HEonGPU: a GPU-based Fully Homomorphic Encryption Library 1.0

Ali Şah Özcan*
alisah@sabanciuniv.edu

Erkay Savaş
erkays@sabanciuniv.edu

October 2024

Abstract

HEonGPU is a high-performance library designed to optimize Fully Homomorphic Encryption (FHE) operations on Graphics Processing Unit (GPU). By leveraging the parallel processing capacity of GPUs, HEonGPU significantly reduces the computational overhead typically associated with FHE by executing complex operation concurrently. This allows for faster execution of homomorphic computations on encrypted data, enabling real-time applications in privacy-preserving machine learning and secure data processing. A key advantage of HEonGPU lies in its multi-stream architecture, which not only allows parallel processing of tasks to improve throughput but also eliminates the overhead of data transfers between the host device (i.e., CPU) and GPU. By efficiently managing data within the GPU using multi-streams, HEonGPU minimizes the need for repeated memory transfers, further enhancing performance. HEonGPU's GPU-optimized design makes it ideal for large-scale encrypted computations, providing users with reduced latency and higher performance across various FHE schemes.

1 Introduction

In contrast to conventional symmetric and asymmetric encryption methods, fully homomorphic encryption (FHE) is a secure cryptographic method that enables meaningful arithmetic and logic calculations to be conducted directly on encrypted data without introducing any security vulnerabilities and without relying on the secret key. Since data remains encrypted and is not decrypted even during homomorphic operations, FHE can ensure the confidentiality of customer data in outsourced (cloud) computing scenarios where privacy preservation is necessary, when faced with both internal and external attacks. Several open-source software libraries implement FHE schemes such as Concrete [1], HEAAN [2], HElib [3], Lattigo [4], Microsoft SEAL [5], OpenFHE [6], PALISADE [7]. However, all of these libraries execute their operations on the Central Processing Unit (CPU), which allows only limited parallel execution capacity. Here, we introduce an open-source FHE library, HEonGPU, optimized for Graphics Processing Unit (GPU, Nvidia® GPUs) utilization, enabling all operations to be conducted on GPU and transferred back-and-forth between GPU and CPU only when strictly needed. This approach effectively eliminates the memory transfer overhead between the CPU and GPU. Furthermore, while HEonGPU is based on CUDA [8], users do not require any prior knowledge of GPU, even at a fundamental level. This is due to the fact all CUDA kernels are embedded within object-oriented written in C++. Additionally, the class interface is exceptionally straightforward, drawing inspiration from the user-friendly design of Microsoft SEAL [5]. Currently, the HEonGPU library exclusively supports Brakerski/Fan-Vercauteren (BFV) [9] and Cheon-Kim-Kim-Song (CKKS) [10] schemes. However, in the near future, HEonGPU will extend its support to include additional FHE schemes along with their new features:

- Bootstrapping support for Both CKKS [11–14] and BFV [15, 16] is expected to be available in the near future.
- It is planned to include other FHE schemes such as Brakerski-Gentry-Vaikuntantan (BGV) [17] and THFE [18], which are not currently included, in the future.

*Developer and corresponding author

- The support for a multi-GPU architecture is planned, along with the implementation of a multi-GPU memory pool structure to facilitate the execution of larger-scale applications.
- Future plans for the HEonGPU library include the addition of various features, such as the integration of Threshold Encryption [19,20] for Federated Learning [21,22], along with the provision of corresponding elementary examples.

This article explains the fundamental features of HEonGPU 1.0 and endeavors to offer a practical guide to GPU-based Fully Homomorphic Encryption (FHE), which provides more efficient implementation compared to other existing FHE libraries by harnessing the parallel execution capacity of GPU devices. HEonGPU is publicly accessible at the following address:

`https://github.com/Alisah-Ozcan/HEonGPU`

2 Roadmap

In Section 3, we briefly outline the notation used in this paper, including RNS, NTT, and FFT. Section 4 provides a comprehensive overview of HEonGPU, encompassing FHE scheme support, the Bootstrapping feature, and the External Product feature. Additionally, we delve into parameter selection considerations for optimizing both performance and security. In Section 5, we present timing results for HEonGPU operations, comparing them with those of other libraries. Furthermore, we contrast the results obtained from Multi-Thread CPU and Multi-Stream GPU implementations across various applications to facilitate a nuanced comparison between GPU and CPU performance. Finally, in Section 6, we discuss its future trajectory.

3 Preliminaries

This section presents the notation used throughout the article. Additionally, the Residue Number System (RNS), which facilitates efficient arithmetic for large moduli, and Number Theoretic Transform (NTT) and Fast Fourier Transform (FFT), which are used for fast polynomial multiplication, are introduced briefly.

3.1 Notation

Table 1 provides a list of notation used, along with their descriptions and corresponding names in HEonGPU 1.0. The notation not included in Table 1, but used throughout the article are as follows: The symbols $\lceil\cdot\rceil$, $\lfloor\cdot\rfloor$, and $\lceil\cdot\rfloor$ indicate rounding up, rounding down, and rounding to the nearest integer, respectively. The notation $[a]_b$ signifies that the integer a is reduced into the range $[-b/2, b/2]$, whereas $|a|_b$ reduces a to the interval $[0, b-1]$. The symbols $+$,$-$ and $\times$ (or just $\cdot$) denote addition, subtraction, and multiplication, respectively, in either $\mathbb{Z}_q$ or $\mathcal{R}_q$. The symbol $\odot$ represents modular element-wise multiplication for vector representations of the elements of $\mathcal{R}_q$ in the NTT or FFT domains; an operation sometimes referred as the Hadamard product. The notation for random sampling is as follows: $a \leftarrow \mathcal{R}_q$ indicates that the coefficients of a are sampled from $\mathbb{Z}_q$ uniformly randomly. On the other hand, $e \leftarrow \chi$ means that the coefficients of e are sampled from the distribution χ.

3.2 Residue Number System (RNS)

We start with a brief overview of the RNS system:

- Q is an integer that can be written as $Q = \prod_{i=0}^{r-1} q_i$, where the base vector $\mathfrak{B}_Q^r = [q_0, q_1, \ldots, q_{r-1}]$ consists of pairwise co-prime integers.
- Any positive integer $X < Q$ can be represented with the residues $[x_0, x_1, \ldots, x_{r-1}]$, where $x_i = X \bmod q_i$. Computing residues of $X < Q$ is sometimes referred as RNS *decomposition.* This is known as RNS representation and by Chinese Remainder Theorem (CRT), any number less than Q has a unique RNS representation.
- Whenever needed, numbers in RNS bases can be converted to actual numbers. This operation is sometimes called *composition.* The algorithm to perform this conversion is defined in Equation 1.

Parameter	Description	Name in HEonGPU
$\tilde{Q} = \prod_{i=0}^{\tilde{\ell}-1} q_i$	Key modulus, ($\tilde{Q} = Q \cdot P$)	`coeff_modulus`
$Q = \prod_{i=0}^{\ell-1} q_i$	Ciphertext modulus, `coeff_modulus`	
$P = \prod_{i=\ell}^{\tilde{\ell}-1} q_i$	Additional primes for base extension	`coeff_modulus`
t	Plaintext modulus	`plain_modulus`
N	Number that is power of 2	n
$x^N + 1$	Reduction polynomial which specifies the ring R	
$\mathcal{R}$	The ring $\mathbb{Z}[x]/(x^N + 1)$	
$\mathcal{R}_Q$	The ring $\mathbb{Z}_Q[x]/(x^N+1)$, i.e., same as the ring R but with coefficients reduced modulo q	
$\tilde{\ell}$	RNS base dimension of $\tilde{Q}$	Q_prime_size
ℓ	RNS base dimension of Q	Q_size
Δ	Quotient on division of Q by t, i.e., $\lfloor Q/t \rfloor$	
χ	Error distribution (a truncated discrete Gaussian distribution)	
σ	Standard deviation of χ	`noise_standard_deviation`

Table 1: Symbols and terminology used in this document.

$$|X|_Q = \Big| \sum_{i=0}^{r-1} \Big| x_i \cdot Q_i^{-1} \Big|_{q_i} \cdot Q_i \Big|_Q, \text{ where } Q_i = \frac{Q}{q_i} \tag{1}$$

As there is a unique RNS representation, addition, subtraction, multiplication, and division modulo Q can be executed in parallel using single precision base elements (q_i) without resorting to multi-precision arithmetic (when Q exceeds the word size). These operations can be performed as

$$X \mp_{RNS} Y \sim \Big| x_i \mp y_i, \Big|_{q_i}, \tag{2}$$

$$X \times_{RNS} Y \sim \Big| x_i \times y_i, \Big|_{q_i}, \tag{3}$$

$$\text{Given } \gcd(Y, Q) \equiv 1 \Rightarrow X \div_{RNS} Y \sim \Big| x_i \times (Y)_{q_i}^{-1}, \Big|_{q_i}, \tag{4}$$

for $i \in \{0, 1, \dots, r-1\}$. RNS representation also supports base extension and base conversion capabilities. In base extension, an integer $X < Q$ in $\mathfrak{B}_Q^{\ell}$ can be represented in a larger base $\mathfrak{B}_{\tilde{Q}}^{\tilde{\ell}} = \mathfrak{B}_Q^{\ell} \cup \{q_\ell, q_{\ell+1}, \dots, q_{\tilde{\ell}-1}\}$, where $\tilde{Q} = Q \times \prod_{i=\ell}^{\tilde{\ell}-1} q_i$. To this end, the residues of X with respect to each new modulus are calculated $x_i = X \bmod q_i$ for $i = \ell, \dots, \tilde{\ell} - 1$.

Base conversion, on the other hand, transforms an RNS representation of an integer into another, by changing the base vector, namely $\mathfrak{B}_Q^{\ell} \to \mathfrak{B}_{P'}^{r'}$, where $\mathfrak{B}_{P'}^{r'} = \{p'_0, \dots, p'_{r'-1}\}$ and $P' = \prod_{i=0}^{r'-1} p'_i$. For accuracy of all integers in $\mathbb{Z}_Q$, we should have $Q \le P'$. However, depending on the range of integers represented one can also consider conversion to a smaller base [23, 24]..

3.3 Number Theoretic Transform (NTT)

Let $n, q \in \mathbb{Z}^+$ where N is a power of 2. The ring, $\mathcal{R}_q \in \mathbb{Z}_q/\langle x^N + 1 \rangle$, consists of polynomials with integer coefficients of degree less than N, where the coefficients are in modulo q. Given $a(x), b(x) \in \mathcal{R}_q$, the multiplication operation $c(x) = a(x) \times b(x) \in \mathcal{R}_q$ is one of the most time consuming operations in RLWE based homomorphic encryption schemes. To compute $c(x)$, traditional techniques, such as the schoolbook multiplication given in Equation 5, necessitate $\mathcal{O}(N^2)$ multiplication operations and the subsequent polynomial reduction operation given in Equation 6.

$$\tilde{c}(x) = a(x) \times b(x) = \sum_{i=0}^{N-1} \sum_{j=0}^{N-1} a_i b_j x^{i+j} \mod q \tag{5}$$

$$c_i = \tilde{c}_i - \tilde{c}_{N+i} \mod q, \; i \in \{0, 1, \dots, N-1\} \tag{6}$$

On the other hand, with Number Theoretic Transform (NTT) which is a form of Discrete Fourier Transform (DFT), the multiplication in $\mathcal{R}_q$ can be performed in $\mathcal{O}(N \log(N))$. The coefficients of a

$a(x) \in \mathcal{R}_q$ can be considered as a vector of integers, $\boldsymbol{a} = [a_0, a_1, \ldots, a_{N-1}]$, which can be converted to another vector $\bar{\boldsymbol{a}} = [\bar{a}_0, \bar{a}_1, \ldots, \bar{a}_{N-1}]$ of the same dimension using NTT, where $\bar{\boldsymbol{a}} = NTT(\boldsymbol{a})$. The inverse of the NTT operation, denoted as $iNTT$ converts a vector $\bar{a}$ in the NTT domain back to coefficient domain with $a = iNTT(\bar{a})$. Then, the NTT-based multiplication can be defined as in Equation 7.

$$c(x) = INTT(NTT(a(x) \odot NTT(b(x)) \tag{7}$$

where the symbol $\odot$ stands for the element-wise multiplication in $\mathbb{Z}_q$; i.e., $\bar{c}_i = \bar{a}_i \cdot \bar{b}_i \bmod q$ for $i = 0, 1, \ldots, N-1$.

The N-point negacyclic NTT and iNTT operations can be defined as, respectively,

$$\bar{a}_i = \sum_{j=0}^{N-1} a_j \psi^{i \times j} \mod q, \;\; i \in \{0, 1, 2, \ldots, N-1\} \tag{8}$$

$$a_i = \frac{1}{N} \sum_{j=0}^{N-1} \bar{a}_j \psi^{-i \times j} \mod q, \tag{9}$$

for $i \in \{0, 1, \ldots, N-1\}$. The definition of negacyclic NTT requires the existence of a NTT friendly prime $q \in \mathbb{Z}^+$, where $q \equiv 1 \mod 2N$ and an integer value ($2N$-th root of unity) $\psi \in \mathbb{Z}_q$. ψ has to satisfy both conditions $\psi^{2N} \equiv 1 \mod q$ and $\psi^i \neq 1 \mod q, \;\; \forall i < 2N$. The HEonGPU library uses an iterative algorithm to compute NTT, which represents the state-of-the-art in the literature to perform NTT-iNTT operations on GPU [25][1].

3.4 Fast Fourier Transform (FFT)

Fast Fourier Transform (FFT) algorithm computes Discrete Fourier Transform (DFT), in a similar fashion to the Number Theoretic Transform (NTT). A fundamental difference with NTT, FFT works with complex numbers. Similar to the NTT, the FFT can be used for polynomial multiplication in $\mathcal{R}_q$. As the complex numbers are represented as floating point numbers in computers, the precision of the mantissa of the floating point number determine the largest value of the modulus q. In the IEEE 754 standard, single and double precision floating point numbers have 23- and 52-bit mantissa size[2], respectively. Thus, the coefficients of polynomials must not exceed $\sqrt{\frac{2^{23}}{\log N}}$ any $\sqrt{\frac{2^{52}}{\log N}}$ for single and double precision, respectively. The N-point negacyclic versions of FFT and inverse FFT operations can be defined as follows:

$$\bar{a}_k = \sum_{j=0}^{2N-1} a_j \exp\left(\frac{-2\pi ijk}{2N}\right), \tag{10}$$

$$a_k = \frac{1}{2n} \sum_{j=0}^{2N-1} \bar{a}_j \exp\left(\frac{2\pi ijk}{2N}\right), \tag{11}$$

for $k = 0, 1, \ldots, N-1$. Then, we have $\bar{\boldsymbol{a}} = FFT(\boldsymbol{a})$ and $\boldsymbol{a} = iFFT(\bar{\boldsymbol{a}})$.

In HEonGPU, the NTT is preferred over the FFT for polynomial multiplications due to its higher efficiency. Nonetheless, the FFT remains in use for CKKS encoding. The HEonGPU Library leverages the Merge FFT function[3] from the GPU-FFT library, which is considered the state-of-the-art in the literature, to execute FFT and iFFT operations on GPU [25].

4 Overview of HEonGPU

HEonGPU is a high-performance, GPU-based Fully Homomorphic Encryption library written in CUDA. Despite its dependency on CUDA, it is an object-oriented library that can be fully utilized with C++, similar to HEAAN [2], HElib [3], Microsoft SEAL [5], OpenFHE [6] and PALISADE [7] libraries. Moreover, its interface is designed to resemble Microsoft SEAL [5] for user-friendliness, as visualized in Listings 1 and 2. The library aims to incorporate all HE schemes in terms of cryptographic capability. Currently, HEonGPU supports only BFV and CKKS, and its comparison with other libraries in terms of cryptographic capability is presented in Table 2.

[1]See `https://github.com/Alisah-Ozcan/GPU-NTT` for the source code.

[2]`https://ieeexplore.ieee.org/document/8766229`

[3]`https://github.com/Alisah-Ozcan/GPU-FFT`

```
#include "seal.h"
int main() {
    seal::EncryptionParameters parms(seal::
        scheme_type::ckks);
    size_t poly_modulus_degree = 8192;
    parms.set_poly_modulus_degree(
        poly_modulus_degree);
    parms.set_coeff_modulus(seal::
        CoeffModulus::Create(
    poly_modulus_degree, { 60, 40, 40, 60 }))
        ;
    double scale = pow(2.0, 40);

    seal::SEALContext context(parms);

    seal::KeyGenerator keygen(context);
    seal::SecretKey sk = keygen.secret_key();
    seal::PublicKey pk;
    keygen.create_public_key(pk);
    seal::RelinKeys rlk;
    keygen.create_relin_keys(rlk);

    seal::Encryptor encryptor(context, pk);
    seal::Evaluator evaluator(context);
    seal::Decryptor decryptor(context, sk);
    seal::CKKSEncoder encoder(context);

    size_t slot_count = encoder.slot_count();
    size_t row_size = slot_count / 2;

    std::vector<double> M1(slot_count, 0.5);
    seal::Plaintext P1;
    encoder.encode(M1, scale, P1);

    seal::Ciphertext C1;
    encryptor.encrypt(P1, C1);

    evaluator.multiply_inplace(C1, C1);
    evaluator.relinearize_inplace(C1, rlk);
    evaluator.rescale_to_next_inplace(C1);

    seal::Plaintext P2;
    decryptor.decrypt(C1, P2);

    std::vector<double> M2;
    encoder.decode(P2, M2);

    return EXIT_SUCCESS;
}
```

Listing 1: Code Example of SEAL

```
#include "heongpu.cuh"
int main() {
    size_t poly_modulus_degree = 8192;
    heongpu::Parameters context(
        heongpu::scheme_type::ckks,
        heongpu::keyswitching_type::
            KEYSWITHING_METHOD_I);
    context.set_poly_modulus_degree(
        poly_modulus_degree);
    context.set_coeff_modulus({60, 40, 40},
        {60});
    context.generate();
    double scale = pow(2.0, 40);

    heongpu::HEKeyGenerator keygen(context);
    heongpu::Secretkey sk(context);
    keygen.generate_secret_key(sk);
    heongpu::Publickey pk(context);
    keygen.generate_public_key(pk, sk);
    heongpu::Relinkey rlk(context);
    keygen.generate_relin_key(rlk, sk);

    heongpu::HEEncryptor encryptor(context, pk)
        ;
    heongpu::HEOperator operators(context);
    heongpu::HEDecryptor decryptor(context, sk)
        ;
    heongpu::HEEncoder encoder(context);

    size_t row_size = poly_modulus_degree / 2;
    std::vector<double> message(row_size, 0.5);

    heongpu::Message M1(message, context);
    heongpu::Plaintext P1(context);
    encoder.encode(P1, M1, scale);

    heongpu::Ciphertext C1(context);
    encryptor.encrypt(C1, P1);

    operators.multiply_inplace(C1,C1);
    operators.relinearize_inplace(C1,rlk);
    operators.rescale_inplace(C1);

    heongpu::Plaintext P2(context);
    decryptor.decrypt(P2, C1);

    heongpu::Message M2(context);
    encoder.decode(M2, P2);

    return EXIT_SUCCESS;
}
```

Listing 2: Code Example of HEonGPU

In this section, we explore the supported FHE schemes of the HEonGPU library, its forthcoming feature set, and its structural attributes.

Library/ Scheme	BGV	BFV	CKKS	CKKS Bootstrapping	TFHE
HEAAN [2]			✓	✓	
HELib [3]	✓	✓	✓		
HEonGPU	**Very Soon**	✓	✓	**Soon**	*
OpenFHE [6]	✓	✓	✓	✓	✓
PALISADE [7]	✓	✓	✓	✓	✓
SEAL [5]	✓	✓	✓		

* Plan with Collaboration

Table 2: Current Capabilities and Schemes of Existing Fully Homomorphic Encryption (FHE) Libraries

4.1 FHE Schemes

Practical lattice-based fully homomorphic encryption (FHE) schemes rely on the hardness of the Ring Learning with Errors (RLWE) problem to ensure their security. Thus, ciphertext contains certain level of noise, which increases as operations are performed homomorphically over ciphertext, eventually potentially making it undecipherable after a certain number of homomorphic operations. The ciphertext modulus Q is the main factor that determines the noise budget which is a measure of the number of homomorphic operations that are allowed. While a larger ciphertext modulus extends the noise budget, it reduces the security level of the ciphertext when the ring dimension N is fixed. For this, larger ring dimensions are utilized to maintain security, which further raises the cost of homomorphic operations. An alternative approach involves using a smaller ciphertext modulus and ring dimension, and the noise budget is replenished periodically by applying the so-called *bootstrapping* operation to the ciphertext, which is essentially a homomorphic operation that reduces the noise level in the ciphertext, which allows further homomorphic operations. Although using large ring dimensions increases the homomorphic operation cost, BFV, BGV, and CKKS schemes support batching whereby a ciphertext encrypts many plaintext in what is called as its slots. This way, a single homomorphic operation can be applied on plaintext in the slots, which is commonly known as single instruction multiple data (SIMD) paradigm. In other words, it can perform homomorphic operations over vectors of integers or real numbers in a SIMD manner. The number of plaintext slots is related to the ring dimension, N; in BGV and BFV the number of slots is N, while it is $N/2$ in CKKS.

HEonGPU currently implements only the Residue Number System (RNS) variants of the BFV and CKKS schemes for efficiency reason. As elaborated in Section 3.2, the RNS enables the representation of large integers by smaller ones that fit in computer word size (e.g., 64 bit), facilitating fast and concurrent arithmetic operations of large integers, typically used in HE operations. Due to inherent parallelism in RNS arithmetic, the approach enables the efficient utilization of the GPU's multi-core architecture. Although Nvidia GPUs are equipped with 32-bit Arithmetic Logic Units (ALUs), the HEonGPU library performs 64-bit arithmetic operations as well. While using 32-bit RNS bases may seem to enhance parallelism, our experience has shown that, in terms of computational overhead, 64-bit operations are the optimal choice when compared to 32-bit or 128-bit alternatives.

Although HEonGPU presently supports only the BEHZ [26] variant, future updates will include the HPS [27] variant and its optimized forms [28], along with the addition of leveled implementation for each variant. At present, users must manually perform the relinearization operation following homomorphic multiplication. Further homomorphic multiplication is not permitted without applying relinearization after multiplication and the elimination of the nonlinear term, $ct[2]$; thus, the formation of an additional term $ct[3]$ is not allowed. However, operations such as homomorphic addition and homomorphic subtraction can still be performed while the nonlinear term is present.

The RNS variant of the CKKS scheme employs a uniform scaling factor across all levels. Given that the scaling factor is a power of 2 (2^p), the RNS primes corresponding to the multiplication levels must be selected very close to 2^p $\;(2^p \approx q_i)$; otherwise, the error amount will increase significantly [29]. As with the BFV implementation, the relinearization operation after homomorphic multiplication in CKKS must be performed manually. Additionally, CKKS requires a modulus division, or rescaling operation, to reduce error and this must also be done also manually. If not performed, HEonGPU will not permit further multiplication operations. All these manual operations will be automated and integrated into the library in the near future. A brief overview of the basic operations in both BFV and CKKS schemes are provided below.

Let $Q = \prod_{i=0}^{\ell-1} q_i$, $P = \prod_{i=\ell}^{\tilde{\ell}-1} q_i$ and $\tilde{Q} = Q \cdot P$, where $\tilde{\ell} = \ell + \pi$ for some integer π. Note also that $q_i \leq P$ for $i = 0, \ldots, \ell - 1$.

- **SecretKeyGeneration(λ):** $sk = s$, where $s \leftarrow \mathcal{R}_2$.
- **PublicKeyGeneration(sk):** $\boldsymbol{pk} = (p_0, p_1) = (|-(a \cdot s + e)|_{\tilde{Q}}, a)$, where $a \leftarrow \mathcal{R}_{\tilde{Q}}$, $e \leftarrow \chi$.
- **EvaluationKeyGeneration(sk):** $\boldsymbol{evk_i} = \left(\left| -a' \cdot s + s' \cdot P \cdot \frac{Q}{q_i} \cdot \left| (\frac{Q}{q_i})^{-1} \right|_{q_i} + e' \right|_{\tilde{Q}}, a' \right)$

  for $0 \leq i < \ell - 1$, where $s = \boldsymbol{sk}$, $a' \leftarrow \mathcal{R}_{\tilde{Q}}$, $e' \leftarrow \chi$. Here, $\boldsymbol{evk}$ can be relinearization, key switching or rotation keys. If it is a relinearization key, then $s' = s^2$ or a higher power of the secret key, depending on the ciphertext size. For key switching, it is the new key while for rotation it is the rotation key. Also, note also that each part of the evaluation key is written in RNS representation with $\mathfrak{B}_{\tilde{Q}}^{\tilde{\ell}}$. Finally, the term $\frac{Q}{q_i} \cdot \left| (\frac{Q}{q_i})^{-1} \right|_{q_i}$ is the RNS gadget for $\mathcal{R}_Q$ and included for RNS composition with respect to Q, as will be observed in the subsequent section devoted to relinearization. Therefore, the evaluation keys can be rewritten as $rlk_i = \left(\left| -a' \cdot s + s^2 \cdot P \cdot g_i + e' \right|_{\tilde{Q}}, a' \right)$ for $i = 0, \ldots \ell - 1$.
- **Encryption($\boldsymbol{pk}, m$):**

◇ **BFV:**

$$\boldsymbol{ct} = \left(\left| \Delta \cdot m + \frac{|p_0 \cdot u + e_1|_{\tilde{Q}}}{P} \right|_Q, \left| \frac{|p_1 \cdot u + e_2|_{\tilde{Q}}}{P} \right|_Q \right), \tag{11}$$

where $m \in \mathcal{R}_t$, $u \leftarrow \mathcal{R}_2$ and $e_1, e_2 \leftarrow \chi$.

◇ **CKKS:**

$$\mathbf{ct} = \left(\left| \acute{m} + \frac{|p_0 \cdot u + e_1|_{\tilde{Q}}}{P} \right|_Q, \left| \frac{|p_1 \cdot u + e_2|_{\tilde{Q}}}{P} \right|_Q \right), \tag{11}$$

where $m \in \mathbb{R}$, $\acute{m} = \text{encode}(m)$, $u \leftarrow \mathbf{R}_2$ and $e_1, e_2 \leftarrow \chi$.

Note that we only show the message encoding of the CKKS scheme, which is slightly different than that of BFV as it encodes messages that are real numbers.

- **Decryption($sk, \boldsymbol{ct}$):**

◇ **BFV:**

$$\left|\left\lceil \frac{t}{Q} \cdot |ct[0] + ct[1] \cdot s|_Q \right\rfloor\right|_t \tag{11}$$

◇ **CKKS:**

$$|ct[0] + ct[1] \cdot s|_Q \tag{11}$$

Note that we do not show the message decoding operations, which should be normally applied after the decryption operations.

- **Addition($\boldsymbol{ct}_0, \boldsymbol{ct}_1$):** $(ct_0[0] + ct_1[0], ct_0[1] + ct_1[1])$.
- **Subtraction($\boldsymbol{ct}_0, \boldsymbol{ct}_1$):** $(ct_0[0] - ct_1[0], ct_0[1] - ct_1[1])$.
- **Multiplication($\boldsymbol{ct}_0, \boldsymbol{ct}_1$):** $\boldsymbol{ct}_2 = \boldsymbol{ct}_0 \times \boldsymbol{ct}_1$

◇ **BFV:**

$$ct_2[0] = \left|\left\lceil \frac{t}{Q} \cdot ct_0[0] \cdot ct_1[0] \right\rfloor\right|_Q$$

$$ct_2[1] = \left|\left\lceil \frac{t}{Q} \cdot (ct_0[0] \cdot ct_1[1] + ct_0[1] \cdot ct_1[0]) \right\rfloor\right|_Q$$

$$ct_2[2] = \left|\left\lceil \frac{t}{Q} \cdot ct_0[1] \cdot ct_1[1] \right\rfloor\right|_Q \tag{9}$$

◇ **CKKS:**

$$\begin{aligned} ct_2[0] &= |ct_0[0] \cdot ct_1[0]|_Q \\ ct_2[1] &= |ct_0[0] \cdot ct_1[1] + ct_0[1] \cdot ct_1[0]|_Q \\ ct_2[2] &= |ct_0[1] \cdot ct_1[1]|_Q \end{aligned} \tag{7}$$

- **Relinearization($\boldsymbol{ct}, \boldsymbol{rlk}$)** :

$$\tilde{ct}[0] = \left| ct[0] + \left\lceil \frac{ct[2] \cdot \boldsymbol{rlk}[0]}{P} \right\rfloor \right|_Q,$$

$$\tilde{ct}[1] = \left| ct[1] + \left\lceil \frac{ct[2] \cdot \boldsymbol{rlk}[1]}{P} \right\rfloor \right|_Q,$$

4.2 Key Switching Operations

Despite their fundamentally different encryption, and decryption methods, both cryptosystems (BGV can also be included) require a 'key switching' operation for rotation and relinearization, the latter of which is necessary after homomorphic multiplication operations. In summary, key switching is a core operation in HE. HEonGPU support three different variants of key switching for both BFV and CKKS schemes, where we refer them as Method I [9], Method II [30, 31], and Method III [32], respectively. While Method I and Method II are essentially based on the same algorithm, they are implemented as separate methods in the library to provide users with ease of deployment depending on the application requirements. The primary distinction of these three methods are that they require different numbers of NTT, iNTT, Hadamard product, and base conversion operations, which are listed in Table 3.

Method[1]	NTT	INTT	H.P.	Base Conv
I	$\ell \times \tilde{\ell}$	$2 \times \tilde{\ell}$	$2 \times \ell \times \tilde{\ell}$	-
II	$d \times \tilde{\ell}$	$2 \times \tilde{\ell}$	$2 \times d \times \tilde{\ell}$	d
III	$d \times r'$	$2 \times \tilde{d} \times r'$	$2 \times d \times \tilde{d} \times r'$	$d + 2\tilde{d}$

[1]Definitions of d, $\tilde{d}$, and r' are given in [32]

Table 3: The number of NTT, iNTT, hadamard product, and base conversion Operations

The numbers of both NTT operations and Hadamard products decrease in Method II with respect to Method I while it incurs extra base conversion operations. As the based conversion is not necessarily an expensive operation, Method II is expected to accelerate the key switching significantly for large values of N and Q. On the other hand, the overhead of Method III is more substantial. While it further decreases the number of NTT operations, it may significantly increase the numbers of Hadamard products, iNTT and base conversion operations depending on the values of $\tilde{\ell}$, $\tilde{d}$, and r'. In a CPU implementation of Method II presented in [32] for large values of N and Q, the authors report that Method III leads to significant speedup values over Method II. The advantage of Method III heavily depends on the extent the forward NTT operations dominates the execution time of the external product. However, our observations indicate that the advantage of Method III diminishes when NTT operations are accelerated using the parallel architecture of GPU devices, particularly in comparison to Method II. Additionally, usage of Method III is not recommended for use in HEonGPU due to its significantly larger key size compared to the other methods, and the requirement for separate keys at each level in CKKS. The primary reason for this is that all keys are stored in GPU memory, which can quickly filled GPU memory at high ring dimensions and large RNS bases. Lastly, while Method I and Method II involve key switching, rotation, and relinearization operations, Method III is limited to relinearization only, due to the aforementioned constraints.

4.3 Bootstrapping

Bootstrapping in homomorphic encryption is a technique used to reduce the noise that accumulates during computations on encrypted data. As operations are performed, the noise increases and can eventually corrupt the plaintext data, limiting the number of permissible homomorphic operations. Bootstrapping refreshes the ciphertext by decrypting it within the encrypted domain, effectively resetting the noise level. This theoretically enables an unlimited number of homomorphic operations, rendering the encryption scheme 'fully' homomorphic [33]. Additionally, it is useful in scenarios where switching between different FHE schemes is advantageous.

HEonGPU currently does not have a bootstrapping implementation. However, in the short term, approximate bootstrapping will be implemented for the CKKS scheme (with RNS variant) described in word [12–14]. Additionally, when the TFHE scheme is added to the library in the short term, it will be integrated together with bootstrapping [18]. In the long term, bootstrapping operations for the BFV and BGV schemes may be added to the HEonGPU library, though this is not currently on our immediate agenda. We anticipate that, as the library evolves, bootstrapping operations for all schemes and various optimizations will be incorporated over time.

4.4 Encoding

A crucial aspect of making homomorphic encryption both practical and efficient is the selection of an appropriate encoder for the specific task. Since messages are situated in cyclotomic ring; R_t for BFV and

R for CKKS where they can be encoded and converted into vectors, enabling the use of SIMD structures. In its current state, HEonGPU implements only 'batch encoding', but 'integer' and 'fractional encoders' will be added shortly. This limitation does not restrict users from performing integer or fractional encoding, as these methods utilize the batch encoding structure; users simply need to apply a preprocessing step beforehand.

4.5 Noise Estimation

HEonGPU includes invariant noise estimation exclusively for the BFV scheme; it does not provide noise estimation for the CKKS scheme. Consequently, when utilizing the CKKS scheme, users must monitor the ciphertext depth during operations. Specifically for CKKS, users should perform a rescale operation following each homomorphic multiplication until the available RNS bases are exhausted.

In the BFV scheme, HEonGPU employs the error calculation method used by the SEAL library [5]. This method computes the invariant noise budget of a ciphertext, expressed in bits, which quantifies the remaining capacity for noise growth while still ensuring correct decryption. As shown in Equation 8 used for decryption, the invariant noise polynomial, v, associated with a ciphertext, is a rational coefficient polynomial, where correct decryption is guaranteed as long as the absolute value of the polynomial's coefficients remains less than 1/2.

$$\frac{t}{Q}(\boldsymbol{ct} \cdot sk) = \frac{t}{Q}\left(ct[0] + ct[1]s\right) = m + v + at \tag{8}$$

Initially, the noise bugdet is given by Q/t, where Q and t are the modulus of the ciphertext and the modulus of the plaintext, respectively. As homomorphic operations are performed, the noise budget decreases, and once it reaches 0, the ciphertext becomes too noisy to be decrypted correctly [5]. During the processes, the instantaneous value of the noise budget in terms of the number of bits can be estimated as follows:

$$\textbf{Remaining noise budget} = \log Q - \log\left(t \cdot (\boldsymbol{ct} \cdot sk)\right) \tag{9}$$

If the **Remaining noise budget** in Equation 9 is 0 or less, the ciphertext becomes too noisy to be decrypted accurately.

4.6 Random Number Generation

In this work, we employed the CURAND library [34] for random number generation on NVIDIA GPUs to enhance computational efficiency. CURAND offers various random number generators (RNGs), such as XORWOW, MRG32k3a, MTGP32, and Sobol, each optimized for different performance characteristics [35–37]. Our implementation utilizes the `curandState_t` structure, which employs the XORWOW RNG, known for its speed and low memory footprint [34]. However, it is important to note that XORWOW, like the other RNGs provided by CURAND, is not cryptographically secure [38]. While these RNGs are well-suited for high-performance computing tasks, they lack the robustness required for cryptographic applications. Specifically, our use case involved initializing the RNG state with `curand_init` and generating random values using CURAND.

Given these limitations, future versions of the library will explore the integration of a cryptographically secure random number generator (CSPRNG). We plan to implement an AES-based CSPRNG, which offers the requisite security properties for generating unpredictable and tamper-resistant random numbers [38]. This enhancement will allow us to extend the applicability of our methods to security-critical domains, such as cryptographic key generation and secure data processing, where the integrity and confidentiality of random number generation are paramount.

4.7 Memory Management

In HEonGPU, we utilized the RAPIDS Memory Manager (RMM) to implement an efficient memory pool, optimizing the allocation, deallocation, and management of GPU memory [39]. RMM, developed by the RAPIDS AI team at NVIDIA, is an essential component within the RAPIDS ecosystem, specifically designed to handle memory management in GPU-accelerated data science and machine learning workflows. By offering a more sophisticated approach to memory management compared to traditional CUDA APIs, RMM significantly enhances performance, particularly in scenarios involving frequent and dynamic memory allocations.

Since HEonGPU is a GPU-based HE library and store data on GPU, the design and implementation of a dedicated memory pool for both on CPU and GPU are as important as fast and efficient kernel implementations. Utilizing RMM, we developed a memory pool leveraging the `rmm::pool_memory_resource` class, which allows for efficient and dynamic memory allocation. The design is carefully crafted to provide flexibility by enabling the memory pool to operate with either a `rmm::pool_memory_resource` or a `rmm::cuda_memory_resource`, depending on the specific needs of the application for GPU memory pool. This adaptability ensures that our memory management approach can be tailored to different scenarios, maximizing performance across a variety of workloads. On the CPU memory pool side, a dedicated physical partition is established in the CPU RAM via `rmm::pinned_memory_resource`. This approach significantly accelerates memory transfers between the GPU and CPU compared to the use of pageable memory.

The `rmm::pool_memory_resource` is a sophisticated memory resource class within the RMM library designed to optimize memory allocation by pooling CPU and GPU memory resources. It operates by pre-allocating large blocks of memory and managing these blocks internally, minimizing the need for frequent and expensive calls to `cudaMalloc`, `cudaFree`, `cudaHostAlloc` and `cudaFreeHost` which are standard CUDA functions for memory allocation and deallocation. When a memory request is made, `rmm::pool_memory_resource` checks its internal pool of pre-allocated memory. If a suitable block is available, it is allocated to the request without needing to interact with the CUDA API, resulting in significantly lower latency. If no suitable block is available, `rmm::pool_memory_resource` allocates additional memory blocks as needed, either from the remaining GPU memory or by expanding the pool size, depending on the configuration. This pooling mechanism not only reduces the overhead associated with dynamic memory allocations but also helps to mitigate fragmentation issues that can arise when memory is frequently allocated and deallocated. By reusing memory blocks from the pool, `rmm::pool_memory_resource` ensures that memory is utilized more efficiently, which is particularly important in GPU-accelerated applications where memory is a critical resource.

To further enhance the control and flexibility of the memory pool, we defined the initial and maximum pool sizes in a configuration header file (`define.h`). By default, the GPU memory pool is configured with an initial pool size set to 50% of the available GPU memory, while the maximum pool size is set to 80%. For the CPU memory pool, these parameters are set to 10% of the available system memory for the initial pool size and 20% for the maximum pool size. It should be noted that, since the CPU memory pool allocates pinned memory, creating and destroying large CPU memory pools can be time-consuming. This approach allows us to preconfigure the memory pool's capacity based on the anticipated workload, ensuring that memory is utilized efficiently without unnecessary overhead. By setting these parameters in `define.h`, we can easily adjust the memory pool's behavior across different runs or projects, providing an additional layer of optimization and customization to our computational framework. Currently, our memory pool is designed for single-GPU environments, focusing on optimizing memory usage. However, recognizing the growing importance of multi-GPU setups in high-performance HE computing, we plan to extend this memory pool design to support multi-GPU configurations in the future. This extension will involve ensuring efficient memory allocation, synchronization, and data transfer across multiple GPUs, leveraging the advanced features of RMM to maintain performance scalability.

Additionally, considering the relatively smaller size of GPU memory compared to CPU RAM, managing data overflow effectively is crucial. As the dataset sizes continue to grow, there may be instances where the GPU memory becomes saturated. To address this, we are also implemented a specialized structure that allows for seamless data migration from GPU memory to CPU memory when necessary. This mechanism will ensure that our applications can handle large datasets without running into memory limitations, by temporarily offloading excess data to CPU RAM and retrieving it as needed.

The custom memory pool we developed plays a crucial role in our workflow, particularly in managing GPU memory for data structures such as vectors. To this end, we designed a specialized `heongpu::DeviceVector` class that integrates seamlessly with our memory pool. The `heongpu::DeviceVector` not only benefits from the efficient memory management provided by the memory pool but also interfaces with RMM's `rmm::device_uvector`, which is an advanced data structure within the RAPIDS ecosystem that enables efficient and flexible GPU memory handling, ensuring that operations such as memory allocation, resizing, and deallocation are optimized for high-performance computing tasks. The `heongpu::HostVector` is a specialized adaptation of `std::vector`, targeting with our custom pinned memory allocator. This design enables efficient utilization of the CPU memory pool allocated as pinned memory, while also leveraging the extensive features provided by `std::vector`.

4.8 Multi-Stream

In the realm of GPU computing, CUDA streams are integral to the efficient execution of parallel tasks. A CUDA stream represents a sequence of operations that the GPU executes in the order they are issued. Each stream operates independently, which allows for multiple streams to execute concurrently, thereby harnessing the full potential of the GPU's parallel processing capabilities. One of the most significant benefits of utilizing CUDA streams is the ability to overlap different operations, such as computation and data transfer. For example, while one stream is engaged in transferring data between the host and the GPU, another stream can simultaneously execute a kernel on the GPU. This concurrent execution ensures better resource utilization and minimizes idle times, leading to enhanced overall performance [40].

CUDA streams also provide developers with fine-grained control over the execution order of operations. This capability is particularly valuable when orchestrating complex workflows, where managing data dependencies and reducing unnecessary synchronization can lead to substantial performance gains. By strategically assigning tasks to different streams, it is possible to optimize the execution flow and maximize throughput in GPU-accelerated applications. While streams operate independently, CUDA allows for synchronization across streams when required, achieved through the use of events, which serve as synchronization points. Events ensure that operations in different streams only proceed after specific conditions are met, providing a balance between parallelism and the need for accurate, deterministic results.

Moreover, CUDA streams exhibit even greater performance when they are managed by different CPU threads, with each thread controlling a distinct stream. This approach takes advantage of the CPU's ability to handle multiple threads concurrently, allowing each thread to independently issue commands to its associated stream on the GPU. By aligning CPU threads with CUDA streams in this manner, the workload can be distributed more efficiently, reducing contention and ensuring that the GPU is kept consistently busy. Thanks to this configuration, the HEonGPU library is particularly effective in scenarios involving complex multitasking or real-time processing, where minimizing latency and maximizing parallelism are critical.

4.9 Interface and Parameter Selection

The HEonGPU library is a GPU-based library built using C++ classes. The initial class that needs to be instantiated is `heongpu::Parameters`. This class is responsible for creating the encryption parameters and transferring all necessary data (e.g., the primitive root of unity tables for NTT) to the GPU memory. The inputs required for `heongpu::Parameters` are as follows;

- **scheme**: Either BFV or CKKS for the time being
- **poly_modulus_degree**: A power of 2 in the range of [2^{12}, 2^{16}]
- **log_Q_bases_bit_sizes**: An integer array that stores the bit sizes of the RNS bases that constitutes the large modulus Q, which is the ciphertext modulus.
- **log_P_bases_bit_sizes**: An integer array that stores the bit sizes of the RNS bases that constitute the modulus P, which is used in the key generation process.(Remark: $\tilde{Q} = QP$ is the modulus for keys.)
- **plain_modulus**: It is used for the BFV plaintext modulus (CKKS do not use plaintext modulus).

HEonGPU, similar to SEAL and OpenFHE, cen be used with default parameters as well as custom parameters, based on the security level and **poly_modulus_degree**. The modulus sizes for the generated default parameters corresponding to 128, 192, and 256-bit security levels are listed in Table 4. All numbers in this table are approximate and have been derived from lattice-estimator [41][4].

Additionally, while generating `heongpu::Parameters`, it is necessary to select the key switching method using `Parameters::set_keyswitching_type`[5]. The bit choice and the size of P (i.e., $\lceil \log P \rceil$) may differ depending on the chosen switchkey method. For instance, if Method I is used, P consists of a single prime, which must be larger than each prime in the RNS bases used for$Q$[6].However, if you plan

[4]Standard deviation of $\sigma = 3.2$ and secret key is selected uniformly randomly from the ternary field with elements $\{-1, 0, 1\}$

[5]Method I, Method II, and Method III in Section 4.2

[6]Our implementation allows users to enter the bit sizes of the primes in Q and P and it generates primes of the given bit sizes. Note that $\log P \geq \max \log q_i$ and when $\log P = \max \log q_i$, then $P > \max q_i$.

Bit-length of $\tilde{Q} = QP$			
$\log N$	128-bit security	192-bit security	256-bit security
12	109	74	57
13	218	149	115
14	438	300	232
15	881	605	465
16	1761	1212	930

Table 4: Bit-length of $\tilde{Q}$ for different ring dimensions (i.e., $\log n$) and security levels.

to use Method II or Method III (see Section 4.2 for switchkey methods), P may have to contain multiple primes; otherwise, the method will not function correctly or may return an error. Necessary details are provided in the examples in the library.

The `heongpu::HEKeyGenerator` class is instantiated with `heongpu::Parameters` and includes the functions necessary for generating large keys. Since HEonGPU supports many operations, including key generation, on the GPU, it also stores all keys in GPU memory. In future versions, these keys will optionally be stored on the CPU. The main reason for this is that Galois keys occupy significant space on the GPU, particularly for large polynomial sizes and RNS base numbers. The classes for these key types are provided in the following

- **heongpu::Secretkey**: Stores secret key parameters and data.
- **heongpu::Publickey**: Stores public key parameters and data.
- **heongpu::Relinkey**: Stores relinearization key parameters and data.
- **heongpu::Galoiskey**: Stores multiple Galois-key (Permutation of the secret key) parameters and data.
- **heongpu::Switchkey**: Stores switch-key parameters and data.

Normally, the `heongpu::Galoiskey` class contains permutations of the secret key with respect to powers of 2, as specified by the default values in `define.h`. This approach allows calculating all necessary permutations with a logarithmic complexity of $\log N$ rather than storing a Galois-keys for each rotation value (which would result in N, otherwise), albeit with increased error. However, since applications typically require only a specific and relatively small number of Galois keys, this class can be utilized to store only the necessary values.

Similar to the aforementioned key classes, there are classes representing messages, plaintexts, and ciphertexts. Since all operations are performed on the GPU, the data associated with these classes also resides in GPU memory. For this purpose, `heongpu::devicevector` described in Section 4.7 is used. As explained before all data remains in GPU memory unless the user explicitly transfers it back to the CPU. However, it can be manually brought to CPU memory when needed. Future versions will include more efficient and hybrid approaches, allowing data to be stored both on CPU and GPU. In the current version, this process requires manual management by the user. In other words, if the GPU memory is insufficient for the application, it is the user's responsibility to manage this. Since the data initially resides on CPU, it can be transferred from CPU to GPU. The overhead of data transfer is mitigated by using multi-stream techniques, allowing data transfer to occur simultaneously with ongoing operations.

There are classes that enable encoding, decoding, encryption, decryption, and homomorphic operations. The `heongpu::HEEncoder` class provides the necessary features and functions for both encoding and decoding. The `heongpu::HEEncryptor` class allows for encryption, but currently, it only supports asymmetric encryption; symmetric encryption will be added in the future. The `heongpu::HEDecryptor` class has the necessary features and functions for decryption and includes invariant noise estimation for BFV, as explained Section 4.5.

Now, we can discuss the most important class namely, `heongpu::HEOperator`, which includes functions that allow all homomorphic operations for both BFV and CKKS. These functions and their descriptions are listed as follows

- **HEOperator::add**: Homomorphic addition of two ciphertexts.
- **HEOperator::sub**: Homomorphic subtraction of two ciphertexts.

- **HEOperator::negate**: Homomorphic negation of a ciphertext.
- **HEOperator::add_plain**: Homomorphic addition of a ciphertext and a plaintext.
- **HEOperator::sub_plain**: Homomorphic subtraction of a ciphertext and a plaintext. As the order is important, the first operand must be ciphertext.
- **HEOperator::multiply**: Homomorphic multiplication of two ciphertexts.
- **HEOperator::multiply_plain**: Homomorphic multiplication of a ciphertext and a plaintext.
- **HEOperator::relinearization**: Relinearization of the non-linear part of a ciphertext.
- **HEOperator::rotate_rows**: Rotation of a ciphertext by a given shift amount.
- **HEOperator::rotate_columns**: Switching columns of a ciphertext (, which works only for BFV).
- **HEOperator::keyswitch**: Homomorphic generation of a ciphertext encrypted with another key given a ciphertext.
- **HEOperator::rescale**: Dividing of the ciphertext polynomial to the last modulus in the current RNS basis (, which works only for CKKS).
- **HEOperator::mod_drop**: Reducing the ciphertext or plaintext polynomial modulus by removing the last modulus in RNS bases (, which works only for CKKS)
- and others

5 Experimental Results and Comparison

In this section, we present the performance of the HEonGPU library for certain set of homomorphic operations and applications. We begin by describing our experimental setup, followed by a comparison of the HEonGPU library's performance against CPU-based libraries such as SEAL [5] and OpenFHE [6], using different ring dimensions and schemes. Additionally, we use the HEonGPU library in three different real-world applications to assess their performance: Private Information Retrieval (PIR) [42], Hybrid Homomorphic Encryption (HHE) [43], and Homomorphic ML inference [44].

5.1 Setup

Our experimental setup is configured with a specific blend of hardware and software components. The tests were conducted on a Linux system running Ubuntu, equipped with an AMD Ryzen 9 7950X3D CPU and an NVIDIA GeForce RTX 4090 GPU, supported by 128GB of RAM, running kernel version 5.15.0, and utilizing CUDA SDK version 12.1. Detailed specifications and key features of the devices are provided in Table 5. Given the comparison between a GPU-based library and CPU-based libraries, we chose the best CPU and GPU configurations available to standard users. Additionally, the libraries used for comparison, along with their respective versions, include Microsoft SEAL v4.1.2 [5] and OpenFHE v1.2.0 [6]. For each library, i.e., HEonGPU, Microsoft SEAL, and OpenFHE, identical encryption parameters were used, ensuring consistency in security levels, polynomial degrees, and scaling factors.

Feature	CPU	GPU
Model	Ryzen 9 7950X3D	GeForce RTX 4090
# of Cores	16(32 Threads)	16384
Frequency	4.20 GHz	2520 MHz
RAM	128 GB (3600 MHz)	24 GB
L1 Cache	64 KB (per core)	128 KB (per SM)
L2 Cache	1 MB (per core)	72 MB
L3 Cache	128 MB	-
Bandwidth	-	1.01 TB/s
Thermal Power	120 W	450 W

Table 5: System Specifications

5.2 Comparative Performance Analysis with Other Libraries

We compare the execution times of HEonGPU with Microsoft SEAL and OpenFHE specifically for homomorphic encryption primitives. In the performance comparison presented in this section, the timing results for HEonGPU focus solely on execution times, as the data is already resident in the GPU memory. However, in the subsequent sections, we also include data transfer timings to reflect real-world application scenarios. Despite this, the overhead introduced by data transfer remains minimal due to the efficient use of multi-stream processing, which helps overlap computation and communication. The timing results in Table 6 clearly indicate that HEonGPU outperforms SEAL across all homomorphic encryption operations.

HEonGPU's execution times are significantly lower, particularly in more complex tasks such as homomorphic multiplication and relinearization operations[7]. For operations such as encryption, decryption, addition, and multiplication, HEonGPU achieves even much faster results as the $\log N$ and $\log \tilde{Q}$ increase. The inherent parallelism of the GPU architecture enables significantly faster execution times, particularly in operations that involve large data or complex computations, which is the main motivation that HEonGPU is specifically developed as a GPU-optimized library for FHE.

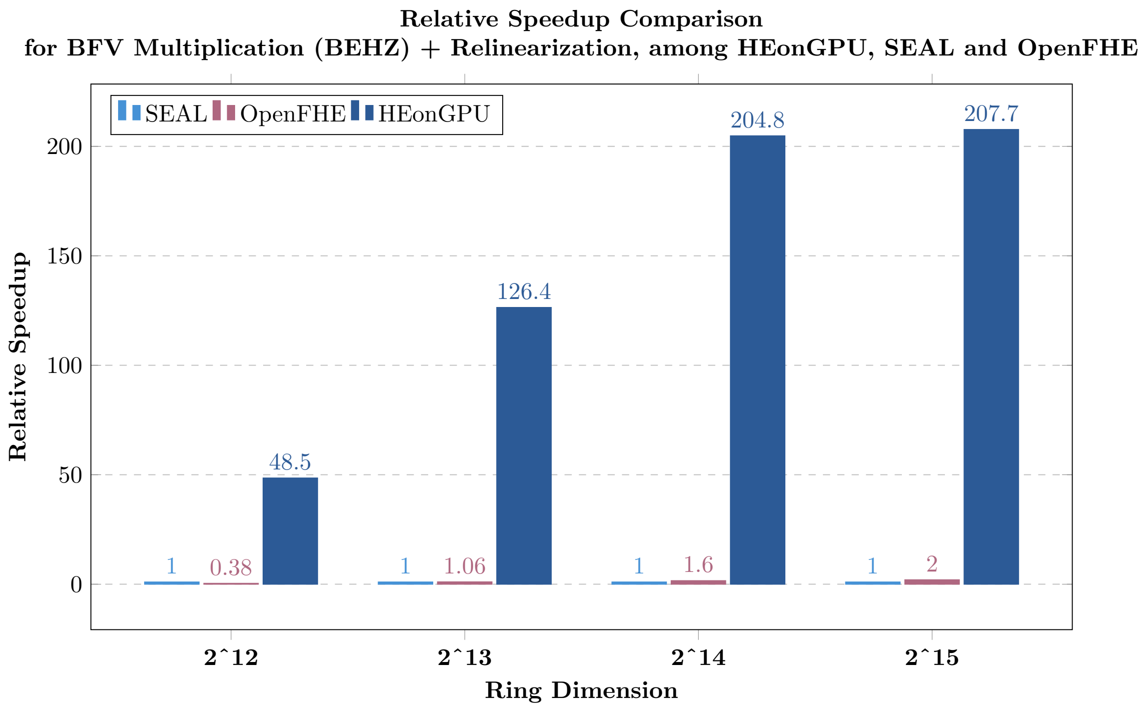

Figure 1: Relative Speedup Comparison for BFV Multiplication (BEHZ) + Relinearization, among HEonGPU, SEAL and OpenFHE

Figure 1 illustrates the relative speedup achieved by HEonGPU compared to SEAL[8] and OpenFHE[9] for BFV multiplication (BEHZ variant) + relinearization operations[10], across varying ring dimensions. The ring dimensions are represented in the range from 2^{12} up to 2^{15}, and the speedup is measured relative to SEAL, which is normalized to 1 for each ring size. As shown in Figure 1, HEonGPU exhibits a significant performance advantage over both SEAL and OpenFHE across all ring sizes. For smaller ring dimensions, such as 2^{12}, HEonGPU achieves a speedup factor of 48.5, demonstrating that HEonGPU is substantially faster for this operation. As the ring dimension increases, the performance gap widens further, with HEonGPU reaching a speedup factor of 204.8 at 2^{14} and 207.7 at 2^{15}.

[7]Method I used for relinearization operation for both library.
[8]Single thread on CPU
[9]Single thread on CPU
[10]Method II used for relinearization operation for HEonGPU

Operation	$\log N$	$\log \tilde{Q}$	HEonGPU		SEAL		Speedup	
			BFV	CKKS	BFV	CKKS	BFV	CKKS
Encryption	12	109	22.77 μs	31.01 μs	780 μs	1204 μs	×34.25	×38.82
	13	218	25.23 μs	35.62 μs	2005 μs	3335 μs	×79.47	×93.62
	14	438	40.36 μs	57.92 μs	7049 μs	10749 μs	×174.65	×185.58
	15	881	88.93 μs	148.82 μs	29382 μs	38738 μs	×330.39	×260.30
	16	1761	382.17 μs	631.09 μs	- μs	- μs	×	×
Decryption	12	109	21.85 μs	3.88 μs	215 μs	46 μs	×9.84	×11.85
	13	218	24.04 μs	4.08 μs	710 μs	147 μs	×29.53	×36.03
	14	438	33.06 μs	5.22 μs	2922 μs	508 μs	×88.38	×97.31
	15	881	64.08 μs	12.12 μs	11336 μs	1914 μs	×176.90	×157.92
	16	1761	202.47 μs	78.34 μs	- μs	- μs	×	×
Addition/Subtraction	12	109	3.66 μs	3.89 μs	9 μs	13 μs	×2.46	×3.34
	13	218	3.81 μs	4.06 μs	29 μs	39 μs	×7.61	×9.60
	14	438	4.83 μs	5.32 μs	112 μs	134 μs	×26.10	×25.18
	15	881	7.99 μs	8.59 μs	426 μs	495 μs	×53.31	×57.62
	16	1761	27.27 μs	38.28 μs	- μs	- μs	×	×
Plain Add/Plain Sub	12	109	3.63 μs	3.89 μs	22 μs	6 μs	×6.06	×1.56
	13	218	3.80 μs	4.09 μs	72 μs	20 μs	×18.94	×4.89
	14	438	4.29 μs	4.13 μs	273 μs	68 μs	×63.63	×16.46
	15	881	6.79 μs	8.48 μs	4010 μs	253 μs	×590.57	×29.83
	16	1761	15.30 μs	46.06 μs	- μs	- μs	×	×
Multiplication	12	109	32.80 μs	3.94 μs	2204 μs	116 μs	×67.19	×29.44
	13	218	57.53 μs	4.64 μs	7450 μs	372 μs	×129.49	×80.17
	14	438	179.61 μs	6.14 μs	33675 μs	1350 μs	×187.49	×219.87
	15	881	746.04 μs	12.57 μs	150110 μs	4630 μs	×201.20	×368.33
	16	1761	5131.93 μs	53.92 μs	- μs	- μs	×	×
Plain Multiplication	12	109	25.05 μs	3.48 μs	360 μs	53 μs	×14.37	×15.23
	13	218	26.33 μs	3.99 μs	1329 μs	169 μs	×50.74	×42.35
	14	438	41.07 μs	5.01 μs	5590 μs	621 μs	×136.10	×123.95
	15	881	97.28 μs	9.11 μs	22537 μs	2087 μs	×231.67	×229.08
	16	1761	361.71 μs	26.62 μs	- μs	- μs	×	×
Relinearization	12	109	22.44 μs	38.90 μs	487 μs	784 μs	×21.70	×20.15
	13	218	28.38 μs	49.77 μs	2280 μs	3391 μs	×80.33	×68.13
	14	438	86.94 μs	124.84 μs	14197 μs	18799 μs	×163.30	×150.58
	15	881	501.08 μs	801.61 μs	85526 μs	105000 μs	×170.68	×130.98
	16	1761	4289.73 μs	5640.22 μs	- μs	- μs	×	×
Rescale	12	109	- μs	21.19 μs	- μs	217 μs	×	×10.24
	13	218	- μs	22.84 μs	- μs	752 μs	×	×32.94
	14	438	- μs	32.27 μs	- μs	2911 μs	×	×90.20
	15	881	- μs	68.19 μs	- μs	11191 μs	×	×164.11
	16	1761	- μs	315.19 μs	- μs	- μs	×	×
Rotate Row	12	109	22.58 μs	45.68 μs	493 μs	803 μs	×21.83	×17.57
	13	218	30.23 μs	59.68 μs	2290 μs	3459 μs	×75.75	×57.95
	14	438	104.48 μs	154.83 μs	14161 μs	18887 μs	×135.53	×121.98
	15	881	554.11 μs	910.84 μs	86749 μs	105731 μs	×156.55	×116.08
	16	1761	4565.97 μs	6130.17 μs	- μs	- μs	×	×

Table 6: Execution times of the HEonGPU and SEAL library operations and speedup values.

Figure 1 highlights the scalability and efficiency of HEonGPU in performing both homomorphic multiplication and relinearization, particularly as the problem size grows. The parallel processing capabilities of the GPU allow for increasingly greater speed improvements as the ring dimension increases, showcasing HEonGPU's superior performance over CPU-based libraries of SEAL and OpenFHE.

5.3 Performance Improvements for Homomoprhic Applications

In this section, we explore the practical use of HEonGPU in real-world scenarios, demonstrating its effective use in applications requiring high-performance homomorphic processinf of data. These applications, namely private information retrieval (PIR), hybrid homomorphic encryption (HHE), and privacy-preserving Iinference on genomic data utilizing XGBoost decision trees, are detailed in the subsequent sections.

5.3.1 Private Information Retrieval (PIR)

Private Information Retrieval (PIR) is a cryptographic protocol that enables a client to retrieve an item from a database without revealing which item is being accessed. The goal is to protect the client's privacy while still allowing access to large, remote databases. Traditional PIR techniques rely on sending queries to multiple servers, which complicates deployment. However, modern PIR schemes focus on single-server solutions using FHE, which allows queries to be encrypted and evaluated without decrypting them.

In an FHE-based PIR scheme, the database query is encrypted and sent to the server, which then performs operations on the encrypted data to retrieve the desired record. The result is returned to the client, still in encrypted form, where it is decrypted to reveal the requested information. Throughout the process, the server remains oblivious to the specific data the client is requesting. This approach is made practical by compressing FHE ciphertexts, reducing the computational overhead while ensuring data privacy [45, 46].

We used SealPIR [42], which is publicly available on GitHub[11], to demonstrate HEonGPU's performance in PIR applications. The methodology remains unchanged; we simply replaced SEAL's operations with HEonGPU's operations to measure performance improvements. No modifications were made to the underlying PIR methodology itself.

We used the parameters provided in work [42], specifically: **number_of_items** $= 2^{16}$, **size_per_item** $= 288$ bytes, and $\mathbf{d} = 2$. Additionally, we assumed that the entire database resides in the GPU for this evaluation. In future work, we will implement scenarios with larger databases where the data is stored on the CPU. However, having the database on the CPU is not expected to introduce significant overhead due to the use of multi-stream processing, which will efficiently manage data transfers.

Table 7 presents a comparison between the performance of PIR applications running on SEAL (single-threaded) and HEonGPU (single GPU with the default stream). The results show a relatively low speedup value at a ring size of 4096, primarily due to insufficient GPU utilization at smaller ring sizes. However, as the ring size increases, the performance improvement becomes more pronounced. When comparing the best performance of SEAL with that of HEonGPU, we observe a speedup of **252.0/4.48 = 56.25**. Same initial noise budget is used across all ring sizes, and the primary goal of increasing the ring size is to improve batching efficiency. However, as increasing the ring size also increases the size of the query, it causes potential network overhead.

Ring Dimension (N)	4096	8192	16384	32768
SEAL (ms)	252.0	352.0	444.0	633.0
HEonGPU (ms)	17.11	9.15	5.99	4.48
Speedup	×14.72	×38.46	×74.12	×141.29

Table 7: Runtime of a single PIR operation (single thread CPU and single GPU with default stream) with respect to different ring dimensions.

To better highlight the performance difference between the GPU and CPU, we implemented a scenario, in which all resources in the CPU are utilized. Since the current PIR implementation is not inherently suitable for parallelization within itself, we ensured that each thread on the CPU was tasked with executing independent PIR operations. Simultaneously, on the GPU, we maximized resource usage by employing multi-stream processing. Table 8 compares the results from this experiment, showcasing the PIR execution times for SEAL (using 16 threads on the CPU) and HEonGPU (using a single GPU with two streams).

Ring Dimension (N)	**PIR**
SEAL (ms)	349.0
HEonGPU (ms)	56.49
Speedup	×6.18

Table 8: Runtime of 16 PIR operations (multi thread CPU and single GPU with multi stream) with respect to different ring dimensions.

[11] https://github.com/microsoft/SealPIR

While SEAL completes 16 PIR operations in approximately 349 ms, HEonGPU significantly outperforms it, completing the same operations in 56.49 ms, achieving a speedup of 6.18[12]. However, when examining the results from a latency perspective, the difference is even more pronounced. On the CPU, the latency per operation remains around 349 ms, whereas on the GPU, with two streams, the latency for completing two PIR operations is just 7.06 ms. This results in a latency improvement of more than ×**49.43**, emphasizing the efficiency gains provided by GPU parallelization and multi-stream processing.

5.3.2 Hybrid Homomorphic Encryption (HHE)

Hybrid homomorphic encryption (HHE) is an approach that seeks to combine the benefits of both symmetric encryption and homomorphic encryption to optimize performance, particularly in bandwidth usage and computational efficiency. The core concept of HHE is to use homomorphic encryption for computations on encrypted data while employing symmetric encryption methods to reduce ciphertext size. This strategy minimizes the amount of data that needs to be transferred, significantly reducing network costs, which is often a major challenge in traditional homomorphic encryption schemes due to large ciphertext sizes. To further ensure efficient and fast homomorphic decryption, fast symmetric cryptographic schemes optimized for HHE are proposed.

In a Hybrid Homomorphic Encryption (HHE) setup, data is first encrypted using a symmetric encryption scheme, and then the symmetric key is encrypted homomorphically. This approach minimizes the need for homomorphic encryption on the entire dataset, reducing both computational and network costs, as only the much shorter symmetric key is processed homomorphically. The encrypted data is then sent to the location where homomorphic operations will occur, and the ciphertext is decrypted homomorphically with symmetric key (an operation is referred here as HE_DEC), allowing the data to be decrypted and processed securely. This method efficiently balances security with performance by combining symmetric encryption for data handling and homomorphic encryption for secure key management [43]..

We implemented the PASTA-3 homomorphic decryption operation using the HEonGPU library. For this, we integrated HEonGPU into the existing work [43] codebase[13] without making any modifications to the underlying methodology. The only change was substituting the original homomorphic operations with HEonGPU's optimized operations, leveraging the GPU for improved performance.

In the comparison of the PASTA-3 cipher implemented with both SEAL (single thread CPU) and HEonGPU (single GPU default stream), we observe significant performance improvements when leveraging GPU acceleration through HEonGPU. Table 9 showcases two different parameter sets, $N = 2^{14}$ and $N = 2^{15}$. For homomorphic decryption, HEonGPU demonstrates a speedup of ×88.5 and ×171.7 over SEAL for $N = 2^{14}$ and $N = 2^{15}$, respectively. Since identical parameters were used for both libraries, the remaining noise budgets are consistent after the homomorphic decryption operation. The primary advantage of utilizing different ring dimensions lies in managing the required noise budget for homomorphic evaluation, allowing users to determine the appropriate size for their secret key encryption based on noise budget requirements. This flexibility is crucial for optimizing both performance and security in homomorphic encryption applications.

			SEAL		**HEonGPU**				
Cipher	**log N**	t	ENC KEY	HE DEC	ENC KEY	HE DEC	Remain Noise B.	S_1	S_2
PASTA-3	14	65537	7.95 ms	4677.28 ms	1.89 ms	52.82 ms	96 bit	×4.2	×88.5
PASTA-3	15	65537	28.68 ms	20404.89 ms	3.45 ms	118.82 ms	525 bit	×8.3	×171.7

Table 9: 1 PASTA-3 Runtime (single thread CPU and single GPU with default stream) and noise budget of the small HHE use case in the SEAL and HEonGPU library (security level = 128 bit).

In order to highlight the performance differences between GPU and CPU, a scenario was designed to fully utilize the CPU's resources. Since the current PASTA-3 homomorphic decryption implementation is not inherently suitable for parallelization within itself as in the case of HHE implementation, we ensured that each thread on the CPU was tasked with executing independent HHE operations. On the GPU side, multi-stream processing was employed to maximize resource utilization, allowing simultaneous execution of multiple tasks and further improving performance.

Table 10 provides a comparison between SEAL (16-thread CPU) and HEonGPU (single GPU with multi-stream) for PASTA-3 homomorphic decryption. On the SEAL side, 16 threads were used, where

[12]In the comparison, $N = 2^{12}$, which yielded the best performance in SEAL, and $N = 2^{15}$, which produced the best result in HEonGPU, and same modulus set were used. This ensures that each library is evaluated based on its optimal parameter configurations, highlighting the best possible performance for each in the specific PIR use case.

[13]https://github.com/IAIK/hybrid-HE-framework

each CPU core performed an independent HE_DEC operation. In contrast, HEonGPU utilized 4 streams, with each stream sequentially processing 4 HE_DEC operations, maximizing the parallel processing capabilities of the GPU.

			SEAL	**HEonGPU**		
Cipher	**log N**	t	HE DEC	HE DEC	Remain Noise B.	S
PASTA-3	14	65537	5704.02 ms	356.99 ms	96 bit	×15.97
PASTA-3	15	65537	30997.23 ms	1570.41 ms	525 bit	×19.74

Table 10: 16 PASTA-3 Homomorphic Decryption Runtime (multi thread CPU and single GPU with multi stream)

The results demonstrate a substantial performance improvement in favor of HEonGPU. For $N = 2^{14}$, SEAL, utilizing 16 threads, required 5704.02 ms to complete the decryption, whereas HEonGPU performed the same 16 HE_DEC operations in just 356.99 ms, achieving a ×15.97 speedup. Similarly, for $N = 2^{15}$, SEAL took 30997.23 ms, while HEonGPU completed the task in 1570.41 ms, yielding a ×19.74 speedup. Additionally, the latency for each operation on the CPU remains approximately 5704.02 ms, whereas on the GPU, using multi-stream processing, the latency for completing two operations drops to around 89.24 ms. This results in a latency improvement factor of ×**63.91**, further underscoring the efficiency gains provided by GPU parallelization and multi-stream processing.

5.3.3 Privacy-Preserving Inference on Genomic Data Utilizing XGBoost Decision Trees

XGBoost, a decision tree based machine learning algorithm, employs ensembles of Extreme Gradient Boosting. The model comprises classification trees that are constructed based on training data, and each tree in the ensemble classifies the test data into one of its leaves. The final prediction score is obtained by summing the numerical values assigned by each tree. Mağara et al. in [44] proposed a privacy-preserving framework for gradient boosting inference, leveraging homomorphic encryption via the SEAL library to classify encrypted genomic data from various tumor types. Basically in their model, to minimize the model's complexity and reduce the circuit depth for homomorphic evaluation, shallow trees were chosen. We adapted their CPU-based framework to HEonGPU to evaluate the performance of HEonGPU without making any modifications to their original implementation.

As detailed in [44], the test data is encrypted, and the homomorphic evaluation of the XGBoost trees is performed on 258 test samples. This evaluation involves 1290 homomorphic multiplications, 1,806 rotations, 1806 subtractions, 1290 plaintext multiplications, 3354 additions, and 2322 relinearization operations. These computations are critical in determining the efficiency and feasibility of applying homomorphic encryption to real-world machine learning tasks.

		SEAL		**HEonGPU**		
log N	**log Q**	Single T.	Multi T.	GPU	S_1	S_2
13	218 bit	14.05 s	1.11 s	0.107 s	×131.31	×10.37
14	438 bit	69.27 s	6.85 s	0.522 s	×132.70	×13.12

Table 11: Machine Learning Model Perfomance and Comparison on SEAL and HEonGPU. S_1 :The ratio of single-thread results over HEonGPU. S_2: The ratio of multi-thread results over HEonGPU

To assess the framework's performance, we executed the inference on both GPU and CPU, utilizing all possible optimization and parallelization strategies for the CPU implementation. As shown in Table 11, the HEonGPU library offers considerable acceleration (including data transfer time) compared to the CPU implementation (based on SEAL). Specifically, the GPU achieved a speedup of ×131.31 and ×132.70 for ring sizes of 8192 and 16384, respectively, relative to the single-threaded CPU execution. Even with the multi-threaded CPU version, the GPU still outperformed, achieving speedups of ×10.37 and ×13.12 for ring sizes of 8192 and 16384, respectively.

These results emphasize the effectiveness of HEonGPU in accelerating computationally intensive operations, such as homomorphic encryption-based XGBoost inference on encrypted genomic data. The ability to efficiently evaluate such models homomorphically shows that HEonGPU can be alternative to CPU-based FHE libraries for accelerating privacy-preserving machine learning.

6 Future Plan

In the future, several key improvements are planned to further enhance HEonGPU's functionality and usability. The library will expand its support for additional homomorphic encryption schemes, introduce bootstrapping capabilities, and improve user accessibility through high-level interfaces. These developments aim to make HEonGPU more versatile and suitable for a wider range of applications, including secure multi-party computations. The planned additions are listed below:

- BGV and TFHE schemes will be added to expand the cryptographic capabilities of the library.
- CKKS bootstrapping will be introduced first, followed by bootstrapping for other schemes.
- A Python wrapper will be developed to increase accessibility and usability for users.
- Threshold encryption will be implemented to support MPC (Multi-Party Computation), allowing for secure collaborative computations.

Acknowledgement

The authors would like to acknowledge the support of this work by the European Union's Horizon Europe research and innovation programme under grant agreement number 101079319.

References

[1] "Concrete: Tfhe compiler that converts python programs into fhe equivalent," https://github.com/zama-ai/concrete, 2022, accessed page 3.

[2] J. H. Cheon, A. Kim, M. Kim, and Y. Song, "Heaan," https://github.com/snucrypto/HEAAN, 2016, accessed page 3.

[3] S. Halevi and V. Shoup, "Helib - an implementation of homomorphic encryption," https://github.com/shaih/HElib/, accessed Feb 2014.

[4] EPFL-LDS and T. I. SA, "Lattigo v5," Online: https://github.com/tuneinsight/lattigo, November 2023, accessed page 3.

[5] "Microsoft seal," https://github.com/Microsoft/SEAL, 2020, accessed page 3.

[6] A. A. Badawi, A. Alexandru, J. Bates, F. Bergamaschi, D. B. Cousins, S. Erabelli, N. Genise, S. Halevi, H. Hunt, A. Kim, Y. Lee, Z. Liu, D. Micciancio, C. Pascoe, Y. Polyakov, I. Quah, S. R.V., K. Rohloff, J. Saylor, D. Suponitsky, M. Triplett, V. Vaikuntanathan, and V. Zucca, "Openfhe: Open-source fully homomorphic encryption library," Cryptology ePrint Archive, Paper 2022/915, 2022, https://eprint.iacr.org/2022/915. [Online]. Available: https://eprint.iacr.org/2022/915

[7] Y. Polyakov, R. Rohloff, G. W. Ryan, and D. Cousins, "Palisade lattice cryptography library (release 1.11.5)," https://palisade-crypto.org/, pp. 3, 4, 15, September 2021, https://gitlab.com/palisade/palisade-release/-/blob/master/doc/palisade_manual.pdf.

[8] NVIDIA Corporation, "Cuda toolkit documentation," https://developer.nvidia.com/cuda-toolkit, 2023, accessed May 2023.

[9] J. Fan and F. Vercauteren, "Somewhat practical fully homomorphic encryption," *IACR Cryptology ePrint Archive*, vol. 2012, p. 144, 2012, accessed pages 4, 6. [Online]. Available: https://eprint.iacr.org/2012/144

[10] J. H. Cheon, A. Kim, M. Kim, and Y. Song, "Homomorphic encryption for arithmetic of approximate numbers," in *International Conference on the Theory and Application of Cryptology and Information Security*. Springer, 2017, pp. 409–437, accessed page 4.

[11] J. H. Cheon, K. Han, A. Kim, M. Kim, and Y. Song, "Bootstrapping for approximate homomorphic encryption," Cryptology ePrint Archive, Paper 2018/153, 2018, https://eprint.iacr.org/2018/153. [Online]. Available: https://eprint.iacr.org/2018/153

[12] H. Chen, I. Chillotti, and Y. Song, “Improved bootstrapping for approximate homomorphic encryption,” Cryptology ePrint Archive, Paper 2018/1043, 2018. [Online]. Available: https://eprint.iacr.org/2018/1043

[13] J. H. Cheon, K. Han, and M. Hhan, “Faster homomorphic discrete fourier transforms and improved FHE bootstrapping,” Cryptology ePrint Archive, Paper 2018/1073, 2018. [Online]. Available: https://eprint.iacr.org/2018/1073

[14] J.-P. Bossuat, C. Mouchet, J. Troncoso-Pastoriza, and J.-P. Hubaux, “Efficient bootstrapping for approximate homomorphic encryption with non-sparse keys,” Cryptology ePrint Archive, Paper 2020/1203, 2020. [Online]. Available: https://eprint.iacr.org/2020/1203

[15] R. Geelen and F. Vercauteren, “Bootstrapping for bgv and bfv revisited,” Cryptology ePrint Archive, Paper 2022/1363, 2022, https://eprint.iacr.org/2022/1363. [Online]. Available: https://eprint.iacr.org/2022/1363

[16] J. Kim, J. Seo, and Y. Song, “Simpler and faster bfv bootstrapping for arbitrary plaintext modulus from ckks,” Cryptology ePrint Archive, Paper 2024/109, 2024, https://eprint.iacr.org/2024/109. [Online]. Available: https://eprint.iacr.org/2024/109

[17] Z. Brakerski, C. Gentry, and V. Vaikuntanathan, “(leveled) fully homomorphic encryption without bootstrapping,” *ACM Transactions on Computation Theory (TOCT)*, vol. 6, no. 3, pp. 1–36, 2014, accessed pages 4, 5.

[18] I. Chillotti, N. Gama, M. Georgieva, and M. Izabachène, “Tfhe: Fast fully homomorphic encryption over the torus,” Cryptology ePrint Archive, Paper 2018/421, 2018, https://eprint.iacr.org/2018/421. [Online]. Available: https://eprint.iacr.org/2018/421

[19] K. Boudgoust and P. Scholl, “Simple threshold (fully homomorphic) encryption from LWE with polynomial modulus,” Cryptology ePrint Archive, Paper 2023/016, 2023. [Online]. Available: https://eprint.iacr.org/2023/016

[20] A. Jain, P. M. R. Rasmussen, and A. Sahai, “Threshold fully homomorphic encryption,” Cryptology ePrint Archive, Paper 2017/257, 2017. [Online]. Available: https://eprint.iacr.org/2017/257

[21] J. H. Cheon, K. Han, A. Kim, M. Kim, and Y. Song, “Privacy preserving federated learning using ckks homomorphic encryption,” in *International Conference on the Theory and Application of Cryptology and Information Security (ASIACRYPT)*. Springer, 2019, pp. 360–384.

[22] J. Ma, S. A. Naas, S. Sigg, and X. Lyu, “Privacy-preserving federated learning based on multi-key homomorphic encryption,” *International Journal of Intelligent Systems*, 2022.

[23] P. Trebicki and S. Grabowski, “Modular multiplication and base extensions in residue number systems,” in *2010 13th Euromicro Conference on Digital System Design: Architectures, Methods and Tools*, 2010, pp. 217–220.

[24] ——, “Fast base extension using a redundant modulus in rns,” *IEEE Transactions on Computers*, vol. 42, no. 8, pp. 1011–1014, 1993.

[25] A. Şah Özcan and E. Savaş, “Two algorithms for fast gpu implementation of ntt,” Cryptology ePrint Archive, Paper 2023/1410, 2023, https://eprint.iacr.org/2023/1410. [Online]. Available: https://eprint.iacr.org/2023/1410

[26] J.-C. Bajard, J. Eynard, A. Hasan, and V. Zucca, “A full RNS variant of FV like somewhat homomorphic encryption schemes,” Cryptology ePrint Archive, Paper 2016/510, 2016. [Online]. Available: https://eprint.iacr.org/2016/510

[27] S. Halevi, Y. Polyakov, and V. Shoup, “An improved RNS variant of the BFV homomorphic encryption scheme,” Cryptology ePrint Archive, Paper 2018/117, 2018. [Online]. Available: https://eprint.iacr.org/2018/117

[28] A. Kim, Y. Polyakov, and V. Zucca, “Revisiting homomorphic encryption schemes for finite fields,” Cryptology ePrint Archive, Paper 2021/204, 2021. [Online]. Available: https://eprint.iacr.org/2021/204

[29] J. H. Cheon, K. Han, A. Kim, M. Kim, and Y. Song, “A full RNS variant of approximate homomorphic encryption,” Cryptology ePrint Archive, Paper 2018/931, 2018. [Online]. Available: https://eprint.iacr.org/2018/931

[30] J.-P. Bossuat, C. Mouchet, J. Troncoso-Pastoriza, and J.-P. Hubaux, “Efficient bootstrapping for approximate homomorphic encryption with non-sparse keys,” Cryptology ePrint Archive, Paper 2020/1203, 2020. [Online]. Available: https://eprint.iacr.org/2020/1203

[31] A. Kim, Y. Polyakov, and V. Zucca, “Revisiting homomorphic encryption schemes for finite fields,” Cryptology ePrint Archive, Paper 2021/204, 2021. [Online]. Available: https://eprint.iacr.org/2021/204

[32] M. Kim, D. Lee, J. Seo, and Y. Song, “Accelerating HE operations from key decomposition technique,” Cryptology ePrint Archive, Paper 2023/413, 2023, https://eprint.iacr.org/2023/413. [Online]. Available: https://eprint.iacr.org/2023/413

[33] C. Gentry, “A Fully Homomorphic Encryption Scheme,” Ph.D. dissertation, Stanford University, 2009, https://crypto.stanford.edu/craig.

[34] NVIDIA Corporation, *CURAND Library User Guide*, 2021, https://docs.nvidia.com/cuda/curand/index.html.

[35] P. L’Ecuyer, “Good parameters and implementations for combined multiple recursive random number generators,” *Operations Research*, vol. 47, no. 1, pp. 159–164, 1999.

[36] I. M. Sobol, “On the distribution of points in a cube and the approximate evaluation of integrals,” *USSR Computational Mathematics and Mathematical Physics*, vol. 7, no. 4, pp. 86–112, 1967.

[37] M. Matsumoto and T. Nishimura, “Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator,” *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, vol. 8, no. 1, pp. 3–30, 1998.

[38] N. Ferguson, B. Schneier, and T. Kohno, *Cryptography Engineering: Design Principles and Practical Applications*. John Wiley & Sons, 2010.

[39] RAPIDS AI, *RAPIDS Memory Manager (RMM) Documentation*, 2020, https://github.com/rapidsai/rmm.

[40] NVIDIA Corporation, *CUDA C++ Programming Guide*, 2021, https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html.

[41] M. R. Albrecht, R. Player, and S. Scott, “On the concrete hardness of learning with errors,” Cryptology ePrint Archive, Paper 2015/046, 2015. [Online]. Available: https://eprint.iacr.org/2015/046

[42] S. Angel, H. Chen, K. Laine, and S. Setty, “PIR with compressed queries and amortized query processing,” Cryptology ePrint Archive, Paper 2017/1142, 2017. [Online]. Available: https://eprint.iacr.org/2017/1142

[43] C. Dobraunig, L. Grassi, L. Helminger, C. Rechberger, M. Schofnegger, and R. Walch, “Pasta: A case for hybrid homomorphic encryption,” Cryptology ePrint Archive, Paper 2021/731, 2021. [Online]. Available: https://eprint.iacr.org/2021/731

[44] S. S. Magara, C. Yildirim, F. Yaman, B. Dilekoglu, F. R. Tutas, E. Öztürk, K. Kaya, Ö. Tastan, and E. Savas, “ML with HE: privacy preserving machine learning inferences for genome studies,” *CoRR*, vol. abs/2110.11446, 2021. [Online]. Available: https://arxiv.org/abs/2110.11446

[45] C. Gentry and S. Halevi, “Compressible FHE with applications to PIR,” Cryptology ePrint Archive, Paper 2019/733, 2019. [Online]. Available: https://eprint.iacr.org/2019/733

[46] S. J. Menon and D. J. Wu, “Spiral: Fast, high-rate single-server PIR via FHE composition,” Cryptology ePrint Archive, Paper 2022/368, 2022. [Online]. Available: https://eprint.iacr.org/2022/368