compromise security or correctness.

Our research directly confronts the **performance bottleneck** by establishing a framework for accelerating HE primitives on Graphics Processing Units (GPUs).

1.1 Motivation for GPU Acceleration

The rationale for targeting GPUs stems from the algebraic structure of modern HE schemes. The computational core of RLWE-based schemes like BFV and CKKS is polynomial arithmetic in rings such as $\mathcal{R}_q = (\mathbb{Z}/q\mathbb{Z})[X]/\langle \Phi_M(X) \rangle$. Operations such as polynomial addition and multiplication on high-degree polynomials are computationally intensive. The most efficient algorithm for polynomial multiplication is the Number Theoretic Transform (NTT), an analogue of the Fast Fourier Transform (FFT) adapted for finite fields.

The NTT, like the FFT, is an inherently parallel algorithm. It decomposes a large polynomial multiplication into a large number of smaller, independent point-wise operations. The fundamental computational units of the NTT, known as butterfly operations, can be executed simultaneously across the entire set of polynomial coefficients. This computational pattern maps directly and efficiently to the architecture of modern GPUs. A GPU is a massively parallel processor comprising thousands of simple cores designed to execute the same instruction on different data concurrently, a model known as SIMT (Single Instruction, Multiple Threads). The structure of the NTT is an ideal fit for the SIMT paradigm, suggesting that GPUs offer a promising path toward significant acceleration of the core HE primitives.

1.2 Our Contributions

This paper details the foundational work for a high-performance, GPU-accelerated HE library, which we term HEonGPU. Our contributions are twofold:

1. We developed a flexible prototyping environment for the TFHE scheme using pure Python. This "sandbox" allows for the rapid verification of complex protocols, such as transciphering, thereby de-risking the more time-consuming C++/CUDA implementation.

2. We established a rigorous CPU benchmark of various encrypted matrix multiplication techniques within the CKKS scheme using Microsoft SEAL. This provides a data-driven roadmap for our GPU development and a baseline against which acceleration can be quantified.

1.3 Related Work

The field of HE acceleration is active. Hardware acceleration has been explored using FPGAs, ASICs, and GPUs. Several software libraries provide implementations of HE schemes, most notably Microsoft SEAL, HElib [HS14], PALISADE, and TFHE-rs. Our work builds upon the algorithms presented by Halevi and Shoup for matrix operations [HS14], which introduced efficient data packing techniques, and more recent encoding strategies like the Bicycle encoding [CYW+24]. We aim to benchmark these against the state-of-the-art method proposed by Park et al. [PKL23] to guide our own GPU library development.

2 Preliminaries

2.1 Notation

Let λ be the security parameter. We denote the ring of integers as $\mathbb{Z}$ and the finite field of q elements as $\mathbb{Z}_q = \mathbb{Z}/q\mathbb{Z}$. Polynomial rings are denoted $\mathcal{R} = \mathbb{Z}[X]/\langle \Phi_M(X) \rangle$, where $\Phi_M(X)$ is the M-th cyclotomic polynomial, an irreducible polynomial of degree $N = \phi(M)$, where ϕ is Euler's totient function. The quotient ring modulo q is $\mathcal{R}_q = \mathcal{R}/q\mathcal{R} = \mathbb{Z}_q[X]/\langle \Phi_M(X) \rangle$. We denote sampling from a distribution χ as $x \leftarrow \chi$. Typically, χ is a discrete Gaussian distribution with a small standard deviation. For a vector $\mathbf{v}$, we use $\|\mathbf{v}\|$ to denote a chosen norm, usually the infinity norm $\|\mathbf{v}\|_\infty = \max_i |v_i|$.

2.2 Lattices

An n-dimensional lattice $\mathcal{L}$ is any subset of $\mathbb{R}^n$ that is both an additive subgroup and discrete. The minimum distance of a lattice $\mathcal{L}$ is the length of a shortest nonzero lattice vector: $\lambda_1(\mathcal{L}) := \min_{\mathbf{v}\in\mathcal{L}\setminus\{\mathbf{0}\}} \|\mathbf{v}\|$.

2.3 The LWE and RLWE Problems

The security of most modern HE schemes is based on the hardness of the Learning With Errors (LWE) problem or its ring-based variant, Ring-LWE (RLWE). Their security is reducible to worst-case hardness of problems on ideal lattices, such as the Shortest Vector Problem (SVP).

Definition 2.1 (LWE [Reg05])**.** For a secret vector $\mathbf{s} \in \mathbb{Z}_q^n$, the LWE distribution $A_{\mathbf{s},\chi}$ is obtained by choosing a vector $\mathbf{a} \in \mathbb{Z}_q^n$ uniformly at random and an error $e \leftarrow \chi$ from a small-norm distribution χ, and outputting the pair $(\mathbf{a}, b = \langle \mathbf{a}, \mathbf{s}\rangle + e) \in \mathbb{Z}_q^n \times \mathbb{Z}_q$. The decisional LWE problem is to distinguish an arbitrary number of samples from $A_{\mathbf{s},\chi}$ from samples drawn uniformly from $\mathbb{Z}_q^n \times \mathbb{Z}_q$.

The LWE problem's structure is amenable to encryption. To encrypt a message μ, we scale it by a factor $\Delta = q/p$ to move it into a higher-order portion of $\mathbb{Z}_q$, and add it to the LWE sample: $b = \langle \mathbf{a}, \mathbf{s}\rangle + e + \Delta\mu$. To decrypt, one computes $b - \langle \mathbf{a}, \mathbf{s}\rangle = e + \Delta\mu$. If the error term e is small enough, it can be removed by rounding, recovering $\Delta\mu$.

Definition 2.2 (RLWE [LPR10])**.** For a secret polynomial $s \in \mathcal{R}_q$, the RLWE distribution is obtained by choosing a polynomial $a \in \mathcal{R}_q$ uniformly at random and an error polynomial $e \leftarrow \chi$ (where coefficients are sampled from χ), and outputting the pair $(a, b = a \cdot s + e) \in \mathcal{R}_q \times \mathcal{R}_q$. The decisional RLWE problem is to distinguish samples from the RLWE distribution from uniform samples from $\mathcal{R}_q \times \mathcal{R}_q$.

RLWE offers significant efficiency gains over LWE. A single RLWE sample can encrypt N values (the coefficients of a polynomial), and polynomial multiplication in $\mathcal{R}_q$ can be performed efficiently using Number Theoretic Transforms (NTT), an analogue of the Fast Fourier Transform (FFT) over finite fields.

2.4 Homomorphic Encryption Schemes

2.4.1 CKKS: The Cheon-Kim-Kim-Song Scheme

The CKKS scheme [CKKS17] is designed for approximate arithmetic over real or complex numbers. Its core innovation is to treat the noise inherent in LWE-based cryptography not as a nuisance to be eliminated, but as part of the error that naturally occurs in approximate computations. This allows for a more efficient handling of real-number arithmetic.

Encoding and Plaintext Space. The plaintext space of CKKS is the polynomial ring $\mathcal{R} = \mathbb{Z}[X]/(\Phi_M(X))$. To handle vectors of complex or real numbers, CKKS uses the canonical embedding $\sigma : \mathcal{R} \to \mathbb{C}^N$ to map a polynomial to a vector of its evaluations at the odd powers of the M-th primitive root of unity. This map is an isometric ring homomorphism, meaning it preserves the magnitude of the plaintext and any associated errors during the encoding and decoding processes. To encode a vector $\mathbf{z} \in \mathbb{C}^{N/2}$, it is first scaled by a factor Δ, then mapped to a polynomial $m(X) \in \mathcal{R}$ via the inverse embedding σ^{-1}. Upon decryption, the resulting polynomial $m'(X)$ is decoded back to a vector via σ and scaled down by Δ^{-1}.

Decryption and Noise. A fresh ciphertext $c = (b, a)$ encrypting a plaintext polynomial m has the form $b + as \approx \Delta m \pmod{q}$. More formally, decryption computes $\langle c, sk \rangle = b + as = \Delta m + e \pmod{q}$, where $sk = (1, s)$ and e is a small error polynomial. The key insight of CKKS is that as long as the magnitude of the error e is small relative to the scaled message $\Delta \cdot \mathbf{z}$, the value $m' = \Delta m + e$ can be treated as an approximate representation of the original message.

Rescaling. Homomorphic multiplication of two ciphertexts encrypting m_1 and m_2 results in a ciphertext that decrypts to $(\Delta m_1 + e_1)(\Delta m_2 + e_2) = \Delta^2 m_1 m_2 + \Delta(m_1 e_2 + m_2 e_1) + e_1 e_2$. The new error is significantly larger, and the message is now scaled by Δ^2. Without intervention, the message and noise would quickly exceed the modulus q. To manage this, CKKS introduces a rescaling procedure. Given a ciphertext c at modulus q, rescaling produces a new ciphertext $c' = \lfloor p^{-1} \cdot c \rceil$ at a smaller modulus $q' = q/p$. This operation has the effect of dividing the underlying plaintext and noise by the factor p. If we choose $p \approx \Delta$, the plaintext is returned to its original scaling factor. It is analogous to the rounding step in floating-point arithmetic, where least significant bits are truncated to manage the size of the mantissa. This procedure ensures that the bit-length of the modulus grows only linearly with the multiplicative depth of the circuit, a major advantage over schemes that require exponential growth.

2.4.2 TFHE: Torus Fully Homomorphic Encryption

The TFHE scheme [CGGI20] is optimized for bit-wise operations and features a highly efficient bootstrapping procedure. Its plaintext space consists of individual bits or small integers encoded as elements on the real torus $\mathbb{T} = \mathbb{R}/\mathbb{Z}$, which for practical implementations is discretized to $\mathbb{T}_q = q^{-1}\mathbb{Z}/\mathbb{Z}$ for some modulus q. A plaintext message μ is an element of a smaller discretized torus $\mathbb{T}_p = p^{-1}\mathbb{Z}/\mathbb{Z}$ where $p|q$.

A TFHE ciphertext over the discretized torus, specifically a TLWE sample, is an LWE sample $(\mathbf{a}, b) \in \mathbb{T}_q^n \times \mathbb{T}_q$, where $b = \langle \mathbf{a}, \mathbf{s} \rangle + \mu + e$. Here, $\mathbf{s} \in \{0, 1\}^n$ is the secret key, $\mu \in \mathbb{T}_p$ is the encoded plaintext, and e is a small noise term drawn from a discrete Gaussian distribution. Decryption is performed by computing the phase $\phi(b, \mathbf{a}) = b - \langle \mathbf{a}, \mathbf{s} \rangle = \mu + e$. If the noise e is sufficiently small, specifically $\|e\|_\infty < 1/(2p)$, the original plaintext μ can be recovered by rounding ϕ to the nearest valid plaintext in $\mathbb{T}_p$. The critical challenge is that homomorphic operations increase this noise. Once the noise grows too large, it corrupts the message, rendering decryption impossible. TFHE's key feature is its fast bootstrapping mechanism, which resets the noise, allowing for the evaluation of arbitrary-depth circuits.

3 Prototyping Sandbox for TFHE

To accelerate development and mitigate risk, we constructed a pure Python implementation of TFHE. While not performant, this "sandbox" is invaluable for de-risking the much more complex and time-intensive GPU implementation in C++/CUDA. Its primary purpose is to prototype advanced HE functionalities, such as scheme switching, allowing us to verify the correctness of complex, multi-stage homomorphic processes before committing to a low-level implementation. The high-level, abstract nature of Python allows for rapid iteration on the algebraic structures and protocol flows, ensuring logical correctness without the overhead of memory management, parallelization, and low-level optimizations. This allows us to focus purely on the cryptographic logic.

3.1 TFHE Implementation on Python Sandbox

3.1.1 TFHE Plaintexts and Ciphertexts

In our implementation, plaintexts and ciphertexts follow the torus-based origin of TFHE [CGGI20, Joy22] while adapting it to a discretized, polynomial-based setting suitable for prototyping. We distinguish between plaintext representation and ciphertext formats.

Plaintexts and Encoding Plaintexts are represented on the discretized torus

$$\mathbb{T}_q = \tfrac{1}{q}\mathbb{Z} \,/\, \mathbb{Z}, \qquad q = 2^{32},$$

which can be identified with $\mathbb{Z}_q$ [Joy22]. In practice, an integer message $m \in \mathbb{Z}_t$ is embedded in the most significant bits of the constant coefficient of a polynomial in $\mathbb{Z}_q[X]/(X^N+1)$ by scaling with $\Delta = q/t$:

$$m \;\mapsto\; \tilde{m}(X) = (m \cdot \Delta) \bmod q, \quad \tilde{m}(X) \in \mathbb{Z}_q[X]/(X^N+1).$$

For Learning With Errors (LWE) ciphertexts we set $N = 1$, so plaintexts reduce to constant coefficients.

Ciphertext Formats Ciphertexts in TFHE are LWE- or GLWE-based depending on the operation:

- **LWE ciphertexts** encrypt messages encoded as constant plaintexts and take the form
$$c = (\mathbf{a}, b) \in \mathbb{Z}_q^n \times \mathbb{Z}_q,$$
where $b = \langle \mathbf{a}, \mathbf{s} \rangle + m + e \pmod q$, with secret key $\mathbf{s} \in \mathbb{Z}_q^n$, message $m \in \mathbb{T}_q$, and error $e \leftarrow \chi$ drawn from a discrete Gaussian distribution [Reg05].
- **GLWE ciphertexts** are used in bootstrapping and higher-level procedures. They live in
$$(\mathbf{a}(X), b(X)) \;\in\; \big(\mathbb{Z}_q[X]/(X^N+1)\big)^k \times \mathbb{Z}_q[X]/(X^N+1),$$
where the secret key is a vector of polynomials $\mathbf{s}(X) \in \big(\mathbb{Z}_q[X]/(X^N+1)\big)^k$.

In both cases, ciphertexts are of the form $(\mathbf{a}(X), b(X))$, with $N = 1$ for LWE and larger N for GLWE.

Encryption and Decryption To encrypt a plaintext $\tilde{m}(X)$, we first sample a secret key $\mathbf{s}(X)$ from the binary polynomial ring $\big(\mathbb{Z}_2[X]/(X^N+1)\big)^k$, which is standard in TFHE to enable efficient bootstrapping [CGGI20, Joy22]. We then sample a random mask $\mathbf{a}(X) \leftarrow \big(\mathbb{Z}_q[X]/(X^N+1)\big)^k$ and a small error polynomial $e(X) \leftarrow \chi$. The ciphertext is formed as

$$c = (\mathbf{a}(X),\ b(X)), \quad \text{where } b(X) = \langle \mathbf{a}(X), \mathbf{s}(X) \rangle + \tilde{m}(X) + e(X) \pmod q.$$

This construction ensures semantic security under the decisional LWE/GLWE assumptions. Decryption consists of computing the *phase* $\varphi_{\mathbf{s}}(c) = b - \langle \mathbf{a}, \mathbf{s} \rangle \pmod q$, which recovers $m+e$. The message is then recovered by scaling and rounding: $m \approx \mathrm{Round}(\varphi_{\mathbf{s}}(c)/\Delta) \bmod t$.

3.1.2 Basic Homomorphic Operations

Once plaintexts are encrypted, TFHE supports several homomorphic operations that preserve correctness while operating directly on input ciphertexts without additional operations. In our implementation, we provide the following basic operations [CGGI20, Joy22].

- **Ciphertext–Ciphertext Addition:** Given $c_1 = (\mathbf{a}_1, b_1)$ and $c_2 = (\mathbf{a}_2, b_2)$, their sum is $c_{\text{sum}} = (\mathbf{a}_1 + \mathbf{a}_2,\ b_1 + b_2 \bmod q)$.
- **Ciphertext–Plaintext Addition:** Given a ciphertext $c = (\mathbf{a}, b)$ and a plaintext $\tilde{p}$, their sum is $c' = (\mathbf{a},\ b + \tilde{p} \bmod q)$.
- **Ciphertext–Plaintext Multiplication:** Given a ciphertext $c = (\mathbf{a}, b)$ and a small plaintext p, their product is $c' = (p \cdot \mathbf{a},\ p \cdot b \bmod q)$.

Each operation increases the noise, which must remain below the decryption threshold $\Delta/2$ for correctness.

3.1.3 Ciphertext–Ciphertext Multiplication via TGGSW

A crucial building block of TFHE is the use of *TGGSW* ciphertexts, which enable ciphertext–ciphertext multiplication through the external product. This mechanism relies on polynomial decomposition, the GLev representation, and the TGGSW encryption structure [CGGI20, Joy22].

Polynomial Decomposition. The mechanism relies on gadget decomposition. For a given base B and decomposition length ℓ, any polynomial $p(X) \in \mathbb{Z}_q[X]/(X^N + 1)$ can be expressed approximately as

$$p(X) \approx \sum_{j=1}^{\ell} d_j(X) \cdot \frac{q}{B^j},$$

where each $d_j(X)$ is a polynomial with coefficients in $\mathbb{Z}_B$, i.e. "small" digits. This expansion shows how large coefficients of $p(X)$ can be represented using small digits $d_j(X)$ combined with gadget weights q/B^j.

To represent these digits, we define the decomposition operator

$$\text{Decomp}^{B,\ell}(p(X)) = \big(d_1(X), d_2(X), \ldots, d_\ell(X)\big),$$

which returns the vector of ℓ polynomials corresponding to the decomposition digits of $p(X)$. This decomposition is used to break down large polynomial multiplications into combinations of smaller multiples, aligned with the elements of the Glev representation.

GLev Representation. Given a plaintext $\mu(X)$, the GLev representation is the vector of GLWE ciphertexts

$$\text{GLev}(\mu(X)) = \big(\text{GLWE}(q/B \cdot \mu(X)),\ \text{GLWE}(q/B^2 \cdot \mu(X)),\ \ldots,\ \text{GLWE}(q/B^\ell \cdot \mu(X))\big).$$

Each row of the TGGSW ciphertext consists of such a GLev vector, encoding scaled versions of $\mu(X)$ or $-s_i(X)\mu(X)$.

TGGSW Structure. A TGGSW ciphertext encrypting $\mu(X)$ is thus a $(k+1) \times \ell$ matrix:

$$\mathrm{TGGSW}(\mu(X)) = \begin{bmatrix} \mathrm{GLev}(-s_1(X)\mu(X)) \\ \vdots \\ \mathrm{GLev}(-s_k(X)\mu(X)) \\ \mathrm{GLev}(\mu(X)) \end{bmatrix}.$$

Here each row is a *GLev vector*, i.e. a sequence of ℓ GLWE ciphertexts corresponding to different decomposition levels. The first k rows encode $-s_i(X)\mu(X)$ for each component $s_i(X)$ of the secret key $\mathbf{s}(X) = (s_1(X), \ldots, s_k(X))$, while the final row encodes $\mu(X)$ itself. This structure ensures that when combined with another ciphertext, the secret key contributions cancel in a controlled way, leaving a valid encryption of the product. When $\mathrm{N} = 1$, the TGGSW reduces to a TGSW ciphertext.

Unscaled Message Embedding. A key feature of TGSW/TGGSW is that, unlike standard LWE/GLWE encryption where messages are scaled by $\Delta = q/t$, here the message is *not scaled*. Instead, the ciphertext directly encodes multiples of the raw message in its structure. This design is what makes the external product possible: the message itself acts as a multiplicative factor on another ciphertext. As a result, if $C = \mathrm{TGGSW}(\mu(X))$ and $c = \mathrm{GLWE}(\Delta \cdot m(X))$, then the external product yields

$$C \star c \;\approx\; \mathrm{GLWE}\big(\Delta \cdot \mu(X) \cdot m(X)\big),$$

showing how the unscaled message $\mu(X)$ in the TGGSW acts as a homomorphic multiplier on the encrypted message $m(X)$.

External Product. Given a TGGSW ciphertext $C = \mathrm{TGGSW}(\mu(X))$ and a GLWE ciphertext $c = (\mathbf{a}(X), b(X))$ encrypting $m(X)$, their external product is defined as

$$C \star c \;=\; \sum_{i=1}^{k} \Big\langle \mathrm{GLev}(-s_i(X)\mu(X)),\ \mathrm{Decomp}(a_i(X)) \Big\rangle \;+\; \Big\langle \mathrm{GLev}(\mu(X)),\ \mathrm{Decomp}(b(X)) \Big\rangle.$$

Here $\mathrm{Decomp}(\cdot)$ denotes the gadget decomposition returning a vector of ℓ polynomials, and $\mathrm{GLev}(\cdot)$ is the corresponding vector of GLWE ciphertexts. The notation $\langle \cdot, \cdot \rangle$ denotes the inner product between these two vectors. The result is a new GLWE ciphertext encrypting $\mu(X) \cdot m(X)$ (up to noise).

This mechanism, first introduced by Chillotti *et al.* [CGGI20], is central to TFHE bootstrapping. It allows one ciphertext to act as a homomorphic multiplier on another, turning encrypted bits into programmable control signals.

Proof. Let $c = (\mathbf{a}(X), b(X))$ be a GLWE ciphertext encrypting $m(X)$ under secret key $\mathbf{s}(X) = (s_1(X), \ldots, s_k(X))$ with

$$b(X) \equiv \langle \mathbf{a}(X), \mathbf{s}(X) \rangle + m(X) + e(X) \pmod{q},$$

and let $C = \mathrm{TGGSW}(\mu(X))$ be a TGGSW encryption of an *unscaled* message $\mu(X)$. For a fixed gadget base B and level ℓ, we define decomposition operator as

$$\mathrm{Decomp}^{B,\ell}\big(a_i(X)\big) = \big(d_1^{(i)}(X), \ldots, d_\ell^{(i)}(X)\big), \qquad \mathrm{Decomp}^{B,\ell}\big(b(X)\big) = \big(\delta_1(X), \ldots, \delta_\ell(X)\big),$$

so that the (approximate) reconstructions hold:

$$a_i(X) \approx \sum_{j=1}^{\ell} d_j^{(i)}(X) \cdot \frac{q}{B^j}, \quad b(X) \approx \sum_{j=1}^{\ell} \delta_j(X) \cdot \frac{q}{B^j}.$$

By definition of the GLev representation,

$$\mathrm{GLev}(\mu(X)) = \big(\mathrm{GLWE}(q/B \cdot \mu(X)), \ldots, \mathrm{GLWE}(q/B^{\ell} \cdot \mu(X))\big),$$

and similarly for $\mathrm{GLev}(-s_i(X)\mu(X))$. The external product is defined as the sum of inner products:

$$C \star c = \sum_{i=1}^{k} \Big\langle \mathrm{GLev}(-s_i(X)\mu(X)),\, \mathrm{Decomp}\big(a_i(X)\big) \Big\rangle \;+\; \Big\langle \mathrm{GLev}(\mu(X)),\, \mathrm{Decomp}\big(b(X)\big) \Big\rangle.$$

With basic homomorphic plaintext multiplication and ciphertext addition each inner product equal to the corresponding weighted sum:

$$\Big\langle \mathrm{GLev}(\mu(X)),\, \mathrm{Decomp}\big(b(X)\big) \Big\rangle \approx \mathrm{GLWE}\left(\sum_{j=1}^{\ell} \delta_j(X) \cdot \frac{q}{B^j} \cdot \mu(X) \right) \approx \mathrm{GLWE}\big(b(X) \cdot \mu(X)\big),$$

and, for each i,

$$\begin{aligned} \Big\langle \mathrm{GLev}(-s_i(X)\mu(X)),\, \mathrm{Decomp}\big(a_i(X)\big) \Big\rangle &\approx \mathrm{GLWE}\left(\sum_{j=1}^{\ell} d_j^{(i)}(X) \cdot \frac{q}{B^j} \cdot (-s_i(X)\mu(X)) \right) \\ &= \mathrm{GLWE}\left((-s_i(X)\mu(X)) \cdot \sum_{j=1}^{\ell} d_j^{(i)}(X) \cdot \frac{q}{B^j} \right) \\ &\approx \mathrm{GLWE}\big(-s_i(X)\mu(X) \cdot a_i(X) \big). \end{aligned}$$

Summing over i yields

$$C \star c \;\approx\; \mathrm{GLWE}\Big(b(X)\mu(X) - \sum_{i=1}^{k} s_i(X)\mu(X)a_i(X) \Big) = \mathrm{GLWE}\Big(\big(\langle \mathbf{a}, \mathbf{s} \rangle + m + e\big)\mu - \langle \mathbf{a}, \mathbf{s} \rangle \mu \Big),$$

so

$$C \star c \;\approx\; \mathrm{GLWE}\big(\mu(X) \cdot m(X) \;+\; \mu(X) \cdot e(X)\big).$$

Thus the external product produces a valid GLWE encryption of $\mu(X) \cdot m(X)$ with noise scaled by $\mu(X)$, which establishes correctness up to the usual approximation from decomposition [CGGI20, Joy22].

CMux Operation. The *conditional multiplexer* (CMux) is a fundamental homomorphic primitive in TFHE, enabling one to select between two encrypted values d_0 and d_1 under the control of an encrypted bit b [CGGI20]. Formally, the output is

$$\mathrm{CMux}(b, d_0, d_1) \;=\; (1 - b) \cdot d_0 \;+\; b \cdot d_1,$$

which evaluates to d_0 if $b = 0$ and d_1 if $b = 1$.

In practice, the control bit b is encrypted as a TGGSW ciphertext $B = \mathrm{TGGSW}(b)$, while d_0 and d_1 are GLWE ciphertexts. The CMux can then be implemented using the external product:

$$\mathrm{CMux}(B, d_0, d_1) \;=\; B \star (d_1 - d_0) \;+\; d_0,$$

where $\star$ denotes the TGGSW–GLWE external product. Intuitively, if $b = 0$, the external product term vanishes and the result is d_0; if $b = 1$, the external product adds $(d_1 - d_0)$, yielding d_1.

This construction is central in TFHE bootstrapping and gate evaluation: it allows encrypted bits to control the flow of computation, effectively implementing encrypted conditional branching entirely within the ciphertext domain.

3.1.4 Bootstrapping Primitives

Key Switching. Key switching is a fundamental mechanism in TFHE that allows a ciphertext encrypted under one secret key $\mathbf{s}$ to be transformed into an equivalent ciphertext under a different secret key $\mathbf{s}'$, without decrypting it [CGGI20]. This is crucial after operations such as sample extraction or bootstrapping, where ciphertexts naturally shift between different key domains.

Formally, the key switching key (KSK) consists of encryptions of the old secret key digits under the new secret key. Conceptually, each row of the KSK can be viewed as a GLev encryption of a secret key coefficient s_i, i.e.

$$\mathrm{KSK}[i] \;\approx\; \mathrm{GLev}(s_i),$$

where the GLev levels correspond to different decomposition digits. More precisely, for each coefficient s_i of the old key and for each gadget digit q/B^j with $j = 1, \ldots, \ell$, the KSK stores

$$\mathrm{KSK}[i, j] = \mathrm{GLWE}_{\mathbf{s}'}\left(\tfrac{q}{B^j} \cdot s_i\right).$$

To switch a ciphertext $c = (\mathbf{a}, b)$ under $\mathbf{s}$ into a ciphertext c' under $\mathbf{s}'$, each component a_i is first decomposed:

$$\mathrm{Decomp}^{B,\ell}(a_i) = (d_1^{(i)}, \ldots, d_\ell^{(i)}).$$

Then, using the KSK, we homomorphically compute

$$\sum_{i=1}^{n} \sum_{j=1}^{\ell} d_j^{(i)} \cdot \mathrm{KSK}[i, j],$$

which yields an encryption of $\sum_i a_i s_i$ under the new key $\mathbf{s}'$. Finally, we set

$$c' = \big(-\mathbf{a}',\; b - \langle \mathbf{a}, \mathbf{s} \rangle\big),$$

where $\mathbf{a}'$ and the adjusted term in b come from the accumulated KSK contributions. Intuitively, the KSK homomorphically produces an encryption of $\langle \mathbf{a}, \mathbf{s} \rangle$ under the new key $\mathbf{s}'$. By subtracting this quantity from b, we remove the dependence on the old key $\mathbf{s}$ and replace it with the corresponding term under $\mathbf{s}'$. At the same time, $\mathbf{a}'$ is negated to preserve the standard GLWE/LWE ciphertext form $(\mathbf{a}', b')$ where $b' = \langle \mathbf{a}', \mathbf{s}' \rangle + m + e$. This ensures that c' decrypts correctly under $\mathbf{s}'$ to the same plaintext message as the original ciphertext c, up to noise growth [CGGI20]..

Thus, key switching provides a seamless way to migrate ciphertexts across different secret keys, ensuring composability of homomorphic operations while keeping noise under control.

Sample Extraction. Sample extraction is the procedure of converting a GLWE ciphertext of degree N into an LWE ciphertext by isolating a single coefficient of the encrypted polynomial. Given a GLWE ciphertext

$$c = (\mathbf{a}(X), b(X)) \in \big(\mathbb{Z}_q[X]/(X^N+1)\big)^k \times \mathbb{Z}_q[X]/(X^N+1),$$

which encrypts a polynomial message $m(X)$ under secret key $\mathbf{s}(X) = (s_1(X), \ldots, s_k(X))$, the *sample extraction* operator at position $p \in \{0, \ldots, N-1\}$ produces an LWE ciphertext

$$\mathrm{SampleExtract}_p(c) = (\mathbf{a}', b') \in \mathbb{Z}_q^{n'} \times \mathbb{Z}_q,$$

where $b' = b_p$ is the p-th coefficient of $b(X)$ and $\mathbf{a}'$ is formed from a linear combination of the coefficients of each $a_i(X)$, adjusted for the cyclotomic relation $X^N + 1 = 0$ [CGGI20]. This guarantees that

$$b' - \langle \mathbf{a}', \mathbf{s}' \rangle \;\approx\; m_p + e,$$

so that the extracted ciphertext encrypts exactly the coefficient m_p of the original polynomial.

The vector $\mathbf{a}'$ is constructed by rearranging the coefficients of the polynomials $a_i(X)$ into length-n' vectors, with sign adjustments applied for indices that wrap around the relation $X^N + 1 = 0$. This ensures consistency with the LWE secret key format and allows the extracted ciphertext to decrypt correctly.

Sample extraction is a key subroutine in bootstrapping: after blind rotation produces a GLWE encryption of the refreshed message, one coefficient is extracted back into an LWE ciphertext, restoring the scheme's original format for further homomorphic operations [CGGI20]. Since the output is now an LWE ciphertext of dimension $n' = k \cdot N$, the corresponding GLWE secret key $\mathbf{s}(X) = (s_1(X), \ldots, s_k(X))$ must be flattened into an LWE secret vector consisting of all coefficients of the polynomials $s_i(X)$:

$$\mathbf{s}' = (s_{1,0}, s_{1,1}, \ldots, s_{1,N-1},\ s_{2,0}, \ldots, s_{k,N-1}) \in \mathbb{Z}_2^{n'}.$$

This flattened key ensures that the extracted ciphertext decrypts correctly under the LWE format as

$$b' - \langle \mathbf{a}', \mathbf{s}' \rangle \;\approx\; m_p + e.$$

Modulus Switching. Modulus switching is the operation of transforming a ciphertext encrypted under a larger modulus q into an equivalent ciphertext under a smaller modulus q', while preserving decryption correctness up to rounding errors [BGV11, CGGI20]. Given a ciphertext

$$c = (\mathbf{a}, b) \in \mathbb{Z}_q^n \times \mathbb{Z}_q,$$

the switched ciphertext is obtained by scaling each coefficient by the ratio q'/q and rounding to the nearest integer:

$$\mathbf{a}' = \left(\left\lfloor \tfrac{q'}{q} \cdot \mathbf{a} \right\rceil\right) \bmod q', \qquad b' = \left(\left\lfloor \tfrac{q'}{q} \cdot b \right\rceil\right) \bmod q'.$$

The resulting ciphertext

$$c' = (\mathbf{a}', b') \in \mathbb{Z}_{q'}^n \times \mathbb{Z}_{q'}$$

decrypts to the same plaintext as c, with proportionally scaled noise. This technique is fundamental in many homomorphic encryption schemes to reduce noise amplitude relative to the modulus.

Blind Rotation. Blind rotation is the central step in TFHE bootstrapping, in which an input LWE ciphertext is used to control the cyclic rotation of an encrypted polynomial accumulator without revealing the underlying message [CGGI20].

Step 1: Initial rotation by $-b$. An accumulator polynomial $V(X)$ is initialized. This $V(X)$, often called the *test polynomial*, encodes the lookup table (LUT) for the function to be bootstrapped. Multiplying $V(X)$ by X^r corresponds to rotating its coefficients cyclically by r positions (with wrap-around defined by $X^N \equiv -1$). The constant term b of the ciphertext is first used to perform a rotation by X^{-b}:

$$V(X) \longmapsto X^{-b} \cdot V(X).$$

Step 2: Controlled rotations by the mask coefficients. Each component a_i of the LWE mask $\mathbf{a}$ is then processed. For each a_i, the algorithm computes a candidate rotation $X^{a_i} \cdot V(X)$ and uses a conditional multiplexing (CMux) gate to select between the original and rotated accumulators. The control signal for this selection is the encrypted secret key bit s_i, provided in the form of a TGGSW ciphertext $\mathrm{TGGSW}(s_i)$. In other words, each s_i is itself encrypted as a TGGSW, which makes it possible to apply the rotation without knowing the value of s_i:

$$\mathrm{Acc} \longmapsto \mathrm{CMux}\big(\mathrm{TGGSW}(s_i),\ \mathrm{Acc},\ X^{a_i} \cdot \mathrm{Acc}\big).$$

Intuitively, if $s_i = 0$ the CMux outputs Acc unchanged, while if $s_i = 1$ it outputs the rotated version $X^{a_i} \cdot \mathrm{Acc}$. Because s_i is only available in encrypted TGGSW form, the evaluator never learns the secret, yet the accumulator is rotated correctly according to the hidden key.

Step 3: Resulting blind rotation. After processing all mask coefficients, the final accumulator has been rotated by a total amount

$$-b + \sum_{i=1}^{n} a_i s_i,$$

which is exactly the phase of the original LWE ciphertext. This arises because each CMux step conditionally multiplies the accumulator by X^{a_i}: if $s_i = 0$, no rotation is applied, and if $s_i = 1$, a rotation by a_i is applied. The product of these conditional rotations therefore accumulates as $X^{\sum_i a_i s_i}$. Combined with the initial X^{-b} shift, the net effect is a rotation by $X^{-b+\sum_i a_i s_i}$, reproducing the encrypted phase of c.

Thus the blind rotation encodes the encrypted message m as a hidden rotation of the test polynomial $V(X)$. The evaluator never learns the rotation index, but the accumulator now carries the refreshed information in polynomial form, ready for sample extraction.

3.1.5 TFHE Bootstrapping

Bootstrapping is the process of *refreshing* a noisy ciphertext by homomorphically evaluating (a variant of) the decryption procedure and re-encrypting the result, thereby restoring a large noise margin and enabling arbitrarily many subsequent operations [Gen09, CGGI20]. In TFHE, this refresh is performed together with a function evaluation via a *lookup table (LUT)* encoded as a polynomial $V(X) \in \mathbb{Z}_q[X]/(X^N + 1)$. A blind rotation uses the input ciphertext to rotate $V(X)$ without revealing the rotation amount; then a single coefficient is extracted (LWE again) and switched back to the target secret key domain. Thus TFHE combines noise reduction and functional evaluation in a single primitive [CGGI20].

Regular Bootstrapping. Let $c = (\mathbf{a}, b) \in \mathbb{Z}_q^n \times \mathbb{Z}_q$ be an LWE ciphertext under secret key $\mathbf{s}$, encrypting a message $m \in \mathbb{Z}_p$ with phase

$$\varphi(c) = b - \langle \mathbf{a}, \mathbf{s} \rangle \equiv \Delta m + e \pmod{q}, \qquad \Delta = \tfrac{q}{p}.$$

TFHE implements bootstrapping with the following steps [CGGI20]:

(1) Modulus switch to $2N$. Map the phase to an integer rotation index

$$d \;\approx\; \left\lfloor \tfrac{2N}{q}\, \varphi(c) \right\rceil \;\in\; \{0, \ldots, 2N-1\}.$$

This makes the encrypted phase directly interpretable as a power of X in $\mathbb{Z}_q[X]/(X^N+1)$, i.e., a cyclic coefficient rotation.

(2) Initialize the LUT polynomial. The next step is to initialize the *test/LUT polynomial* $V(X)$. Its role is to act as a lookup table implementing the *identity function* on the torus: after the appropriate blind rotation, the constant coefficient of $V(X)$ will recover the encrypted phase $\Delta m + e$. To achieve this, $V(X)$ is defined so that its j-th coefficient encodes the discretized value $\frac{1}{p}\lfloor p \cdot j/(2N) \rceil \pmod 1$:

$$V(X) = \sum_{j=0}^{N-1} \left(\frac{\lfloor p\, j/(2N) \rceil \bmod p}{p} \right) X^j. \tag{3.1}$$

Operationally, we store the integer numerators $v_j := \lfloor p\, j/(2N) \rceil \bmod p$ and embed them into $\mathbb{Z}_q$ by scaling with Δ: the accumulator is initialized as the GLWE encryption of the polynomial with coefficients $\Delta \cdot v_j$ (torus encoding).

This construction has the effect of mapping the coefficient index j to the torus fraction $j/(2N) \bmod 1$, quantized with granularity $1/p$. In other words, $V(X)$ contains a discretized "sawtooth" sequence that, when shifted by the encrypted rotation index during blind rotation, aligns its constant term to output precisely (up to discretization) the input phase. This is why $V(X)$ is often described as the *identity LUT polynomial*: it does not alter the message, but instead serves to reconstruct it at the correct position after blind rotation [CGGI20].

(3) Blind Rotation. The input ciphertext $c = (\mathbf{a}, b)$ is then used to rotate the LUT polynomial $V(X)$ by an amount corresponding to its hidden phase. Concretely, the accumulator is first rotated by X^{-b}, and then for each mask coefficient a_i, a CMux gate controlled by a TGGSW encryption of s_i conditionally applies the rotation X^{a_i}:

$$\mathrm{Acc} \;\leftarrow\; \mathrm{CMux}\big(\mathrm{TGGSW}(s_i),\ \mathrm{Acc},\ X^{a_i} \cdot \mathrm{Acc}\big).$$

After processing all mask components, the accumulator has been rotated by the total amount

$$X^{-b+\sum_i a_i s_i} \;=\; X^{-\,(b - \langle \mathbf{a}, \mathbf{s} \rangle)} \;=\; X^{-\varphi(c)} \pmod{X^N+1,\, 2N}.$$

Since multiplying by X^r rotates coefficients cyclically by r (with wrap-around from $X^N \equiv -1$), the constant coefficient of the rotated polynomial becomes the d-th coefficient of the original $V(X)$, where $d \approx \lfloor (2N/q)\, \varphi(c) \rfloor$.

By construction of $V(X)$ (Equation (3.1)), the d-th coefficient equals

$$\Delta \cdot v_d \;=\; \Delta \cdot \left\lfloor \frac{p}{2N}\, d \right\rceil \;\approx\; \Delta \cdot \left\lfloor \frac{p}{q}\, \varphi_q \right\rceil \;\approx\; \varphi_q \;=\; \Delta m + e \pmod{q},$$

where φ_q is the integer representative of the phase. Thus the blind-rotated accumulator has, in its constant term, an encoding of the original phase $\Delta m + e$ in an encrypted form, up to a small quantization error of at most $\Delta/2$, plus the noise from CMux and external-product operations. This value is exactly what is needed to refresh the ciphertext in the subsequent extraction step.

(4) Sample extraction. Convert the rotated GLWE into an LWE by extracting the constant coefficient (position 0). This yields an LWE ciphertext whose phase is (approximately) $\Delta m + e$ as above. Extraction is purely algebraic and does not itself add noise [CGGI20].

(5) Key switching. The extracted LWE is under the *flattened* GLWE key $\mathbf{s}' \in \{0,1\}^{kN}$ (concatenation of the GLWE key coefficients). Using a key switching key, we transform it into an LWE under the target key (typically the original LWE key), obtaining a refreshed ciphertext encrypting the same message m with small noise [CGGI20].

In summary, the LUT $V(X)$ in (3.1) converts the *encrypted* rotation index d back into an approximation of the original phase, so that, after sample extraction and key switching, TFHE outputs an LWE encryption of m with reduced noise. This is the bootstrapping refresh, and by changing $V(X)$ we can simultaneously apply a function to m.

Programmable Bootstrapping. The key insight of TFHE is that the LUT polynomial $V(X)$ need not encode the identity function. By replacing (3.1) with a polynomial whose j-th coefficient encodes the value of an arbitrary function f evaluated on the discretized input point $\frac{\lfloor pj/(2N) \rceil \bmod p}{p}$, we can program the bootstrap to compute $f(m)$:

$$V(X) = \sum_{j=0}^{N-1} f\left(\frac{\lfloor pj/(2N) \rceil \pmod p}{p}\right) X^j.$$

As before, these coefficients are embedded in the torus by multiplying with $\Delta = q/p$. During blind rotation, the encrypted input c selects the appropriate coefficient of $V(X)$, and after sample extraction the result is an LWE ciphertext encrypting $f(m)$ (up to discretization and noise).

This mechanism is known as *programmable bootstrapping*: the bootstrap not only refreshes the ciphertext to reduce noise, but simultaneously evaluates an arbitrary function f [CGGI20].

Gate Bootstrapping. As a special case, setting $V(X)$ to the truth table of a Boolean gate (e.g., AND, OR, NAND, XOR) implements the gate on encrypted inputs while refreshing noise—*gate bootstrapping*. This yields extremely fast private logic (few milliseconds per gate) and underpins TFHE's strength for encrypted control-flow and bitwise computation [CGGI20].

4 Advanced Scheme Switching

A primary use case for our prototyping sandbox is to model and verify advanced functionalities such as scheme switching. A particularly important application of this is transciphering, where a client using a lightweight symmetric encryption scheme sends data to a server, which then homomorphically decrypts it into a fully homomorphic ciphertext format (e.g., CKKS or TFHE) for heavy computation. This is highly desirable in scenarios where

the client is a resource-constrained device (e.g., an IoT sensor) that cannot afford the computational cost of public-key FHE encryption.

4.1 The Ring Packing Problem

The core technical challenge in transciphering from an LWE-based symmetric cipher to an RLWE-based HE scheme is an operation known as ring packing: given N LWE ciphertexts $\{(\mathbf{a}_i, b_i)\}_{i=0}^{N-1}$, each encrypting a message m_i under a secret key $\mathbf{s} \in \mathbb{Z}_q^N$, the goal is to efficiently construct a single RLWE ciphertext that encrypts the polynomial $M(X) = \sum_{i=0}^{N-1} m_i X^i$.

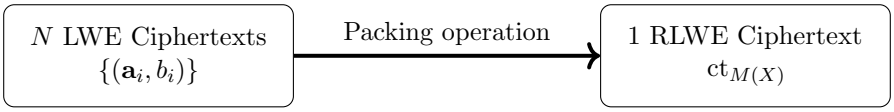

Figure 2: The Ring Packing Problem.

From the LWE decryption equation, we have $m_i = b_i - \langle \mathbf{a}_i, \mathbf{s} \rangle = b_i - \sum_{j=0}^{N-1} a_{ij} s_j \pmod q$. Substituting this into the target polynomial $M(X)$ gives:

$$M(X) = \sum_{i=0}^{N-1} \left(b_i - \sum_{j=0}^{N-1} a_{ij} s_j \right) X^i$$

By swapping the order of summation, we can isolate the secret key components s_j:

$$M(X) = \sum_{i=0}^{N-1} b_i X^i - \sum_{j=0}^{N-1} s_j \left(\sum_{i=0}^{N-1} a_{ij} X^i \right)$$

This reveals the algebraic goal. Let $\beta(X) = \sum_{i=0}^{N-1} b_i X^i$ and $\alpha_j(X) = \sum_{i=0}^{N-1} a_{ij} X^i$. Then we must compute:

$$M(X) = \beta(X) - \sum_{j=0}^{N-1} s_j \cdot \alpha_j(X)$$

The polynomials $\beta(X)$ and $\alpha_j(X)$ are public, constructed from the LWE ciphertexts. The core task is to compute the weighted sum of the public polynomials $\alpha_j(X)$ with the secret scalars s_j homomorphically.

4.2 The Column Method and its Bottleneck

A direct approach is the column method. The server's task is to compute the sum $\beta(X) + \sum_j s_j \alpha_j(X)$ homomorphically, without learning the secret scalars s_j. A naive way to accomplish this is to:

1. For each secret scalar s_j, the client generates an RLWE encryption of the scalar s_j.
2. The server computes the plaintext-ciphertext product for each j: $\mathrm{ct}_j = \alpha_j(X) \odot \mathrm{Enc}(s_j) = \mathrm{Enc}(\alpha_j(X) \cdot s_j)$.
3. The server adds $\beta(X)$ to the sum of the N resulting ciphertexts $\sum_j \mathrm{ct}_j$.

A more precise alternative uses key switching. The client generates a key-switching key swk_j for each scalar s_j. The server then computes $\text{KS}_{\text{swk}_j}(\text{ct}_j)$ for a ciphertext constructed from $\beta(X)$ and $\alpha_j(X)$, and sums the results.

The primary bottleneck of this method is not arithmetic, but memory. Each key-switching key swk_j is a full degree-N RLWE ciphertext, a large cryptographic container of size $\approx 2N \log q$ bits, used to encrypt just one secret scalar. For N secrets, the total key size is $N \times$ (size of one key), which for typical parameters like $N = 2^{10}$ and $\log q \approx 60$ can easily reach hundreds of megabytes or gigabytes, making the approach impractical. The goal of modern methods like HERMES is to pack multiple secret scalars into the same container, thus reducing the total number of keys from $\mathcal{O}(N)$ to $\mathcal{O}(\sqrt{N})$.

4.3 The HERMES System Architecture

The HERMES method [BCK+23] optimizes ring packing by composing several key insights into a three-stage "Map-Gather-Lift" pipeline, shown in Figure 3.

1. **Map (Parallel Base Packing):** The large packing problem of N_{BTS} LWE ciphertexts is broken down into N_{BTS}/N_{KS} smaller, independent packing problems. Each of these is solved in parallel using a memory-efficient algorithm called `BaseHERMES`.

2. **Gather (Ring Switching):** The multiple small-degree RLWE ciphertexts from the map stage are efficiently combined into a single large-degree RLWE ciphertext using a nearly-free 'Ring Switch' operation, which is essentially an algebraic change of representation.

3. **Lift (Moduli Optimization):** The expensive, unstructured packing computation is performed in a "cheap" low-modulus environment (Q_{Enc}). Then, the highly-optimized bootstrapping procedure (`HalfBTS`) is used to "lift" the final result to the "expensive" high-modulus FHE environment (Q_{comp}) where further computations can take place. This leverages the fact that bootstrapping can change the ciphertext modulus as part of its noise-clearing operation.

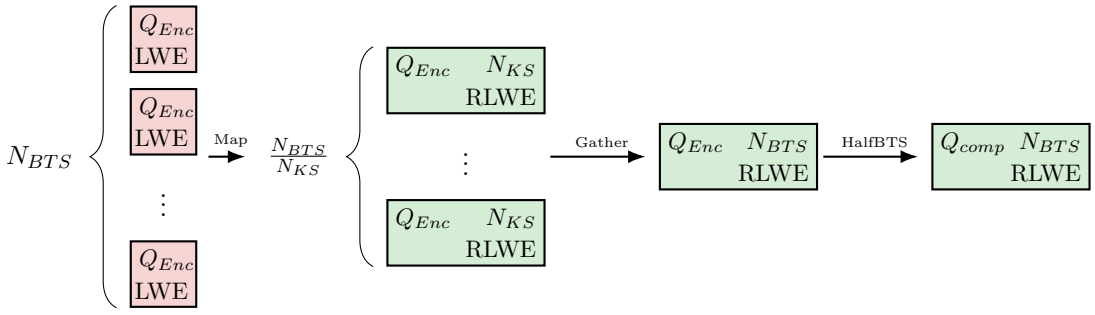

Figure 3: The HERMES "Map-Gather-Lift" Pipeline.

4.4 The Algebraic Foundations of HERMES

The memory efficiency of HERMES stems from re-interpreting the problem using the algebra of Module-LWE (MLWE) schemes. An MLWE scheme of rank k and degree n has secrets that are vectors of k polynomials in $\mathcal{R}_{q,n}$. This allows a block of k LWE secrets to be treated as a single MLWE secret, which can be encrypted into a single, more compact key.

To facilitate this, HERMES defines a toolkit of algebraic operations. A key operation is the Ring Switch (or Combine), which takes k small-degree RLWE ciphertexts and interleaves their coefficients to form one large-degree ciphertext. Let $\{p_0(X), \ldots, p_{k-1}(X)\}$ be k polynomials in the small ring $\mathcal{R}_{q,n}$. The operation constructs a single polynomial $P(Y)$ in the large ring $\mathcal{R}_{q,N}$ (where $N = nk$) as follows:

$$P(Y) = \sum_{i=0}^{k-1} p_i(Y^k) \cdot Y^i$$

This is a structured memory copy, not a computation. It adds no noise and has negligible cost.

The core engine, `ModPack`, generalizes the column method to modules. It takes $m = k/k'$ input MLWE ciphertexts of rank K and produces a single MLWE ciphertext of rank k'. The key insight is to partition the summation over the secret components into blocks of size k', allowing a single MLWE key-switch to be used for each block, thereby reducing the number of required keys. `BaseHERMES` is a hierarchical application of `ModPack` that recursively reduces the rank of the problem, achieving the asymptotic reduction in key memory. The correctness of these procedures relies on a set of algebraic tools (`Embed`, `Extract`, `twist`) that map between MLWE and RLWE representations, allowing standard RLWE key-switching to be used on MLWE objects. The `twist` operation is a specific permutation that ensures a polynomial convolution in the RLWE space correctly computes a module inner product in the MLWE space.

Table 1: Comparison of HERMES variants for packing 2^{12} ciphertexts [BCK+23].

Variant	**Latency (s)**	**Total key size (MB)**
HERMES (column method)	29.1	782
HERMES (mid-point method)	30.7	673

5 Homomorphic Matrix Multiplication Benchmarks

Matrix multiplication lies at the heart of modern computation, spanning scientific simulations, computer vision, natural language processing, and large-scale data analytics. In machine learning, nearly every model from linear regression to deep neural networks relies on multiplying large matrices and vectors. As such, accelerating matrix multiplication has been a core driver of hardware innovation, from GPUs to dedicated tensor accelerators.

In the context of homomorphic encryption (HE), matrix multiplication becomes even more critical. Homomorphic schemes allow computations to be performed directly on encrypted data without revealing sensitive information. This property is highly desirable for applications such as privacy-preserving medical research, encrypted financial analytics, and secure outsourced machine learning. However, these advantages come with costs. Operations that are trivial in the plaintext world become computationally expensive in the encrypted domain due to noise growth, ciphertext expansion, and the heavy algebraic structures underpinning schemes like CKKS and TFHE.

Matrix multiplication intensifies these challenges. It requires a large number of repeated additions, multiplications, and rotations on ciphertexts. A single encrypted matrix–vector product can involve thousands of costly homomorphic operations. When scaled to real-world workloads, this overhead becomes a bottleneck that prevents HE from being deployed widely

in practice. Benchmarks consistently show that even optimized encrypted multiplications can be much slower than their plaintext counterparts.

This motivates the central focus of our project: optimizing matrix multiplication under homomorphic encryption. By systematically evaluating encoding strategies and algorithmic techniques, we aim to reduce the computational overhead and identify pathways toward GPU acceleration. In particular, we explore Microsoft SEAL's implementation of the CKKS scheme, using its SIMD capabilities to batch operations across multiple plaintext slots. Our benchmarks establish baselines for standard row-wise packing and the Halevi–Shoup diagonal method, while future work will extend to more advanced techniques such as Baby-Step Giant-Step and Bicycle encoding. These optimizations are not merely theoretical exercises; they represent the practical foundation required to make privacy-preserving machine learning feasible at scale.

In summary, matrix multiplication is both a cornerstone of modern computation and a bottleneck in encrypted computation. Addressing its inefficiency is therefore not just a technical curiosity but a necessity for advancing secure, real-world applications of homomorphic encryption.

5.1 CKKS Scheme and Microsoft SEAL

One of the most widely used homomorphic encryption schemes for real-valued computations is CKKS (Cheon–Kim–Kim–Song). Unlike schemes designed strictly for exact integer arithmetic, CKKS is optimized for approximate arithmetic, making it particularly suitable for machine learning, data analytics, and signal processing tasks where a small loss of precision is acceptable. The key idea of CKKS is that it encodes a vector of real or complex numbers into a polynomial over a large modular ring. The encryption process then produces ciphertexts that behave like noisy versions of these vectors. Homomorphic operations such as addition and multiplication can be applied directly to ciphertexts, and the results correspond to approximate operations on the underlying plaintext vectors. The scheme also enables SIMD-style operations (Single Instruction, Multiple Data): a single homomorphic operation can simultaneously act on an entire vector of packed values. This property makes CKKS especially powerful for parallelizable tasks such as matrix multiplication. However, these advantages come with computational challenges. Each homomorphic multiplication consumes a portion of the available noise budget, and repeated operations require rescaling and rotation steps. These operations are among the most expensive in HE, and optimizing them is crucial for making large computations feasible.

To implement CKKS in practice, we used Microsoft SEAL, an open-source C++ library developed by Microsoft Research. SEAL is one of the most mature and widely adopted homomorphic encryption libraries, supporting both BFV (for exact integer arithmetic) and CKKS (for approximate arithmetic). It provides a carefully engineered environment with modular arithmetic, efficient polynomial operations, and built-in optimizations for memory management and parallelism. For our project, SEAL served as the backbone for benchmarking different matrix multiplication methods. Its robust support for CKKS allowed us to encode matrix rows, perform homomorphic multiplications, and measure the performance cost of rotations and rescaling operations. This benchmark not only gave us concrete performance data but also provided a baseline for GPU performance. By first understanding the performance profile in SEAL, we can identify which algorithms benefit most from GPU parallelization and which remain bottlenecks. In summary, the CKKS scheme offers a practical balance between correctness and efficiency for encrypted computations, and Microsoft SEAL provides the tools necessary to experiment with real-world implementations. Together, they form the foundation of our work on optimizing

encrypted matrix multiplication.

5.2 Packing Strategies

The main challenge lies in how data is packed into ciphertexts. In CKKS, a single ciphertext can hold a vector of $N/2$ real numbers. Clever packing strategies can dramatically reduce the number of expensive homomorphic operations by maximizing the amount of useful computation performed by each operation.

Row-Wise Packing and SIMD Operations One of the simplest approaches for encrypted matrix multiplication under CKKS is row-wise encoding. To compute the product of an $n \times n$ matrix A and a vector $\mathbf{x}$, each row $\mathbf{a}_i = (a_{i,0}, \ldots, a_{i,n-1})$ of the matrix is encoded into a separate plaintext polynomial. Let $\mathrm{Enc}(\cdot)$ denote the CKKS encryption operation and $\odot$ denote homomorphic component-wise multiplication. The vector $\mathbf{x}$ is encrypted into a single ciphertext $\mathrm{ct}_{\mathbf{x}} = \mathrm{Enc}(\mathbf{x})$. For each row i, the server computes the component-wise product:

$$\mathrm{ct}'_i = \mathrm{Encode}(\mathbf{a}_i) \odot \mathrm{ct}_{\mathbf{x}}$$

This operation results in a ciphertext ct'_i that encrypts the vector $(a_{i,0}x_0, a_{i,1}x_1, \ldots, a_{i,n-1}x_{n-1})$. To obtain the dot product $\langle \mathbf{a}_i, \mathbf{x} \rangle$, all elements in the encrypted vector must be summed. This is achieved by a sequence of homomorphic rotations and additions. Let $\mathrm{Rot}(\mathrm{ct}, k)$ denote a cyclic shift of the plaintext vector in ciphertext ct by k positions. The sum is computed as:

$$\mathrm{ct}_{\mathrm{sum},i} = \sum_{j=0}^{n-1} \mathrm{Rot}(\mathrm{ct}'_i, j)$$

The first slot of the resulting ciphertext $\mathrm{ct}_{\mathrm{sum},i}$ contains the desired inner product. This process is repeated for all n rows of the matrix.

- **Advantages:**
  1. *Conceptual Simplicity:* Easy to implement and aligns well with the native SIMD capabilities of CKKS.
  2. *Good for Dense Matrices:* When most entries are nonzero, row-wise packing minimizes wasted ciphertext space.
- **Limitations:**
  1. *Expensive Rotations:* To align intermediate results (e.g., shifting slots for summation), many ciphertext rotations are required. These are among the most expensive operations in HE.
  2. *Inefficient for Sparse Matrices:* If many entries in the matrix are zero, ciphertext slots are wasted.

Despite these drawbacks, row-wise encoding is an important baseline method. It serves as the natural starting point for benchmarking more advanced algorithms, such as the Halevi–Shoup diagonal method and Bicycle encoding. In our project, we measured the operation count for row-wise multiplication and found that the method incurred significantly higher numbers of additions and rotations compared to optimized alternatives. These findings highlight why optimizing encoding strategies is critical for making homomorphic matrix multiplication practical.

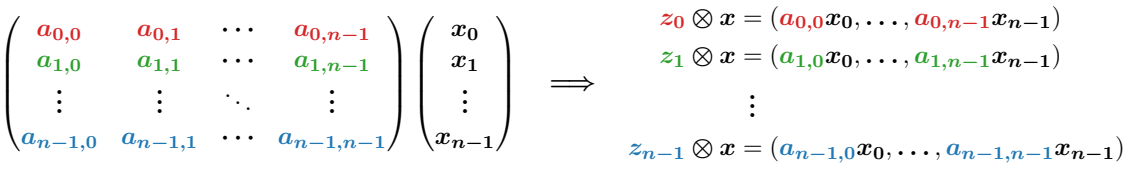

Figure 4: Row-wise packing for matrix-vector multiplication.

Halevi-Shoup (Diagonal) Method The Halevi–Shoup method reorganizes the matrix multiplication to minimize rotations. Instead of packing rows, it packs the diagonals of the matrix. For an $n \times n$ matrix A, the k-th diagonal is the vector $\mathbf{d}_k = (a_{0,k}, a_{1,k+1}, \ldots, a_{n-1,k-1})$, where indices are taken modulo n. Each of the n diagonals is encoded into a plaintext. The encrypted vector $\mathrm{ct}_{\mathbf{x}}$ is rotated k times for each diagonal. The i-th component of the result vector $\mathbf{y} = A\mathbf{x}$ is given by $y_i = \sum_{j=0}^{n-1} a_{i,j} x_j$. By rearranging the sum, we can express the entire result vector $\mathbf{y}$ as a sum over the diagonals:

$$\mathbf{y} = \sum_{k=0}^{n-1} \mathbf{d}_k \odot \mathrm{Rot}(\mathbf{x}, k)$$

This formulation translates directly into an efficient homomorphic algorithm. The server computes:

$$\mathrm{ct}_{\mathbf{y}} = \sum_{k=0}^{n-1} (\mathrm{Encode}(\mathbf{d}_k) \odot \mathrm{Rot}(\mathrm{ct}_{\mathbf{x}}, k))$$

This requires n homomorphic multiplications and $n-1$ rotations in total, a significant improvement over the $\mathcal{O}(n \log n)$ rotations required by the row-wise method.

- **Advantages:**
  1. *Fewer Rotations and Additions:* Diagonal packing minimizes slot shifts needed for summation.
  2. *High Efficiency for Dense Matrices:* Well-suited when the entire matrix is used repeatedly in computations.
- **Limitations:**
  1. *Implementation Complexity:* Packing diagonals requires careful indexing and setup.
  2. *Less Flexible for Sparse Data:* Benefits are maximized for dense, structured matrices rather than highly sparse ones.

In our benchmarks, the Halevi–Shoup method required an order of magnitude fewer additions and rotations than row-wise packing, while keeping the same number of multiplications and rescaling steps. For example, when multiplying an $n \times n$ encrypted matrix with a vector, the Halevi–Shoup method achieved approximately 4,095 additions and 4,095 rotations, compared to 53,248 additions and 49,152 rotations with row-wise packing. The Halevi–Shoup method forms the optimized baseline for encrypted matrix multiplication in our study. It outperforms the naive row-wise approach and demonstrates that algorithmic reorganization of data can yield significant speedups, even before hardware acceleration is introduced.

Baby-Step Giant-Step (BSGS) The Baby-Step Giant-Step approach is a time-space tradeoff method. It decomposes the computation into small "baby steps" that can be

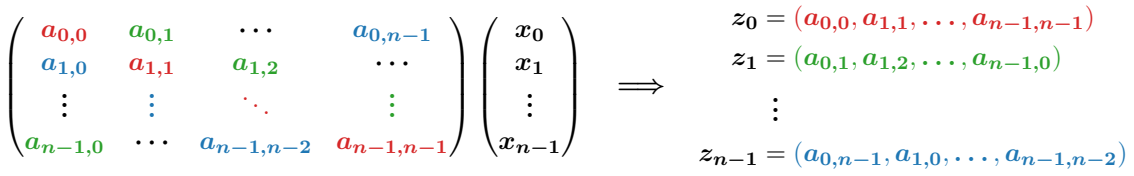

Figure 5: Halevi-Shoup (diagonal) packing for matrix-vector multiplication.

precomputed and stored, and larger "giant steps" that combine these results during the main computation.

- **Advantages:**
  1. *Reduces the number of expensive homomorphic rotations.*
  2. *Useful for large-scale problems where repeated structure can be exploited.*
- **Limitations:**
  1. *Requires significant memory for storing baby-step precomputations.*
  2. *Setup overhead can be heavy if the computation is not reused.*

This technique could yield speedups in encrypted linear algebra tasks, provided that memory constraints are managed carefully.

Bicycle Encoding Bicycle Encoding is a specialized data layout that converts a matrix into a vector, enabling a natural traversal for matrix multiplication. It utilizes the Chinese Remainder Theorem for conversion and traversing over the matrix.

- **Advantages:**
  1. *Native traversal nature enables us to skip unnecessary operations.*
  2. *Makes indexing easier by auto-aligning the elements that are meant to multiply.*
- **Limitations:**
  1. *Expensive encoding step: unlike row-wise and Halevi-Shoup method, encoding is expensive in Bicycle encoding.*
  2. *Not ideal for sparse matrices.*
  3. *Requires co-prime dimensions to work perfectly.*

5.3 Complexity Comparison

The choice of packing strategy has a profound impact on the computational complexity. For an $n \times n$ matrix-vector multiplication:

- **Row-Wise Packing:** This method is the most expensive. To compute each of the n output elements, it first performs n element-wise multiplications. Then, for each of the n resulting vectors, it must perform a sum-all-elements operation, which requires $\log_2(n)$ rotations. Finally, it needs to mask and combine these n results. This leads to a complexity of $\mathcal{O}(n)$ multiplications and $\mathcal{O}(n \log n)$ rotations.
- **Halevi-Shoup Method:** This method is significantly more efficient. It requires n homomorphic multiplications (one for each packed diagonal) and $n-1$ homomorphic rotations to align and sum the intermediate results. The total complexity is $\mathcal{O}(n)$ multiplications and $\mathcal{O}(n)$ rotations.

- **Baby-Step Giant-Step Method:** This method optimizes the rotations of the Halevi-Shoup method. It still requires $\mathcal{O}(n)$ multiplications, but it reduces the number of online rotations to $\mathcal{O}(\sqrt{n})$. This comes at the cost of precomputation and storage, but for applications where the same matrix is used repeatedly, it can be a very effective optimization.

6 Experiments and Results

Our benchmarks provide a clear performance profile for matrix-vector multiplication. The Halevi-Shoup method demonstrated superior performance by reducing the number of costly additions and rotations compared to the naive row-packing method. This is a direct consequence of its SIMD-friendly data layout, which leverages the parallel nature of polynomial arithmetic in CKKS.

Table 2 highlights the difference in the number of homomorphic operations required. While the number of multiplications is the same, the Halevi-Shoup method reduces the number of additions and rotations by an order of magnitude.

Table 2: Homomorphic Operation Count for 64×64 Matrix-Vector Multiplication

Operation	**Halevi-Shoup**	**Row-wise**
Add	4,095	53,248
Rotate	4,095	49,152
Rescale	4,096	4,096
Multiply	4,096	4,096
Encoding	4,096	4,096

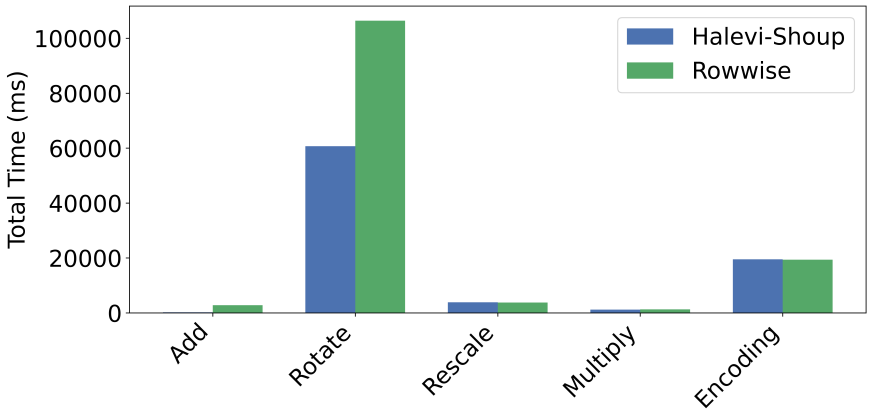

Figure 6: Execution Time Comparison (ms) for 64×64 Matrix-Vector Multiplication.

7 Conclusion and Future Direction

Our work establishes necessary utilities and groundwork for improving our GPU library HEonGPU.

- **Python Sandbox:** The Python implementation proved to be a vital tool for rapidly prototyping and validating advanced HE features like the HERMES scheme switching, ensuring our future GPU library is built on a solid, verified foundation of correct cryptographic logic.
- **C++ Benchmarking:** The comprehensive C++ benchmark provides a definitive performance profile of key matrix multiplication algorithms, identifying optimal strategies (like Halevi-Shoup) for GPU implementation and providing a concrete baseline against which to measure acceleration.

Future work will involve implementing the BSGS and Bicycle Encoding methods to complete our benchmark, followed by the porting of the most promising algorithms to our HEonGPU library. Rigorous quantification of the resulting acceleration will be a key outcome.

References

[BCK+23] Youngjin Bae, Jung Hee Cheon, Jaehyung Kim, Jai Hyun Park, and Damien Stehlé, *Hermes: Efficient ring packing using mlwe ciphertexts and application to transciphering*, Springer-Verlag, 2023.

[BGV11] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan, *Fully homomorphic encryption without bootstrapping*, Cryptology ePrint Archive, Paper 2011/277, 2011.

[CGGI20] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène, *TFHE: Fast fully homomorphic encryption over the torus*, Journal of Cryptology **33** (2020), no. 1, 34–91.

[CKKS17] Jung Hee Cheon, Andrey Kim, Miran Kim, and Yongsoo Song, *Homomorphic encryption for arithmetic of approximate numbers*, Advances in Cryptology - ASIACRYPT 2017, Springer, 2017, pp. 409–437.

[CYW+24] Jingwei Chen, Linhan Yang, Wenyuan Wu, Yang Liu, and Yong Feng, *Homomorphic matrix operations under bicyclic encoding*, Cryptology ePrint Archive, Paper 2024/1762, 2024.

[Gen09] Craig Gentry, *A fully homomorphic encryption scheme*, Ph.D. thesis, Stanford University, 2009.

[Gen10] Craig Gentry, *Computing arbitrary functions of encrypted data*, Commun. ACM **53** (2010), no. 3, 97–105.

[HS14] Shai Halevi and Victor Shoup, *Algorithms in helib*, Advances in Cryptology - CRYPTO 2014, Springer, 2014, pp. 554–571.

[Joy22] Marc Joye, *SoK: Fully homomorphic encryption over the [Discretized] torus*, IACR Transactions on Cryptographic Hardware and Embedded Systems **2022** (2022), no. 4, 661–692.

[LPR10] Vadim Lyubashevsky, Chris Peikert, and Oded Regev, *On ideal lattices and learning with errors over rings*, Advances in Cryptology - EUROCRYPT 2010, Springer, 2010, pp. 1–23.

[PKL23] Jai Hyun Park, Dong-gue Kim, and Young-Sik Lee, *Packed matrix-matrix multiplication from plaintext-matrix-vector multiplication in homomorphic encryption*, Information Security and Cryptology - ICISC 2022, Springer, 2023, pp. 3–23.

[Reg05] Oded Regev, *On lattices, learning with errors, random linear codes, and cryptography*, Proceedings of the Thirty-seventh Annual ACM Symposium on Theory of Computing, ACM, 2005, pp. 84–93.

Appendix: Benchmark Parameters

The parameters for the CKKS scheme used in the C++ benchmarks are summarized in Table 3. These were chosen to provide sufficient precision for the 64×64 matrix-vector multiplication experiments.

Table 3: CKKS Parameters for Benchmarks.

Parameter	**Value**
Scheme	CKKS
Polynomial Modulus Degree (N)	8192
Coefficient Modulus (q)	Primes with bit-lengths {50, 40, 40, 40, 40}
Plaintext Scale (Δ)	2^{40}

A API Reference for Python Sandbox

This appendix provides a formal reference for the core components of the Python implementation, which serves as the prototyping sandbox for this research.

A.1 Core TFHE Components

`TFHEParameters` Encapsulates cryptographic parameters: TGLWE rank (k) and degree (N), ciphertext modulus (q), plaintext modulus (t), gadget decomposition base (B) and length (l), and noise distribution (σ).

`TFHEEncoder` Handles encoding of integer messages to plaintext polynomials on the Torus via scaling by $\Delta = q/t$, and decoding back.

`TFHEKeyGenerator` Generates TGLWE secret keys $\mathbf{s} \in (\mathcal{R}_{\{0,1\}})^k$ and key-switching keys.

`TFHEEncryptor` Encrypts plaintexts into TGLWE or TGGSW ciphertexts under a given secret key.

`TFHEDecryptor` Decrypts TGLWE ciphertexts by computing the phase $m' = b + \langle \mathbf{a}, \mathbf{s} \rangle \pmod q$ and decoding.

`TFHEEvaluator` Contains logic for homomorphic operations such as addition, external product (for multiplication), and the homomorphic multiplexer (CMux).

A.2 SchemeSwitcher Class

Handles scheme switching operations from LWE to RLWE, implementing algorithms from the HERMES paper.

`column_method(lwe_ciphertexts, swk_list, ckks_evaluator)` Implements the basic column method for ring packing. It converts a list of N LWE ciphertexts into a single RLWE ciphertext using N corresponding switching keys.

`pack_using_ring_switching(lwe_ciphertexts, switching_keys, base_evaluator)` Implements the parallel packing strategy from HERMES. It breaks a large packing problem into smaller, parallel problems and combines the results using coefficient interleaving (`_combine`).

`modpack(mlwe_ciphertexts, switching_keys, ...)` Implements the core ModPack algorithm. It takes a list of MLWE ciphertexts of rank K and packs them into a single MLWE ciphertext of a smaller rank k', using a reduced number of switching keys.

`basehermes(lwe_ciphertexts, switching_keys_map, ...)` Implements the multi-stage `BaseHERMES` algorithm by recursively applying ModPack. It uses MLWE midpoints to significantly reduce the total switching key size.

`hermes(lwe_ciphertexts, basehermes_keys, ...)` Orchestrates the full HERMES packing procedure. It manages parallel calls to `basehermes` for different batches of LWE ciphertexts and combines their results into a final, large-degree RLWE ciphertext.

B API Reference for C++ Matrix Multiplication Benchmark

This appendix provides a reference for the core C++ functions developed for the matrix multiplication benchmark using the Microsoft SEAL library.

B.1 Row-wise Packing Methods

`rowwise_matvec(...)` Computes the product of a plaintext matrix and an encrypted vector.

- **Parameters:** A `vector<vector<double>>` for the plaintext matrix, a `Ciphertext` for the encrypted vector, the matrix dimension, and SEAL context objects (`Evaluator`, `CKKSEncoder`, `GaloisKeys`, scale).
- **Returns:** A single `Ciphertext` containing the encrypted result vector.

`rowwise_encmat_encvec(...)` Computes the product of an encrypted matrix (where each row is a ciphertext) and an encrypted vector.

- **Parameters:** A `vector<Ciphertext>` for the encrypted matrix rows, a `Ciphertext` for the encrypted vector, dimension, and SEAL context objects (including `RelinKeys`).
- **Returns:** A single `Ciphertext` containing the encrypted result vector.

`rowwise_encmat_encmat(...)` Computes the product of two encrypted matrices, A and B. Matrix A is packed row-wise, and matrix B is packed column-wise.

- **Parameters:** Two `vector<Ciphertext>` objects for the matrices, dimension, and SEAL context objects.
- **Returns:** A `vector<Ciphertext>` where each ciphertext corresponds to a column of the resulting encrypted matrix.

B.2 Diagonal-wise (Halevi-Shoup) Packing Methods

`diagonal_matvec(...)` Computes the product of a plaintext matrix and an encrypted vector using the Halevi-Shoup diagonal packing method.

- **Parameters:** A `vector<vector<double>>` for the plaintext matrix, a `Ciphertext` for the encrypted vector, dimension, and SEAL context objects.

- **Returns:** A single `Ciphertext` containing the encrypted result vector.

`diagonal_encmat_encvec(...)` Computes the product of an encrypted matrix (packed by diagonals) and an encrypted vector.

- **Parameters:** A `vector<Ciphertext>` for the encrypted matrix diagonals, a `Ciphertext` for the encrypted vector, dimension, and SEAL context objects.
- **Returns:** A single `Ciphertext` containing the encrypted result vector.

`diagonal_encmat_encmat(...)` Computes the product of two encrypted matrices. Matrix A is packed by diagonals, and matrix B is packed by columns.

- **Parameters:** Two `vector<Ciphertext>` objects for the matrices, dimension, and SEAL context objects.
- **Returns:** A `vector<Ciphertext>` where each ciphertext corresponds to a column of the resulting encrypted matrix.