GPU Acceleration of High-Precision Homomorphic Computation Utilizing Redundant Representation

Shintaro Narisada
KDDI Research, Inc.
Saitama, Japan
sh-narisada@kddi.com

Hiroki Okada
KDDI Research, Inc.
Saitama, Japan
ir-okada@kddi.com

Kazuhide Fukushima
KDDI Research, Inc.
Saitama, Japan
ka-fukushima@kddi.com

Shinsaku Kiyomoto
KDDI Research, Inc.
Saitama, Japan
sh-kiyomoto@kddi.com

Takashi Nishide
University of Tsukuba
Ibaraki, Japan
nishide@risk.tsukuba.ac.jp

ABSTRACT

Fully homomorphic encryption (FHE) can perform computations on encrypted data, allowing us to analyze sensitive data without losing its security. The main issue for FHE is its lower performance, especially for high-precision computations, compared to calculations on plaintext data. Making FHE viable for practical use requires both algorithmic improvements and hardware acceleration. Recently, Klemsa and Önen (CODASPY'22) presented fast homomorphic algorithms for high-precision integers, including addition, multiplication and some fundamental functions, by utilizing a technique called redundant representation. Their algorithms were applied on TFHE, which was proposed by Chillotti et al. (Asiacrypt'16).

In this paper, we further accelerate this method by extending their algorithms to multithreaded environments. The experimental results show that our approach performs 128-bit addition in 0.41 seconds, 32-bit multiplication in 4.3 seconds, and 128-bit Max and ReLU functions in 1.4 seconds using a Tesla V100S server.

CCS CONCEPTS

• **Security and privacy** → **Public key encryption**; • **Computer systems organization** → **Multicore architectures**.

KEYWORDS

FHE; redundant binary; GPU acceleration

1 INTRODUCTION

Fully homomorphic encryption (FHE), proposed by Gentry [17], has attracted attention in the research domain of privacy-preserving applications since it can carry out arbitrary operations in encrypted form. However, it remains a challenge to efficiently perform high-precision operations on encrypted data. BGV [6], BFV [16], and CKKS [7], which are called leveled homomorphic schemes, rely on polynomial evaluations or arithmetic approximations to perform arbitrary functions. It has been shown that the time complexity of polynomial evaluations grows exponentially with increasing precision (see, e.g. [24]). For CKKS, since bit precision decreases in the process of performing operations, developers must pay attention to the lower bit precision bound throughout the entire process [14].

Another type of FHE is called the fast bootstrapping scheme, which started with GSW [18] and was developed into FHEW [15] and TFHE [8, 9]. Bootstrapping is an essential procedure for FHE, which contains noise in its ciphertext. Repeatedly performing homomorphic operations on ciphertexts will increase noise and cause decryption errors. Through bootstrapping, the noise in the ciphertext can be homomorphically initialized. A main feature of TFHE is its ability to evaluate an arbitrary discrete function (with a look-up table (LUT)) at no additional cost during bootstrapping. This procedure is called programmable bootstrapping (PBS) [11] or functional bootstrapping [4]. Unlike conventional gate bootstrapping [9], which deals with binary ciphertexts, PBS can support multi-bit ciphertexts.

Since the plaintext space per ciphertext is limited in practice, it is more challenging to efficiently evaluate arbitrary arithmetic operations and functions with high precision. For instance, the plaintext space for TFHE can be at most 8 bits in the latest implementation. It is not straightforward to evaluate arbitrary high-precision operations efficiently by combining small-precision ciphertexts only.

To compute *any* function with arbitrary precision in TFHE, a few methods have been proposed: CMux-Tree [9], tree-based PBS [4, 19], and WoP-PBS [3]. However, all of these methods fundamentally require an exponential number of PBS or CMux executions. For some arithmetic operations and discrete functions, there are several methods of computing high-precision operations efficiently: the chaining method [19] and arithmetic algorithms with radix-based encoding [9, 19], CRT-based encoding [3], and redundant representation (RR) encoding [13]. Among the above encoding techniques, both RR- and CRT-based encodings offer parallel addition. Furthermore, RR encoding can construct some fundamental functions such as comparison and max functions because RR encoding preserves the numerical order of multiple ciphertexts.

In this work, we focus on homomorphic algorithms with RR encoding, since it is multithread friendly and acceleration by a GPU can be effective for high-precision numbers. We extend existing algorithms with RR encoding to multithreaded environments and implement them efficiently using the CUDA language. The algorithms we target are as follows: homomorphic addition, scalar multiplication, multiplication, the sign function, comparison, max, the rectified linear unit (ReLU) and a conversion function from binary to RR encoding. We also provide comparative experimental results with up to 128-bit precision under fair conditions and an open-source code for third-party evaluation[1]. To the best of our knowledge, this is the first hardware acceleration and evaluation

[1]Our code is publicly available at https://github.com/sh-narisada/cuParmesan.

method for algorithms on TFHE with RR encoding. We believe that our work will contribute to the practical realization of FHE on high-precision integers.

The rest of the paper is organized as follows. Section 2 describes the notation and preliminaries. In Section 3, we describe previous works. Section 4 presents our extended algorithms. We evaluate our acceleration performance in Section 5. Section 6 gives concluding remarks.

2 PRELIMINARIES

We summarize the symbols used in this work in Table 1. An LWE sample and a TLWE sample are defined as follows.

Table 1: Notation

Notation	Description
$\boldsymbol{a}$	Vector
$\langle \boldsymbol{a}, \boldsymbol{b} \rangle$	Inner product of two vectors $\boldsymbol{a}$ and $\boldsymbol{b}$
$\mathbb{Z}_q = \mathbb{Z}/q\mathbb{Z}$	Quotient ring modulo $q \in \mathbb{Z}$
$\mathbb{T} = \mathbb{R}/\mathbb{Z} = [0, 1)$	Torus
$\mathbb{B} = \mathbb{Z}_2 = \{0, 1\}$	Binary
$\mathbb{X}^m$	Set of m-dimensional vectors consisting of elements from a set $\mathbb{X}$
$a \xleftarrow{\$} \mathbb{X}$	a is sampled uniformly from the set $\mathbb{X}$
$a \leftarrow \mathcal{N}(\mu, \sigma^2)$	a is a real number sampled from a normal distribution with mean μ and variance σ^2

Definition 2.1 (LWE Sample). An LWE sample is a pair $(\boldsymbol{a}, b) \in \mathbb{Z}_q^{m+1}$, where $b = \langle \boldsymbol{a}, \boldsymbol{s} \rangle + \lfloor e \rceil \bmod q$, $\boldsymbol{a} \xleftarrow{\$} \mathbb{Z}_q^m$, $\boldsymbol{s} \xleftarrow{\$} \mathbb{B}^m$ and $e \leftarrow \mathcal{N}(0, \sigma^2)$.

Definition 2.2 (TLWE Sample). A TLWE sample is a pair $(\boldsymbol{a}, b) \in \mathbb{T}^{m+1}$, where $b = \langle \boldsymbol{a}, \boldsymbol{s} \rangle + e \bmod 1$, $\boldsymbol{a} \xleftarrow{\$} \mathbb{T}^m$, $\boldsymbol{s} \xleftarrow{\$} \mathbb{B}^m$ and $e \leftarrow \mathcal{N}(0, \sigma^2)$.

The only difference between the LWE and TLWE samples is whether the element is in $\mathbb{Z}_q$ or $\mathbb{T}$. In the TFHE implementation, we always use a discrete torus with fixed precision (e.g., 32-bit or 64-bit). In this paper, any real number from a discrete torus is converted to an integer in $\mathbb{Z}_q$. Namely, a discrete torus with q elements $\{0/q, 1/q, \ldots, (q-1)/q\} \subset \mathbb{T}$ is regarded as $\mathbb{Z}_q$. We define an LWE ciphertext as follows:

Definition 2.3 (LWE Ciphertext). An LWE ciphertext is $\boldsymbol{c} = (\boldsymbol{a}, b) \in \mathbb{Z}_q^{m+1}$, where $b = \langle \boldsymbol{a}, \boldsymbol{s} \rangle + \Delta x + \lfloor e \rceil$, $\boldsymbol{a} \xleftarrow{\$} \mathbb{Z}_q^m$, $\boldsymbol{s} \xleftarrow{\$} \mathbb{B}^m$ and $e \leftarrow \mathcal{N}(0, \sigma^2)$.

We call $\boldsymbol{s}$ the secret key. We use m from vector $\boldsymbol{s}$ as the key length. $x \in \mathbb{Z}_p$ is a plaintext with a $\log_2 p$-bit plaintext space. $\Delta = q/p$ denotes a scaling factor, which moves x to the most significant bit (MSB) of the LWE ciphertext. In this paper, we set $\log_2 q = 64$ (the ciphertext space is 64-bit) and $\log_2 p = 5$ (the plaintext space is 5-bit without padding), and we do not switch between different plaintext space sizes. An LWE ciphertext that encrypts x is denoted by $\mathsf{LWE}(x)$. $\mathsf{LWE}(\boldsymbol{x}) = [\mathsf{LWE}(x_0), \ldots, \mathsf{LWE}(x_{n-1})]$ is a list of LWE ciphertexts encrypted from a list of plaintexts $\boldsymbol{x} = [x_0, \ldots, x_{n-1}]$.

Ring-LWE (RLWE) is a generalization of LWE from a quotient to a polynomial ring. An RLWE ciphertext with an N-degree polynomial encrypts an N-dimensional plaintext vector. An RLWE ciphertext that encrypts a plaintext vector $\boldsymbol{\ell} \in \mathbb{Z}^N$ is written as $\mathsf{RLWE}(\boldsymbol{\ell})$. Typically, N is chosen from among $2^9, 2^{10}, 2^{11}$ and 2^{12}. TFHE uses both LWE and RLWE ciphertexts for encryption.

TFHE supports elementwise homomorphic addition and elementwise homomorphic scalar multiplication: given two ciphertexts $\mathsf{LWE}(x_1)$ and $\mathsf{LWE}(x_2)$, we can see that $\mathsf{LWE}(x_1) + \mathsf{LWE}(x_2) = \mathsf{LWE}(x_1 + x_2)$. For a ciphertext $\mathsf{LWE}(x)$ and a (small) plaintext k, we have $k \cdot \mathsf{LWE}(x) = \mathsf{LWE}(k \cdot x)$. To achieve full homomorphism, we need an additional operation called programmable bootstrapping (PBS).

Definition 2.4 (PBS). The inputs of PBS are an LWE ciphertext $\mathsf{LWE}(x)$, a bootstrapping key bsk and an RLWE ciphertext $\mathsf{RLWE}(\boldsymbol{\ell})$ that encrypts a vector $\boldsymbol{\ell}$ representing a discrete negacyclic function (LUT) f whose output is $\log_2 p$-bit. PBS outputs $\mathsf{LWE}(f(x))$.

PBS consists of three procedures: modulus switching, blind rotation and sample extraction. For a negacyclic function, it satisfies $f(x + p/2) = -f(x), x \in \mathbb{Z}_{\pm p/2} = \{-p/2, \ldots, p/2 - 1\}$ ($\mathbb{Z}_{\pm p/2}$ is a signed representation of $\mathbb{Z}_p$). We use the same notation for the function f as in [20]. Namely, a $\log_2 p$-bit negacyclic function $f : \mathbb{Z}_{\pm p/2} \to \mathbb{Z}_{\pm p/2}$ is written as a list. Let $\mathbf{f} \in \mathbb{Z}_{\pm p/2}^{p/2}$ be the list of function values for the input range $[0, p/2 - 1]$. The remaining half, for the input range $[-p/2, -1]$, is given by negacyclicity. For instance, given $\mathbf{f} = [0, 1, 0, -1]$ with $p = 4$, the negacyclic part is defined as $[0, -1, 0, 1]$. We also use the ellipsis $\|$ when we want to explicitly show a negative part of the function. For example, $\mathbf{f} = [0, 1 \,\|\, 0, 1]$ is an alternative representation of the same function for the input range $[0, 1 \,\|\, -2, -1]$. If f is a non-negacyclic function, we can convert it to a negacyclic function using an additional bit.

PBS changes the secret key from $\boldsymbol{s} \in \mathbb{B}^m$ to $\boldsymbol{s}' \in \mathbb{B}^N$ by homomorphically re-encrypting the LWE ciphertext $\mathsf{LWE}(x)$ with the new secret key $\boldsymbol{s}'$. This initializes the noise accumulated in the LWE ciphertext encrypted with the old key $\boldsymbol{s}$. To restore the original key length, LWE-to-LWE key switching (KS) is used.

Definition 2.5 (KS). The inputs of KS are an LWE ciphertext $\mathsf{LWE}(x)$ encrypted by a secret key $\boldsymbol{s}'$ and a key-switching key ksk. The KS outputs $\mathsf{LWE}(x)$ encrypted by a secret key $\boldsymbol{s}$.

In this paper, we follow the DP-KS-PBS scheme in [3, 11]. The dot product (DP) represents any sequence composed of elementwise homomorphic addition, scalar multiplication and other elementwise operations such as sign negation and substitution between a list of LWE ciphertexts. We apply KS after DP by switching the key from $\boldsymbol{s}'$ to $\boldsymbol{s}$. Immediately after KS, PBS is executed to initialize the noise accumulated by DP and KS and restore the key from $\boldsymbol{s}$ to $\boldsymbol{s}'$.

3 RELATED WORK

This section describes about redundant representation (RR), parallel addition and parallel homomorphic algorithms with RRs proposed by Klemsa and Önen [20].

3.1 Redundant Representation and Parallel Addition

For a base $\beta \geq 2$, we call $\boldsymbol{x} = [x_{n-1}, \ldots, x_1, x_0] \in \{-\beta+1, -\beta+2, ..., \beta-1\}^n$ the base-β RR of $X \in \mathbb{Z}$ iff

$$X = \sum_{i=0}^{n-1} \beta^i x_i. \tag{1}$$

We may write $\boldsymbol{x}$ as a string $x_{n-1} \cdots x_1 x_0$. For example, $\bar{1}10$ is a base-2 RR of -2, where $\bar{1}$ denotes -1. We assume that $x_i = 0$ for $i < 0$. The base-β RR is a generalization of the standard base-β representation. There exist multiple RRs for an integer X.

Addition with RRs can eliminate carry propagation and allow parallelization. Avizienis presented a parallel algorithm for addition using RRs with $\beta = 10$ [2]. Later, Chow and Robertson improved the algorithm with base-2 RRs [12].

We consider the case of $\beta = 2$, which maximizes the number of parallels. Let $\mathbb{A} = \{\bar{1}, 0, 1\}$ be an integer set of an RR. Each element of $\mathbb{A}$ is encrypted as $31, 0, 1$ with an unsigned integer or as $-1, 0, 1$ with a signed integer using plaintext space $\log_2 p = 5$ on the multivalue TFHE.

3.2 Homomorphic Operations with RR Encoding

In this section, we briefly explain homomorphic algorithms with the RR encoding presented in [20].

3.2.1 Homomorphic Addition.

The inputs of homomorphic addition with RRs are LWE ciphertexts $\mathsf{LWE}(\boldsymbol{x})$ and $\mathsf{LWE}(\boldsymbol{y})$ for the n-bit RR $\boldsymbol{x}, \boldsymbol{y} \in \mathbb{A}^n$, which encode n-bit integers X and Y. The outputs are LWE ciphertexts $\mathsf{LWE}(\boldsymbol{z})$ of the n-bit RR $\boldsymbol{z}$ encoding an integer Z satisfying $Z = X + Y$.

Klemsa and Önen presented several parallel addition algorithms for TFHE using RRs with $\beta = 2$ and $\beta = 4$ [20]. We focus on the most efficient algorithm with $\beta = 2$, described in Algorithm 1. We begin by giving the outline. $\mathsf{LWE}(x)$ is occasionally abbreviated as x for simplicity. First, $w_i \leftarrow x_i + y_i$ is computed in parallel by elementwise addition. Next, q_i is calculated by elementwise addition and scalar multiplication. In Line 4, PBS with a function f_q is applied to each q_i. f_q is a function defined as $[0, 0, 0, 0, 1, 1, 1, 1, 1 \,\|\, -1, -1, -1, -1, 0, 0, 0]$; i.e., $f_q(x) = 1$ if $4 \leq x \leq 8$, $f_q(x) = -1$ if $-8 \leq x \leq -4$, and $f_q(x) = 0$ otherwise.

In the original paper, the number of bits for $\boldsymbol{z}$ was set to $n+1$ to store the carry bit. We unify the number of bits for the input and output as n; i.e., we do not compute w_n, q_n and z_n. If overflow occurs, we address it by simply selecting a larger n. In Line 6, the 2-bit identity function $f_I(x) = x$ for $x \in \{0, 1, -1\}$, defined as $[0, 1, 0, -1]$, is evaluated by PBS. Note that f_I is essential for removing the noise accumulated on w_i. PBS involves $2n$ executions, and the number of parallel operations is n.

Algorithm 1: ParallelAdd [20]

Input: n-bit RRs $\boldsymbol{x}, \boldsymbol{y} \in \mathbb{A}^n$ of $X, Y \in \mathbb{Z}$
Output: n-bit RR $\boldsymbol{z} \in \mathbb{A}^n$ of $Z = X + Y$

1 **for** $i \in [0, n-1]$ **in parallel do**
2 $\quad w_i \leftarrow x_i + y_i$
3 $\quad q_i \leftarrow w_{i-1} + 3w_i$ ▷ sync threads
4 $\quad q_i \leftarrow f_q(q_i)$ ▷ PBS
5 $\quad z_i \leftarrow w_i - 2q_i + q_{i-1}$ ▷ sync threads
6 $\quad z_i \leftarrow f_I(z_i)$ ▷ PBS
7 **return** $\boldsymbol{z}$

3.2.2 Homomorphic Scalar Multiplication.

The inputs of the arithmetic are a plaintext $k \in \mathbb{Z}$ and LWE ciphertexts $\mathsf{LWE}(\boldsymbol{x})$ of the n-bit RR $\boldsymbol{x} \in \mathbb{A}^n$, which encode an n-bit integer $X \in \mathbb{Z}$. The outputs are the LWE ciphertexts $\mathsf{LWE}(\boldsymbol{z})$ of the n-bit RR $\boldsymbol{z}$ encoding an integer Z satisfying $Z = k \times X$. Algorithm 2 describes homomorphic scalar multiplication, which is performed with at most $m/2$ additions, where m is the length of the binary string $\boldsymbol{k}$ of $|k|$. $|k|$ is the absolute value of k. Since $\boldsymbol{k}$ is plaintext, we can skip the addition for $k_i = 0$. Moreover, the number of occurrences of 1 and $\bar{1}$ in $\boldsymbol{k}$ can be reduced by conversion from binary to redundant, as shown in Lines 4 and 5, where $\boldsymbol{s} \sqsubseteq \boldsymbol{k}$ means that $\boldsymbol{s}$ is a substring of $\boldsymbol{k}$.

Homomorphic left shift for LWE ciphertexts $\boldsymbol{x}$ can be performed by popping an LWE ciphertext and appending an LWE ciphertext of zeros: $\mathsf{LWE}(\boldsymbol{x} \ll 1) = [\mathsf{LWE}(x_{n-2}), \ldots, \mathsf{LWE}(x_0), \mathsf{LWE}(0)]$. The sign function $\mathrm{sgn}(k)$ is applied at the end of the algorithm. PBS requires at worst mn executions, and the number of parallel operations is n since we need at most $m/2$ homomorphic additions and each addition requires $2n$ PBSs.

Algorithm 2: ParallelScalarMul [20]

Input: n-bit RR $\boldsymbol{x} \in \mathbb{A}^n$ of $X \in \mathbb{Z}$, $k \in \mathbb{Z}$
Output: n-bit RR $\boldsymbol{z} \in \mathbb{A}^n$ of $Z = k \times X$

1 $\boldsymbol{k} \leftarrow$ standard binary representation of $|k|$ with m bits
2 **while** $01^r \sqsubseteq \boldsymbol{k}$ for some $r \geq 2$ **do**
3 $\quad$ for the largest r, replace 01^r with $10^{r-1}\bar{1}$
4 **end**
5 replace every $\bar{1}1$ with $0\bar{1}$
6 $\boldsymbol{z} \leftarrow 0$
7 **for** $i \in [0, m-1], k_i \neq 0$ **do**
8 $\quad \boldsymbol{z} \leftarrow \mathsf{ParallelAdd}(\boldsymbol{z}, k_i \cdot (\boldsymbol{x} \ll i))$
9 **return** $\mathrm{sgn}(k) \cdot \boldsymbol{z}$

3.2.3 Homomorphic Multiplication.

Klemsa et al. also demonstrated parallel homomorphic multiplication for RRs in their implementation [21]. The input is a pair of LWE ciphertexts $\mathsf{LWE}(\boldsymbol{x})$ and $\mathsf{LWE}(\boldsymbol{y})$, which encode n-bit integers X and Y. The outputs are LWE ciphertexts $\mathsf{LWE}(\boldsymbol{z})$, which encode an integer Z satisfying $Z = X \times Y$. We give the pseudocode in Algorithm 3.

First, we want to compute the homomorphic multiplication of two LWE ciphertexts $\mathsf{LWE}(x_i \cdot y_j)$ for each i, j. $\mathsf{LWE}(x_i \cdot y_j)$ can be calculated by one PBS with $f_\times$ defined as $[0, 0, -1, 0, 1 \,\|\, 1, 0, -1, 0]$. We can verify that $f_\times(3x + y) = x \cdot y$ for $x, y \in \{-1, 0, 1\}$. Then, we sum n vectors $\boldsymbol{w}_j$ for $0 \leq j \leq n-1$ to $\boldsymbol{z}$ by calling the ParallelAdd

algorithm n times. In [21], the authors also applied Karatsuba multiplication to further reduce the time complexity. The required number of PBSs in Algorithm 3 is $3n^2$ since we need n^2 PBSs for $f_\times$ and $2n^2$ PBSs for n parallel additions. The maximum number of parallel operations is n^2 in Line 2.

Algorithm 3: ParallelMul [21]

Input: n-bit RRs $\boldsymbol{x}, \boldsymbol{y} \in \mathbb{A}^n$ of $X, Y \in \mathbb{Z}$
Output: n-bit RR $\boldsymbol{z} \in \mathbb{A}^n$ of $Z = X \times Y$

1 $\boldsymbol{w} \leftarrow 0$ ▷ $n \times n$ matrix
2 **for** $i, j \in [0, n-1]$ **in parallel do**
3 $\quad w_{j,i} \leftarrow f_\times(3x_i + y_j)$ ▷ PBS
4 $\boldsymbol{z} \leftarrow 0$ ▷ n vector
5 **for** $j \in [0, n-1]$ **do**
6 $\quad \boldsymbol{z} \leftarrow \mathsf{ParallelAdd}(\boldsymbol{z}, \boldsymbol{w}_j)$
7 **return** $\boldsymbol{z}$

3.2.4 *Fundamental Functions.*

In addition to the homomorphic arithmetic algorithms described thus far, the authors presented homomorphic sign, comparison, max, and ReLU algorithms in [20]. The next section addresses these algorithms as well. We refer the readers to the original paper [20] for more details.

4 PROPOSED METHOD

Since the homomorphic algorithms described in the previous section can be bitwise parallelized, they are well suited for GPU implementations. Recall that an LWE ciphertext is a vector of length $N+1$, the number of parallels for n-bit operations such as addition can be increased from n to $n(N+1)$ if we perform elementwise parallelization for LWE ciphertexts. Note that unlike multi-core CPUs, GPUs can handle a significantly larger number of threads, making parallel schemes fundamentally different from CPU parallelism.

To implement homomorphic algorithms with RRs for a GPU environment, we use the Concrete-cuda library [25] developed by Zama. All of our implementations are based on the Parmesan library version 0.0.20-alpha [21], which is the official implementation of [20].

4.1 Architecture

We follow the architecture of Concrete-cuda. Data transfer between the CPU (host) and GPU (server) is minimized as much as possible. For example, the keys generated by the host (bsk, ksk) are transferred and stored in the server beforehand. If the set of homomorphic operations to be run on the server is known, the necessary buffers and LUTs can be allocated in GPU memory in advance. During the actual computation process, the input data converted to binary are transferred to the server, where homomorphic operations for RRs are performed. If a binary representation is required in the server, the homomorphic conversion algorithm described later can be used to homomorphically convert the RRs to binary. The outputs are transferred to the host as RRs. Finally, the user decrypts the data and converts them from RRs to desired representations.

The runtime bottleneck in homomorphic operations is PBS. AmortizedPBS is a core procedure of the Concrete-cuda library, which can perform PBS on multiple ciphertexts and LUTs concurrently. The inputs of AmortizedPBS are bsk, ksk, a list of LUTs $[f_0, \ldots, f_{n-1}]$ and the input LWEs $[\mathsf{LWE}(x_0), \ldots, \mathsf{LWE}(x_{n-1})]$. The output is a list of LWEs $[\mathsf{LWE}(f_0(x_0)), \ldots, \mathsf{LWE}(f_{n-1}(x_{n-1}))]$. We develop multithreaded algorithms based on the design concept of minimizing the number of required AmortizedPBS operations as much as possible. Thereafter, each algorithm is evaluated by the number of AmortizedPBS calls.

4.2 Multithreaded Algorithms

In this section, we extend algorithms proposed in [20] for multithreaded environments. Note that all of our algorithms are based on the original algorithms proposed in [20].

4.2.1 *Homomorphic Multiparallel Addition.*

We extend ParallelAdd for multiparallel environments by allowing elementwise parallelization of LWE ciphertext for each procedure. Algorithm 4 describes multiparallel addition for n-bit RRs with 2 AmortizedPBS calls. $\mathsf{LWE}(x)_i$ denotes the i-th element of the LWE ciphertext $\boldsymbol{c} = (\boldsymbol{a}, b) \in \mathbb{Z}^{N+1}$. Namely, $\mathsf{LWE}(x)_i = a_i$ if $0 \le i < N$ and $\mathsf{LWE}(x)_i = b$ if $i = N$. The number of parallels in Line 1 is $n(N+1)$ since we assign each pair (i, j) to each thread in the GPU. In Line 4, AmortizedPBS is executed for the LWE vector $\mathsf{LWE}(\boldsymbol{q})$ of length $n(N+1)$ so that $\boldsymbol{q} \leftarrow f_q(\boldsymbol{q})$ in parallel. To run the algorithm, two additional buffers $\mathsf{LWE}(\boldsymbol{w})$ and $\mathsf{LWE}(\boldsymbol{q})$ of length $n(N+1)$ are required to store the intermediate calculations.

Note that homomorphic subtraction of two integers in the RR can be implemented by $\mathsf{HomMPSub}(\boldsymbol{x}, \boldsymbol{y}) \leftarrow \mathsf{HomMPAdd}(\boldsymbol{x}, -\boldsymbol{y})$, where $-\boldsymbol{y}$ is obtained by negating all elements in $\boldsymbol{y}$: $\mathsf{LWE}(y_i)_j \leftarrow q - \mathsf{LWE}(y_i)_j$ for each i and j. This is because (1) the negation of an RR is obtained by negating all bits, and (2) if a plaintext y is encrypted in $(\boldsymbol{a}, b)$, then $(-\boldsymbol{a}, -b)$ encrypts $-y$. Thus, $\mathsf{HomMPSub}(\boldsymbol{x}, \boldsymbol{y})$ can also be fully parallelized.

Algorithm 4: HomMPAdd

Input: n-bit RRs $\boldsymbol{x}, \boldsymbol{y} \in \mathbb{A}^n$ of $X, Y \in \mathbb{Z}$
Output: n-bit RR $\boldsymbol{z} \in \mathbb{A}^n$ of $Z = X + Y$

1 **for** $i \in [0, n-1], j \in [0, N]$ **in parallel do**
2 $\quad \mathsf{LWE}(w_i)_j \leftarrow \mathsf{LWE}(x_i)_j + \mathsf{LWE}(y_i)_j$
3 $\quad \mathsf{LWE}(q_i)_j \leftarrow \mathsf{LWE}(w_{i-1})_j + 3 \cdot \mathsf{LWE}(w_i)_j$
4 $\boldsymbol{q} \leftarrow \mathsf{AmortizedPBS}(\mathsf{bsk}, \mathsf{ksk}, [f_q] * n, \boldsymbol{q})$ ▷ Parallel PBS
5 **for** $i \in [0, n-1], j \in [0, N]$ **in parallel do**
6 $\quad \mathsf{LWE}(z_i)_j \leftarrow \mathsf{LWE}(w_i)_j - 2 \cdot \mathsf{LWE}(q_i)_j$
7 $\quad \mathsf{LWE}(z_i)_j \leftarrow \mathsf{LWE}(z_i)_j + \mathsf{LWE}(q_{i-1})_j$
8 $\boldsymbol{z} \leftarrow \mathsf{AmortizedPBS}(\mathsf{bsk}, \mathsf{ksk}, [f_I] * n, \boldsymbol{z})$ ▷ Parallel PBS
9 **return** $\boldsymbol{z}$

4.2.2 *Homomorphic Multiparallel Scalar Multiplication.*

Algorithm 5 describes multiparallel scalar multiplication. In Line 2, we use the same conversion algorithm for $\boldsymbol{k}$ as in Algorithm 2. In Line 6, each x_{i-d} is d-bit shifted to the left in parallel by $w_i \leftarrow x_{i-d}$ if $i - d \ge 0$. We discard the d leftmost values by padding the d rightmost values with d zeros. After that, two AmortizedPBS operations are called by HomMPAdd. In total, at most $m/2$ AmortizedPBS operations are required for the algorithm. Note that the number

of AmortizedPBS calls can be further reduced if we use a more sophisticated conversion algorithm for $\boldsymbol{k}$ (e.g. [23]).

Algorithm 5: HomMPScalarMul

Input: n-bit RR $\boldsymbol{x} \in \mathbb{A}^n$ of $X \in \mathbb{Z}$, $k \in \mathbb{Z}$
Output: n-bit RR $\boldsymbol{z} \in \mathbb{A}^n$ of $Z = k \times X$

1 $\boldsymbol{z} \leftarrow \boldsymbol{0}$
2 $\boldsymbol{k} \leftarrow$ compressed pattern of $|k|$ in Algorithm 2 with m bits
3 **for** $d \in [0, m-1], k_d \neq 0$ **do**
4 **for** $i \in [0, n-1], j \in [0, N]$ **in parallel do**
5 **if** $i - d \geq 0$ **then**
6 $\mathsf{LWE}(w_i)_j \leftarrow \mathsf{LWE}(x_{i-d})_j$ ▷ Parallel Shift
7 **else**
8 $\mathsf{LWE}(w_i)_j \leftarrow \mathsf{LWE}(0)_j$ ▷ Parallel Shift
9 $\mathsf{LWE}(w_i)_j \leftarrow k_d \cdot \mathsf{LWE}(w_i)_j$
10 $\boldsymbol{z} \leftarrow \mathsf{HomMPAdd}(\boldsymbol{z}, \boldsymbol{w})$
11 **if** $\mathrm{sgn}(k) = -1$ **then**
12 **for** $i \in [0, n-1], j \in [0, N]$ **in parallel do**
13 $\mathsf{LWE}(z_i)_j \leftarrow -1 \cdot \mathsf{LWE}(z_i)_j$
14 **return** $\boldsymbol{z}$

4.2.3 *Homomorphic Multiparallel Multiplication.*

We perform homomorphic multiplication for n-bit RRs with $1 + 2\log_2 n$ AmortizedPBS calls in Algorithm 6. In Line 3, $u_{jn+i} \leftarrow x_i \cdot y_j$ is computed by AmortizedPBS on n^2 vectors with $f_\times$ in parallel. Then, every j-th vector $\boldsymbol{u}_j \leftarrow \boldsymbol{x} \cdot y_j$ is shifted to the left j times: $\boldsymbol{z}_j \leftarrow \boldsymbol{u}_j \ll j$. Note that we discard elements that are left out by the shift and pad the rightmost values with zeros for each $\boldsymbol{u}_j$. This operation can be fully parallelized using an additional buffer.

Now, we need to take the sum over each vector $\boldsymbol{z}_j$. Parallel reduction is applied to reduce the depth of the homomorphic addition from n to $\log_2 n$. The key here is to generalize HomMPAdd for multiple inputs of RRs. To do so, elementwise addition corresponding to Line 2 of Algorithm 1 is performed for two vectors: $\boldsymbol{w}_j \leftarrow \boldsymbol{z}_{\frac{jn}{2^d}} + \boldsymbol{z}_{\frac{(2j+1)n}{2^{d+1}}}$. In this line, we merge two RRs $\boldsymbol{z}_{\frac{jn}{2^d}}$ and $\boldsymbol{z}_{\frac{(2j+1)n}{2^{d+1}}}$, which correspond to the left and right nodes in the binary tree at depth d. Then, each $\boldsymbol{w}_j$ is concatenated with 0 between them to be processed concurrently later so that the resultant vector after Line 14 is $\boldsymbol{w} = [\boldsymbol{w}_0, 0, \boldsymbol{w}_1, 0, \ldots, 0, \boldsymbol{w}_{2^d-1}, 0]$, which has length $2^d(n+1)$. By inserting one such extra zero between representations in this way, if the i-th element $w_i = \boldsymbol{w}[i]$ of $\boldsymbol{w}$ is the inserted zero, we have $q_i = \boldsymbol{q}[i] = 0$ after Line 17[2], thereby preventing q_{i-1} (= 0) from contaminating v_i in Line 20. Thus, after padding, exactly the same process as in HomMPAdd can be performed on $\boldsymbol{w}$ so that the $\boldsymbol{w}_i$s are processed concurrently. In Line 21, we have a vector containing the sum of two numbers $[v_0, \circ, v_1, \circ, \ldots, \circ, v_{2^d-1}, \circ]$ where the $\circ$s are trash values[3]. For the next depth $d-1$, $\boldsymbol{v}$ is restored to the appropriate position in $\boldsymbol{z}$ by excluding the trash values. Finally, $\boldsymbol{z}_0$ contains the sum of the $\boldsymbol{z}_j$s, i.e., the product of the two integers.

[2]This can be confirmed with a calculation by simply noting that $w_i = 0$ and using the property of f_q in Algorithm 1.
[3]The positions of these trash values correspond to the positions of the inserted zeros.

Algorithm 6: HomMPMul

Input: n-bit RRs $\boldsymbol{x}, \boldsymbol{y} \in \mathbb{A}^n$ of $X, Y \in \mathbb{Z}$
Output: n-bit RR $\boldsymbol{z} \in \mathbb{A}^n$ of $Z = X \times Y$

1 **for** $i, j \in [0, n-1], k \in [0, N]$ **in parallel do**
2 $\mathsf{LWE}(u_{jn+i})_k \leftarrow 3 \cdot \mathsf{LWE}(x_i)_k + \mathsf{LWE}(y_j)_k$
3 $\boldsymbol{u} \leftarrow \mathsf{AmortizedPBS}(\mathsf{bsk}, \mathsf{ksk}, [f_\times] * n^2, \boldsymbol{u})$
4 **for** $i, j \in [0, n-1], k \in [0, N]$ **in parallel do**
5 **if** $i - j \geq 0$ **then**
6 $\mathsf{LWE}(z_{jn+i})_k \leftarrow \mathsf{LWE}(u_{jn+i-j})_k$ ▷ Parallel Shift
7 **else**
8 $\mathsf{LWE}(z_{jn+i})_k \leftarrow \mathsf{LWE}(0)_k$ ▷ Parallel Shift
9 **for** $d \in [\log_2 n - 1, 0]$ **do**
10 **for** $i \in [0, n], j \in [0, 2^d - 1], k \in [0, N]$ **in parallel do**
11 **if** $i \neq n$ **then**
12 $\mathsf{LWE}(w_{j(n+1)+i})_k \leftarrow$ $\mathsf{LWE}(z_{\frac{jn^2}{2^d}+i})_k + \mathsf{LWE}(z_{\frac{(2j+1)n^2}{2^{d+1}}+i})_k$
13 **else**
14 $\mathsf{LWE}(w_{j(n+1)+i})_k \leftarrow \mathsf{LWE}(0)_k$ ▷ Zero Padding
15 **for** $i \in [0, 2^d(n+1)], j \in [0, N]$ **in parallel do**
16 $\mathsf{LWE}(q_i)_j \leftarrow \mathsf{LWE}(w_{i-1})_j + 3 \cdot \mathsf{LWE}(w_i)_j$
17 $\boldsymbol{q} \leftarrow \mathsf{AmortizedPBS}(\mathsf{bsk}, \mathsf{ksk}, [f_q] * 2^d(n+1), \boldsymbol{q})$
18 **for** $i \in [0, 2^d(n+1)], j \in [0, N]$ **in parallel do**
19 $\mathsf{LWE}(v_i)_j \leftarrow \mathsf{LWE}(w_i)_j - 2 \cdot \mathsf{LWE}(q_i)_j$
20 $\mathsf{LWE}(v_i)_j \leftarrow \mathsf{LWE}(v_i)_j + \mathsf{LWE}(q_{i-1})_j$
21 $\boldsymbol{v} \leftarrow \mathsf{AmortizedPBS}(\mathsf{bsk}, \mathsf{ksk}, [f_I] * 2^d(n+1), \boldsymbol{v})$
22 **for** $i \in [0, n], j \in [0, 2^d - 1], k \in [0, N]$ **in parallel do**
23 **if** $i \neq n$ **then** $\mathsf{LWE}(z_{\frac{jn^2}{2^d}+i})_k \leftarrow \mathsf{LWE}(v_{j(n+1)+i})_k$
24 **return** z_0

4.2.4 *Homomorphic Multiparallel Sign and Comparison.*

The sign function can be used as a component to implement the comparison operation, max function, and ReLU function. Bourse et al. proposed homomorphic parallel sign and comparison algorithms for base-β radix representations [5]. Klemsa et al. realized that these algorithms can be directly applied to RRs [20].

We describe a parallel sign algorithm for multiparallel environments using parallel reduction on a binary tree in Algorithm 7. The output of Algorithm 7 is $\mathsf{LWE}(1)$ if $X > 0$, $\mathsf{LWE}(-1)$ if $X < 0$, and $\mathsf{LWE}(0)$ otherwise. Intuitively, the algorithm stores each x_i in a leaf of the binary tree. Each pair of nodes $v_i, v_j (i < j)$ is then merged so that the sign of the node v_j corresponding to the upper digit of the two nodes will be stored in the parent node v'_i. In Line 3, the right node corresponding to the $\frac{ni}{2^d}$-th bit for $\boldsymbol{x}$ and the left node corresponding to the $\frac{(2i+1)n}{2^{d+1}}$-th bit are merged. Then, f_{sgn}, defined as $[0, 1, 1, 1, 0, -1, -1, -1]$, is concurrently applied to each $x_{\frac{ni}{2^d}} + 2x_{\frac{(2i+1)n}{2^{d+1}}}$. After that, the sign of the subtree is stored: $x_{\frac{ni}{2^d}} \leftarrow \mathrm{sgn}(\boldsymbol{x}_{[\frac{ni}{2^d}, \frac{n(i+1)}{2^d})})$. At the end of the loop, $x_0 \leftarrow \mathrm{sgn}(\boldsymbol{x}_{[0,n-1]}) = \mathrm{sgn}([x_{n-1}, \ldots, x_0])$ is computed after $\log_2 n$ calls of AmortizedPBS. We give an example of the parallel reduction algorithm used in HomMPSign in Figure 1.

Algorithm 7: HomMPSign

Input: n-bit RR $\boldsymbol{x} \in \mathbb{A}^n$ of $X \in \mathbb{Z}$
Output: $\mathrm{sgn}(X) \in \{-1, 0, 1\}$

for $d \in [\log_2 n - 1, 0]$ **do**
  for $i \in [0, 2^d - 1], j \in [0, N]$ **in parallel do**
    $\mathsf{LWE}(w_i)_j \leftarrow \mathsf{LWE}(x_{\frac{ni}{2^d}})_j + 2 \cdot \mathsf{LWE}(x_{\frac{(2i+1)n}{2^{d+1}}})_j$
  $\boldsymbol{w} \leftarrow \mathsf{AmortizedPBS}(\mathsf{bsk}, \mathsf{ksk}, [f_{\mathrm{sgn}}] * 2^d, \boldsymbol{w})$
  for $i \in [0, 2^d - 1], j \in [0, N]$ **in parallel do**
    $\mathsf{LWE}(x_{\frac{ni}{2^d}})_j \leftarrow \mathsf{LWE}(w_i)_j$
return x_0

To implement the homomorphic comparison for two integers $\boldsymbol{x}$ and $\boldsymbol{y}$ ($\mathsf{HomMPComparison}(\boldsymbol{x}, \boldsymbol{y})$) in an RR, we first compute the difference of $\boldsymbol{x}$ and $\boldsymbol{y}$ in parallel with $\boldsymbol{z} \leftarrow \mathsf{HomMPSub}(\boldsymbol{x}, \boldsymbol{y})$. Then, $\mathsf{HomMPSign}(\boldsymbol{z}) = \mathsf{HomMPComparison}(\boldsymbol{x}, \boldsymbol{y})$ is satisfied since it outputs $\mathsf{LWE}(1)$ if $\boldsymbol{x} > \boldsymbol{y}$, $\mathsf{LWE}(0)$ if $\boldsymbol{x} = \boldsymbol{y}$, and $\mathsf{LWE}(-1)$ otherwise. $\mathsf{HomMPComparison}(\boldsymbol{x}, \boldsymbol{y})$ requires to perform AmortizedPBS for $2 + \log_2 n$ times.

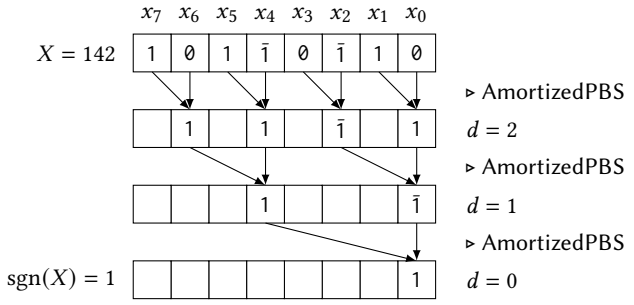

Figure 1: An example of the parallel reduction algorithm used in HomMPSign for an 8-bit RR $\mathtt{10}\bar{\mathtt{1}}\mathtt{0}\bar{\mathtt{1}}\mathtt{10}$ with 3 AmortizedPBS executions. Each $\mathsf{LWE}(x_i)$ is parallelized with N threads.

4.2.5 *Homomorphic Multiparallel Max and ReLU Functions.*

We also extend the homomorphic parallel max algorithm for RRs proposed in [20] by utilizing the AmortizedPBS function as described in Algorithm 8. The output of Algorithm 8 is $\mathsf{LWE}(\boldsymbol{x})$ if $\boldsymbol{x} \geq \boldsymbol{y}$ and $\mathsf{LWE}(\boldsymbol{y})$ otherwise. $\mathsf{HomMPComparison}^+$ uses f_{sgn^+}, defined as $[1, 1, 1, 1, -1 - 1, -1, -1]$ when $d = 0$, instead of f_{sgn} to output $\mathsf{LWE}(1)$ if $\boldsymbol{x} \geq \boldsymbol{y}$ and $\mathsf{LWE}(-1)$ otherwise. Then, we concurrently compute $x_i \leftarrow x_i + 2s$ and $y_i \leftarrow y_i - 2s$ for each i. AmortizedPBS is applied for the concatenation of $\boldsymbol{x}$ and $\boldsymbol{y}$, using the function $f_{\max}$, defined as $[0, -1, 0, 1 \parallel 0, 0, 0]$, so that it outputs x_i (resp. y_i) if $s = 1$ (resp. $s = -1$) and 0 otherwise for each i. Finally, $\max(\boldsymbol{x}, \boldsymbol{y})$ is obtained by performing elementwise parallel addition. We require $3 + \log_2 n$ calls of AmortizedPBS for HomMPMax in total. To compute the ReLU for $\boldsymbol{x}$, we only need to call HomMPMax with $\mathrm{ReLU}(\boldsymbol{x}) = \mathsf{HomMPMax}(\boldsymbol{x}, \boldsymbol{0})$.

4.2.6 *Homomorphic Multiparallel Conversion.*

Due to the redundancy of RRs, performing arithmetic operations on RRs may increase the number of redundant bits. For instance, the redundant addition of $\mathtt{0}\bar{\mathtt{1}}\bar{\mathtt{1}}$ and $\mathtt{100}$ ($-3+4$) yields $\mathtt{01}\bar{\mathtt{1}}$, which is 1 bit longer than the non-redundant representation of one. To remove this redundancy, we want a homomorphic conversion algorithm from redundant to binary. In [20], the authors outlined a conversion algorithm for RRs with base $\beta = 4$. In this work, we present a multithreaded conversion algorithm for $\beta = 2$ with n AmortizedPBS operations in Algorithm 9.

Algorithm 8: HomMPMax

Input: n-bit RRs $\boldsymbol{x}, \boldsymbol{y} \in \mathbb{A}^n$ of $X, Y \in \mathbb{Z}$
Output: n-bit RR $\boldsymbol{z} \in \mathbb{A}^n$ of $Z = \max(X, Y)$

$s \leftarrow \mathsf{HomMPComparison}^+(\boldsymbol{x}, \boldsymbol{y})$ ▷ $s \in \{-1, 1\}$
for $i \in [0, n-1], j \in [0, N]$ **in parallel do**
  $\mathsf{LWE}(x_i)_j \leftarrow \mathsf{LWE}(x_i)_j + 2 \cdot \mathsf{LWE}(s)_j$
  $\mathsf{LWE}(y_i)_j \leftarrow \mathsf{LWE}(y_i)_j - 2 \cdot \mathsf{LWE}(s)_j$
$[\boldsymbol{x}, \boldsymbol{y}] \leftarrow \mathsf{AmortizedPBS}(\mathsf{bsk}, \mathsf{ksk}, [f_{\max}] * 2n, [\boldsymbol{x}, \boldsymbol{y}])$
$\boldsymbol{z} \leftarrow \boldsymbol{x} + \boldsymbol{y}$ ▷ In parallel
return $\boldsymbol{z}$

In the algorithm, we use f_1 and f_2, defined as $[0, 1, 0 \parallel 1]$ and $[0, 1, 0, 0 \parallel 1, 1]$, respectively. It can be shown that the n-bit binary string $\boldsymbol{z}$ of an integer X in the range $-2^{n-1} \leq X \leq 2^{n-1} - 1$ and any RR $\boldsymbol{x}$ of X satisfy $z_i = f_1(x_i + c_i)$ for all $0 \leq i \leq n - 1$ by setting $c_0 = 0$ and $c_{i+1} = f_2(2x_i + c_i)$. The PBS operation for f_1 and f_2 can be run concurrently in AmortizedPBS by inputting the pair of functions $[f_1, f_2]$ and the pair of inputs $[a, b]$. This halves the number of required AmortizedPBS calls from $2n$ to n. We confirmed in our experiments that the runtime of the conversion is reduced by half compared to that of a naive implementation with $2n$ PBSs. Note that a *fully* parallel conversion algorithm does not exist since this concept contradicts the impossibility of parallel addition with a non-redundant representation.

Algorithm 9: HomMPConversion

Input: n-bit RR $\boldsymbol{x} \in \mathbb{A}^n$ of $X \in \mathbb{Z}$
Output: n-bit RR $\boldsymbol{z} \in \mathbb{B}^n$ of X

$c \leftarrow 0$
for $i \in [0, n-1]$ **do**
  for $j \in [0, N]$ **in parallel do**
    $\mathsf{LWE}(a)_j \leftarrow \mathsf{LWE}(x_i)_j + \mathsf{LWE}(c)_j$
    $\mathsf{LWE}(b)_j \leftarrow 2 \cdot \mathsf{LWE}(x_i)_j + \mathsf{LWE}(c)_j$
  $[z_i, c] \leftarrow \mathsf{AmortizedPBS}(\mathsf{bsk}, \mathsf{ksk}, [f_1, f_2], [a, b])$
return $\boldsymbol{z}$

5 EXPERIMENTS

We verify our GPU implementation using one NVIDIA Tesla V100S server with 80 streaming multiprocessors (SMs) as the GPU and one Intel Core i9-12900 with 24 threads as the CPU. Our implementation is based on Parmesan [21] version 0.0.20-alpha[4], Concrete-core [10]

[4]Very recently, Parmesan 0.1.0 was released, which is a more sophisticated version based on the new TFHE-rs library. However, we could not apply the parameter sets listed below due to execution errors. Therefore, we use a previous version in this paper and would like to leave a comparison with the latest version for future work.

Table 2: List of parameters used [22, 26]

Name	Security Level	$\log_2 p$	m	N	ℓ_{PBS}	$\log_2(B_{\mathrm{PBS}})$	ℓ_{KS}	$\log_2(B_{\mathrm{KS}})$	σ^2 for LWE	σ^2 for RLWE
CNCR-M2-C3-128	128	5	776	2048	1	23	5	4	2^{-37}	2^{-104}
TFHE-M2-C0-128	128	2	656	512	2	8	4	3	2^{-29}	2^{-49}
TFHE-M2-C2-128	128	4	742	2048	1	23	5	3	2^{-34}	2^{-103}
TFHE-M2-C3-128	128	5	856	4096	1	22	6	3	2^{-40}	2^{-124}

version 1.0.1, and Concrete-cuda [25] version 0.1.1. We use CUDA 11.0, cargo and rustc version 1.70 to build our implementation.

The TFHE parameter sets used in our experiments are derived from [22] and the TFHE-rs library [26], as shown in Table 2. The security levels are obtained from the lattice-estimator[5] [1].

5.1 Comparison with Parmesan 0.0.2-alpha

Table 3: Runtime of 32/64/128-bit homomorphic addition, scalar multiplication and multiplication compared to that of Parmesan [20] using the parameter CNCR-M2-C3-128.

	Addition [ms]		Scalar Mul. [ms]		Multiplication [s]	
n	[20]	Ours	[20]	Ours	[20]	Ours ($2n$)
32	383	126	1,255	594	21.49	3.00
64	690	142	2,184	628	99.28	11.46
128	1,258	183	5,101	825	400.81	45.66

We compare the runtimes of homomorphic addition, scalar multiplication and multiplication between Parmesan 0.0.2-alpha and our implementation. In this experiment, our implementation also computes $(n+1)$-bit values since Parmesan 0.0.2-alpha outputs $(n+1)$ bits for n-bit addition in the default settings. For scalar multiplication, we set $k = 3195$ (12 bits) as in [22]. For multiplication, we modified our implementation to output $2n$ LWE ciphertexts instead of n to unify the experimental condition with that of Parmesan 0.0.2-alpha. Both implementations used the CNCR-M2-C3-128 parameter. Table 3 shows the results. From the results, we can confirm the acceleration of multithreaded addition for high-precision integers. Furthermore, it has been shown that our GPU implementation provides acceleration for scalar multiplication and multiplication, which consist of multiple parallel additions.

We note that according to the preliminary results for the latest version of Parmesan (version 0.1.0) in [22], they achieved a 284 ms runtime for 32-bit addition using a different security parameter. We leave the comparison with this new version for future work.

5.2 Benchmarks for Other Functions

We measure the runtime of the homomorphic sign, comparison, max (ReLU), and conversion functions with the latest 128-bit security parameter TFHE-M2-C3-128 provided by the TFHE-rs library. Table 4 shows the results. We achieved runtimes of approximately 1 second for the sign, comparison and max functions when operating with 128-bit integers. However, it is observed that the conversion function is costly since a fully parallel algorithm for this function is fundamentally unattainable. Compared with the results in [20], the runtime of the sign function for 31-bit integers, 12 threads, and 90-bit security is 619ms, and that of the max function is 2,028ms.

[5] https://github.com/malb/lattice-estimator

Table 4: Runtime of 32/64/128-bit homomorphic sign, comparison, max (ReLU), and conversion functions using the parameter TFHE-M2-C3-128 in milliseconds.

n	Sign	Comparison	Max (ReLU)	Conversion
32	633	755	926	3,824
64	774	905	1,098	7,603
128	879	1,063	1,415	15,135

5.3 Comparison with a GPU Implementation

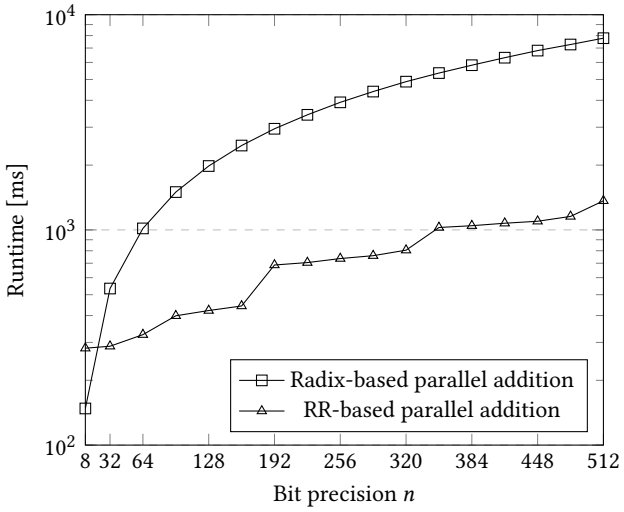

Figure 2: Runtime comparison between multiparallel addition algorithms using radix decomposition [13] with the parameter TFHE-M2-C0-128 and RR with the parameter TFHE-M2-C3-128.

To observe the effect of carry propagation on runtime in the addition algorithm under the same environment, we implemented a homomorphic addition algorithm presented by Clet et al. for a GPU environment with radix decomposition ($\beta = 2$) [13] using the Concrete-cuda library. The pseudocode is presented in the appendix. Although this algorithm requires carry propagation, it can be partially parallelized. We used the TFHE-M2-C0-128 parameter for the algorithm, which is much smaller than TFHE-M2-C3-128

for the RR-based implementation with the same security level. We measured the runtimes for both algorithms up to 512 bits.

The results are shown in Figure 2. Despite the difference in parameters, RR-based addition was found to be more efficient for additions with large-precision integers. We also observed that the runtime growth rate of the algorithm using RR is smaller than that of the radix-based method since the former can parallelize as many threads as possible for a large n.

5.4 Comparison with the TFHE-rs Library

Finally, we compared our implementation with the TFHE-rs library [26], which is a state-of-the-art implementation of TFHE developed by Zama. In this experiment, we used `TFHE-M2-C3-128` for our implementation and `TFHE-M2-C2-128` for TFHE-rs.

Table 5: Runtime of homomorphic addition, scalar multiplication and multiplication compared to TFHE-rs version 0.2.4 [3]. We selected parameters used in TFHE-rs: `TFHE-M2-C3-128` for our implementation and `TFHE-M2-C2-128` for TFHE-rs.

	Addition [ms]		Scalar Mul. [ms]		Multiplication [s]	
n	[3]	Ours	[3]	Ours	[3]	Ours
32	335	**284**	1,652	**1,268**	**2.19**	4.30
64	693	**330**	3,301	**1,469**	**6.95**	15.62
128	1,417	**411**	6,832	**1,950**	**24.11**	70.63

The results can be found in Table 5. For addition and scalar multiplication, our algorithm scales better as precision increases. However, for multiplication, our implementation is less efficient than TFHE-rs. One of the reasons is that our implementation cannot scale well for the Karatsuba algorithm, while TFHE-rs can. This is due to the efficiency of multiple PBSs. We measured the amortized runtime required for one PBS in Table 6. If the number of LWE ciphertexts is sufficiently large, `AmortizedPBS` is faster than TFHE-rs. However, if the number of LWE ciphertexts is small, `AmortizedPBS` is much slower than TFHE-rs. The performance degradation with a low number of threads is a specific challenge for GPUs. One possible solution could be a hybrid approach involving CPU parallelization.

Table 6: Comparison of (amortized) runtime in milliseconds required for one PBS operation under different parameters.

	PBS in TFHE-rs			`AmortizedPBS`		
Param. / #LWEs	1	100	10000	1	100	10000
`TFHE-M2-C2-128`	**19**	**2.03**	**1.75**	66	0.87	0.56
`TFHE-M2-C3-128`	46	4.91	4.22	**151**	**2.08**	**1.26**

6 CONCLUSION

In this paper, we presented a GPU implementation of homomorphic operations with base-2 redundant representation for multibyte integers. Our experimental results show hardware acceleration of some homomorphic arithmetic operations up to 128-bit integers. Future work includes the improvement of multithreaded homomorphic multiplication by utilizing divide-and-conquer methods and pruning unnecessary operations. It is also essential to incorporate the latest TFHE implementations (Parmesan 0.1.0 and TFHE-rs) into hardware implementations and conduct comprehensive experiments comparing them. In addition, it is important to perform a performance comparison with GPU implementations of other FHE schemes and to measure application-level performance.

Acknowledgments. T. Nishide was supported in part by The Telecommunications Advancement Foundation and JSPS KAKENHI Grant Number 23K18459.

REFERENCES

[1] Martin R Albrecht, Benjamin R Curtis, Amit Deo, Alex Davidson, Rachel Player, Eamonn W Postlethwaite, Fernando Virdia, and Thomas Wunderer. 2018. Estimate all the {LWE, NTRU} schemes!. In *International Conference on Security and Cryptography for Networks*. Springer, 351–367.

[2] Algirdas Avizienis. 1961. Signed-digit number representations for fast parallel arithmetic. *IRE Transactions on electronic computers* 3 (1961), 389–400.

[3] Loris Bergerat, Anas Boudi, Quentin Bourgerie, Ilaria Chillotti, Damien Ligier, Jean-Baptiste Orfila, and Samuel Tap. 2023. Parameter Optimization and Larger Precision for (T)FHE. *J. Cryptol.* 36, 3 (2023), 65. https://doi.org/10.1007/s00145-023-09463-5

[4] Christina Boura, Nicolas Gama, Mariya Georgieva, and Dimitar Jetchev. 2019. Simulating Homomorphic Evaluation of Deep Learning Predictions. In *Cyber Security Cryptography and Machine Learning*. 212–230.

[5] Florian Bourse, Olivier Sanders, and Jacques Traoré. 2020. Improved Secure Integer Comparison via Homomorphic Encryption. In *Topics in Cryptology – CT-RSA 2020*. 391–416.

[6] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. 2012. (Leveled) Fully Homomorphic Encryption without Bootstrapping. In *Proceedings of the 3rd Innovations in Theoretical Computer Science Conference (ITCS '12)*. 309–325. https://doi.org/10.1145/2090236.2090262

[7] Jung Hee Cheon, Andrey Kim, Miran Kim, and Yongsoo Song. 2017. Homomorphic Encryption for Arithmetic of Approximate Numbers. In *Advances in Cryptology – ASIACRYPT 2017*. 409–437.

[8] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. 2016. Faster Fully Homomorphic Encryption: Bootstrapping in Less Than 0.1 Seconds. In *Advances in Cryptology – ASIACRYPT 2016*. 3–33.

[9] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. 2020. TFHE: Fast Fully Homomorphic Encryption Over the Torus. *Journal of Cryptology* 33, 1 (2020), 34–91. https://doi.org/10.1007/s00145-019-09319-x

[10] Ilaria Chillotti, Marc Joye, Damien Ligier, Jean-Baptiste Orfila, and Samuel Tap. 2020. CONCRETE: Concrete Operates oN Ciphertexts Rapidly by Extending TFHE. In *WAHC 2020–8th Workshop on Encrypted Computing & Applied Homomorphic Cryptography*, Vol. 15.

[11] Ilaria Chillotti, Marc Joye, and Pascal Paillier. 2021. Programmable Bootstrapping Enables Efficient Homomorphic Inference of Deep Neural Networks. In *Cyber Security Cryptography and Machine Learning*. 1–19.

[12] Catherine Y. Chow and James E. Robertson. 1978. Logical design of a redundant binary adder. In *1978 IEEE 4th Symposium onomputer Arithmetic (ARITH)*. 109–115. https://doi.org/10.1109/ARITH.1978.6155767

[13] Pierre-Emmanuel Clet, Martin Zuber, Aymen Boudguiga, Renaud Sirdey, and Cédric Gouy-Pailler. 2022. Putting up the swiss army knife of homomorphic calculations by means of TFHE functional bootstrapping. *Cryptology ePrint Archive* (2022).

[14] Anamaria Costache, Benjamin R. Curtis, Erin Hales, Sean Murphy, Tabitha Ogilvie, and Rachel Player. 2022. On the precision loss in approximate homomorphic encryption. Cryptology ePrint Archive, Paper 2022/162. (2022). https://eprint.iacr.org/2022/162

[15] Léo Ducas and Daniele Micciancio. 2015. FHEW: Bootstrapping Homomorphic Encryption in Less Than a Second. In *Advances in Cryptology – EUROCRYPT 2015*. 617–640.

[16] Junfeng Fan and Frederik Vercauteren. 2012. Somewhat Practical Fully Homomorphic Encryption. *IACR Cryptol. ePrint Arch.* 2012 (2012), 144.

[17] Craig Gentry. 2009. Fully Homomorphic Encryption Using Ideal Lattices. In *Proceedings of the Forty-First Annual ACM Symposium on Theory of Computing (STOC '09)*. Association for Computing Machinery, New York, NY, USA, 169–178. https://doi.org/10.1145/1536414.1536440

[18] Craig Gentry, Amit Sahai, and Brent Waters. 2013. Homomorphic Encryption from Learning with Errors: Conceptually-Simpler, Asymptotically-Faster, Attribute-Based. In *Advances in Cryptology – CRYPTO 2013*. 75–92.

[19] Antonio Guimaraes, Edson Borin, and Diego F Aranha. 2021. Revisiting the functional bootstrap in TFHE. *IACR Transactions on Cryptographic Hardware and*

Embedded Systems (2021), 229–253.
[20] Jakub Klemsa and Melek Önen. 2022. Parallel Operations over TFHE-Encrypted Multi-Digit Integers. In *Proceedings of the Twelveth ACM Conference on Data and Application Security and Privacy*. 288–299.
[21] Jakub Klemsa and Melek Önen. 2022. Parmesan: Parallel ARithMEticS on TFHE ENcrypted data. (2022). https://github.com/fakub/parmesan/.
[22] Jakub Klemsa and Melek Önen. 2023. PARMESAN: Parallel ARithMEticS over ENcrypted data. Cryptology ePrint Archive, Paper 2023/544. (2023). https://eprint.iacr.org/2023/544
[23] Kenji Koyama and Yukio Tsuruoka. 1993. Speeding up Elliptic Cryptosystems by Using a Signed Binary Window Method. In *Advances in Cryptology — CRYPTO' 92*, Ernest F. Brickell (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 345–357.
[24] Daisuke Maeda, Koki Morimura, and Takashi Nishide. 2022. Efficient Homomorphic Evaluation of Arbitrary Bivariate Integer Functions. In *Proceedings of the 10th Workshop on Encrypted Computing & Applied Homomorphic Cryptography (WAHC'22)*. Association for Computing Machinery, New York, NY, USA, 13–22. https://doi.org/10.1145/3560827.3563378
[25] Zama Team. 2022. Announcing Concrete-core v1.0.0-gamma with GPU acceleration. (2022). https://www.zama.ai/post/concrete-core-v1-0-gamma-with-gpu-acceleration.
[26] Zama. 2022. TFHE-rs: A Pure Rust Implementation of the TFHE Scheme for Boolean and Integer Arithmetics Over Encrypted Data. (2022). https://github.com/zama-ai/tfhe-rs.

A PARALLEL ALGORITHM OF HOMOMORPHIC ADDITION WITH RADIX DECOMPOSITION

The n-bit parallel homomorphic addition algorithm with n calls of AmortizedPBS for base-2 radix representation is described in Algorithm 10. f_a is a function defined as $[0, 1, 0, 1]$ used to compute $(x_i + y_i + c) \bmod 2$, and f_b is a function defined as $[0, 0, 1, 1]$ used to compute $\lfloor (x_i + y_i + c)/2 \rfloor$ for $x_i, y_i, c \in \{0, 1\}$.

Algorithm 10: HomMPRadixAdd

Input: $\boldsymbol{x}, \boldsymbol{y} \in \mathbb{B}^n$ of $X, Y \in \mathbb{Z}$ for some $n \in \mathbb{Z}$
Output: $\boldsymbol{z} \in \mathbb{B}^n$ of $Z = X + Y$

$c \leftarrow 0$
for $i \in [0, n-1]$ **do**
  for $j \in [0, N]$ **in parallel do**
    $\mathsf{LWE}(w)_j \leftarrow \mathsf{LWE}(x_i)_j + \mathsf{LWE}(y_i)_j + \mathsf{LWE}(c)_j$
  $[z_i, c] \leftarrow \mathsf{AmortizedPBS}(\mathsf{bsk}, \mathsf{ksk}, [f_a, f_b], [w, w])$
return z