CPU and GPU Accelerated Fully Homomorphic Encryption

Toufique Morshed*, Md Momin Al Aziz† and Noman Mohammed‡

Computer Science, University of Manitoba

Email: *morshed@cs.umanitoba.ca, †azizmma@cs.umanitoba.ca, ‡noman@cs.umanitoba.ca

Abstract—**Fully Homomorphic Encryption (FHE) is one of the most promising technologies for privacy protection as it allows an arbitrary number of function computations over encrypted data. However, the computational cost of these FHE systems limits their widespread applications. In this paper, our objective is to improve the performance of FHE schemes by designing efficient parallel frameworks. In particular, we choose Torus Fully Homomorphic Encryption (TFHE) [1] as it offers exact results for an infinite number of boolean gate (e.g., AND, XOR) evaluations. We first extend the gate operations to algebraic circuits such as addition, multiplication, and their vector and matrix equivalents. Secondly, we consider the multi-core CPUs to improve the efficiency of both the gate and the arithmetic operations. Finally, we port the TFHE to the Graphics Processing Units (GPU) and device novel optimizations for boolean and arithmetic circuits employing the multitude of cores. We also experimentally analyze both the CPU and GPU parallel frameworks for different numeric representations (16 to 32-bit). Our GPU implementation outperforms the existing technique [1], and it achieves a speedup of $20\times$ for any 32-bit boolean operation and $14.5\times$ for multiplications.**

Index Terms—**Fully Homomorphic Encryption, GPU parallelism, Secure computation on GPU, Parallel FHE Framework**

I. INTRODUCTION

Fully Homomorphic Encryption (FHE) [2] has attracted attention in modern cryptography research. FHE cryptosystems provide strong security guarantee and can compute an infinite number of operations on the encrypted data. Due to the emergence of various data-oriented applications [3, 4, 5] on sensitive data, the idea of computing under encryption has recently gained momentum. FHE is the ideal cryptographic tool that addresses this privacy concern by enabling computation on encrypted data.

Motivating Applications. There has been significant advancements in machine learning techniques and their applications over the last few years. The usage and accuracy of such methods have surpassed the state of the art solutions in manifolds. We can attribute three components behind this improvement: a) better algorithms, b) big data and c) efficient hardware (H/W) enabled parallelism. With the increase of cloud services, several service providers (e.g., Google Prediction API [6], Azure Machine Learning [7]) have combined the three attributes to facilitate machine learning as a service.

In these services, users outsource their data to the cloud server to build a machine learning model. However, data outsourcing exposes the sensitive data to the cloud service provider [8] and thus susceptible to privacy attacks by the employees at the service provider [9]. FHE schemes are practical for such use cases as these schemes facilitate computation on encrypted data. Using FHE, a data owner can encrypt the sensitive data before outsourcing it to the server, and also the server can execute the required machine learning algorithm for data analysis.

A. Current Techniques

The homomorphic encryption schemes can be divided into three major categories: Partially, Somewhat, and FHE schemes. Partially Homomorphic schemes only support one type of operation (e.g., addition or multiplication); such schemes are not useful in performing arbitrary computations on encrypted data.

Somewhat Homomorphic Encryption (SWHE) schemes are more equipped than partially homomorphic encryption schemes. These schemes support both addition and multiplication operations on encrypted data, but for a limited (or predefined) number of times. In addition, these schemes are relatively efficient (see Table I for comparison) and therefore are practical for certain applications. However, even these schemes require complex parameterization and are not powerful enough for more complicated operations such as deep learning.

FHE schemes support both addition and multiplication operations for an arbitrary number of times. This property allows computing any function on the encrypted data. Both SWHE and FHE use the Learning with Error (LWE) paradigm, where an error is introduced with the ciphertext value to guarantee security [18]. This error grows with each operation (especially multiplication) and causes incorrect decryption after a certain number of operations. Therefore, this error needs to be minimized to support arbitrary computation. The process of reducing the error is called Bootstrapping. FHE employs bootstrapping after a certain number of operations resulting in higher computation overhead, while SWHE provides faster execution time by limiting/pre-defining the number of operations on the encrypted data.

The above discussion provides an intuition about the applications of different HE schemes. That is, SWHE is better suited for the applications where the computational depth is shallow and known (/fixed) prior to the computations. However, these schemes are not suitable for applications that require arbitrary depth like deep learning. In order to compute complicated functions like deep learning, the researchers have proposed alternative models that require the existence of a third party [19, 20, 21]. The aim is to minimize the propagated error without executing the costly bootstrapping procedure for SWHE schemes. However, such an assumption (i.e., the existence of a trusted third party) is not always easy to fulfill. In this article, we assume that the computational entity (e.g., cloud server) is standalone, and we show that parallelism can

TABLE I: A comparative analysis of existing Homomorphic Encryption schemes for different parameters on 32-bit number.

	Year	Homomorphism	Bootstrapping	Parallelism	Bit security	Size (kb)	Add. (ms)	Mult. (ms)
RSA [10]	1978	Partial	×	×	128	0.9	×	5
Paillier [11]	1999	Partial	×	×	128	0.3	4	×
TFHE [1]	2016	Fully	Exact	AVX [12]	110	31.5	7044	4,89,938
HEEAN [13]	2018	Somewhat	Approximate	CPU	157	7,168	11.37	1,215
SEAL (BFV) [14]	2019	Somewhat	×	×	157	8,806	4,237	23,954
cuFHE [15]	2018	Fully	Exact	GPU	110	31.5	2,032	1,32,231
NuFHE [16]	2018	Fully	Exact	GPU	110	31.5	4,162	1,86,011
Cigulata [17]	2018	Fully	Exact	×	110	31.5	2,160	50,690
Our Method	-	Fully	Exact	GPU	110	31.5	1,991	33,930

TABLE II: A comparison of the execution times (sec) of TFHE [25] and our CPU, GPU framework for 32-bit numbers

	Gate Op.	Addition		Multiplication	
		Regular	Vector	Regular	Matrix (mins)
TFHE [1]	1.40	7.04	224.31	489.93	8,717.89
CPU-Parallel	0.50	7.04	77.18	174.54	2,514.34
GPU-Parallel	0.07	1.99	11.22	33.93	186.23

TABLE III: Notations used throughout the paper

Notation	Description
$\mathbf{A} \in \mathbb{Z}^{r \times c}$	Integer matrix of dimension $r \times c$
$\overrightarrow{A} \in \mathbb{Z}^{\ell}$	Integer vector of length ℓ
$A, A' \in \mathbb{Z} \subset \mathbb{B}^n$	n-bit integer number and its complement
$a_i \in \mathbb{B}$	Binary bit at position i
$\mathbb{L}^n$	LWE sample vector of length n
n, m	Bit size and Secret key size
$\|, \ll$	Parallel and Left Shift operation
$\wedge, \|, \oplus$	Binary AND, OR and XOR operation

be used to lower the cost of FHE instead of relaxing the security assumptions for the computation model.

Why TFHE? There have been several attempts in improving the asymptotic performance and numerical operations of FHE [22, 23, 24], which are pivotal to this work (Section VIII for details). Torus FHE (TFHE) [1] is one of the most renowned FHE schemes that meets the expectation of arbitrary depth of circuits with faster bootstrapping technique. TFHE also incurs lower storage requirement compared to the other encryption schemes (Table I). The plaintext message space is binary in TFHE. Hence, the computations are based solely on boolean gates, and each gate operation entails a bootstrapping procedure in gate bootstrapping mode.

Why GPU? Most of the FHE schemes are based on the Learning With Errors (LWE), where plaintexts are encrypted using polynomials and can be represented with vectors. Therefore, most computations are operated on vectors that are highly parallelizable. On the contrary, Graphics Processing Units (GPUs) offer a large number of computing cores (compared to CPUs). These cores can be utilized to compute parallel vectors operations. Therefore, we can utilize these cores to parallel the FHE computations. However, we also have to consider the fixed and limited memory of GPUs (8-16GB) and their reduced computing power compared to any CPU core.

B. Contributions

- In this paper, we extended the boolean gate operations (AND, XOR, etc.), from earlier work [1], to higher level algebraic circuits (e.g., addition, multiplications).
- Initially, we constructed a CPU parallel TFHE framework as a baseline to leverage the available computational cores. Experimental results demonstrate the advantage of such construction using multi-threading as they outperform the sequential implementation.
- We devised boolean gates using the GPU parallelism, and employed novel optimizations such as bit coalescing, compound gates, and tree-based additions to implement

the higher level algebraic circuits. We also modified and incorporated parallelism to the existing algorithms: Karatsuba and Cannon for multiplication and matrix counterpart to achieve further speedup. The code is readily available at https://github.com/toufique-morshed/CPU-GPU-TFHE.

- We have conducted several experiments to compare the computation time of the sequential TFHE [1] with our proposed CPU and GPU parallel frameworks. We have also outlined a real-world application with Linear Regression on different datasets [26]. From Table II, our proposed GPU || method is $14.4\times$ and $46.81\times$ faster than the existing technique for regular and matrix multiplications, respectively. We have also benchmarked our performance with existing TFHE frameworks on GPU, namely, cuFHE [15], NuFHE [16].

Notably, the existing GPU enabled TFHE libraries, cuFHE [15] and NuFHE [16], have implemented the TFHE boolean gates using GPUs, whereas our goal was to construct an optimized arithmetic circuit framework. Our design choices and algorithms reflect this improvement and resultantly, our multiplications are around 3.9 and 4.5 times faster than cuFHE and NuFHE, respectively.

II. PRELIMINARIES

A. Torus FHE (TFHE)

In this work, we closely investigate Torus FHE (TFHE) [1] where the plain and ciphertexts are defined over a real torus $\mathbb{T} = \mathbb{R}/\mathbb{Z}$, a set of real numbers modulo 1. The ciphertexts are constructed over Learning with Errors (LWE) [18] and represented as Torus LWE (TLWE) where an error term (sampled over a Gaussian distribution χ) is added to each ciphertext. For a given dimension $m \geq 1$ (key size), secret key $\overrightarrow{S} \in \mathbb{B}^m$ (m-bit binary vector), and error $e \in \chi$, an LWE

sample is defined as $(\overrightarrow{A}, B)$ where $\overrightarrow{A} \in \mathbb{T}^m$ s. t. $\overrightarrow{A}$ is a vector of torus coefficients of length m (key size) and each element A_i is drawn from the uniform distribution over $\mathbb{T}$, and $B = \overrightarrow{A} \cdot \overrightarrow{S} + e$.

The error term (e) in LWE sample grows and propagates with the number of computations (e.g., addition, multiplication). Therefore, bootstrapping is introduced to decrypt and re-encrypt the ciphertexts under encryption to remove the noise.

TFHE considers the binary bits as plaintext and generates LWE samples as ciphertexts. Hence, LWE sample computations in ciphertext are analogous to binary bit computations in plaintext. As a binary vector represents an integer number, an LWE sample vector ($\mathbb{L}^n$) can represent an encrypted integer. For example, an n-bit integer becomes n-LWE sample after encryption. Thus, the boolean gate operations of an addition circuit between two n-bit numbers correspond to the similar operations on LWE samples of encrypted numbers. Throughout this paper, we use *bit* and *LWE sample* interchangeably. Here, we choose TFHE for the following reasons:

- **Fast and Exact Bootstrappping.** TFHE provided the fastest and exact bootstrapping technique which only required around $0.1s$. Some recent encryption schemes [13, 27] also proposed faster bootstrapping and FHE computations in general. However, they do not perform exact bootstrapping and erroneous after successive computations on the same ciphertexts.
- **Ciphertext Size.** Compared to the other HE schemes, TFHE offers smaller ciphertext size as it operates on binary plaintexts as shown in Table I. Nevertheless, this minimal storage advantage allowed us to utilize the limited and fixed memory of GPU when we optimize the gate structures.
- **Boolean Operations.** TFHE also supports boolean operations which are extended to construct arbitrary functions. These binary bits can then be operated in parallel if their computations are independent of each other.

Existing Implementation: The current TFHE implementation comes with the basic cryptographic functions (i.e., encryption, decryption, etc.) and all binary gate operations. Although the gates are computed somewhat sequentially in the original implementation [1], the underlying architecture uses Advanced Vector Extensions (AVX) [12]. AVX is an extension to x86 instruction set from Intel which facilitates parallel vector operations. The bootstrapping procedure requires expensive Fast Fourier Transform (FFT) operations ($O(n \log n)$). The existing implementation uses the Fastest Fourier Transform in the West (FFTW) [28] which inherently uses AVX.

B. Parallelism

Our **CPU Parallel** framework utilizes the CPU cores and existing resources (i.e., Vector Extensions) available in a typical computing machines. We exploit these parallel components on CPUs as we breakdown each algebraic computation into independent parts and distribute them among the available resources. However, one major drawback of the CPU framework

is the limited number of available cores. Contemporary desktop computers come with 4 to 8 cores containing a maximum of 16 threads. There have been multiple attempts [29] to use a large number of CPUs collectively for parallel operations, whereas we show that single GPU is equivalent (and better performing) for most FHE operations.

In contrast to CPUs, **GPU Parallel** framework offers a significant number of cores available solely from the hardware. Thus, the expense of increasing cores with multiple CPUs is reduced by integrating one GPU. Nevertheless, we had to consider the two major shortcomings which are: a) limited global memory and b) communication time through PCIe.

III. Sequential Framework

A. Addition

A carry-ahead 1-bit full adder circuit takes two input bits along with a carry to compute the sum and a new carry that propagates to the next bit's addition. Therefore, in a full adder, we have three inputs as a_i, b_i and c_{i-1}, where i denotes the bit position. Here, the addition of bit a_1 and b_1 in $A, B \in \mathbb{B}^n$ requires the carry bit from a_0 and b_0. This dependency enforces the addition operation to be sequential for n-bit numbers [30].

B. Multiplication

1) Naive Approach: For two n-bit numbers $A, B \in \mathbb{Z}$, we multiply (AND) the number A with each bit $b_i \in B$, resulting in n numbers. Then, these numbers are left shifted by i bits individually resulting in $[n, 2n]$-bit numbers. Finally, we accumulate (reduce by addition) the n shifted numbers using the addition.

2) Karatsuba Algorithm: We consider the divide-and-conquer Karatsuba's algorithm for its improved time complexity $O(n^{\log^3})$ [31]. It relies on dividing the large input numbers and performing smaller multiplications. For n-bit inputs, Karatsuba's algorithm splits them into smaller numbers of $n/2$-bit size and replaces the multiplication by additions and subsequent multiplications (Line 11 of Algorithm 1). Later, we introduce parallel vector operations for further optimizations.

IV. CPU-based Parallel Framework

We propose a CPU ‖ framework utilizing the multiple cores available in computers. Since the existing TFHE implementation uses AVX2, we employ that in our CPU ‖ framework.

A. Addition

Figure 1 illustrates the bitwise addition operation considered in our CPU framework. Here, any resultant bit r_i depends on its previous c_{i-1} bit. The dependency restricts incorporating any data-level parallelism in the addition circuit construction.

Here, it is possible to exploit task-level parallelism where two threads execute the XOR and AND operations (Figure 1), simultaneously. We observed that the time required to perform such fork-and-join between two threads is higher than executing them serially. This is partially due to the costly

Algorithm 1: Karatsuba Multiplication [31]

Input: X, Y $\in \mathbb{B}^n$
Output: Z $\in \mathbb{B}^{2n}$

1 **if** $n < n_0$ **then**
2   **return** BaseMultiplication(X, Y)
3 **end**
4 $X_0 \leftarrow X \mod 2^{n/2}$
5 $Y_0 \leftarrow Y \mod 2^{n/2}$
6 $X_1 \leftarrow X/2^{n/2}$
7 $Y_1 \leftarrow Y/2^{n/2}$
8 $Z_0 \leftarrow$ KaratsubaMultiply(X_0, Y_0)
9 $Z_1 \leftarrow$ KaratsubaMultiply(X_1, Y_1)
10 $Z_2 \leftarrow$ KaratsubaMultiply($X_0 + Y_0$, $X_1 + Y_1$)
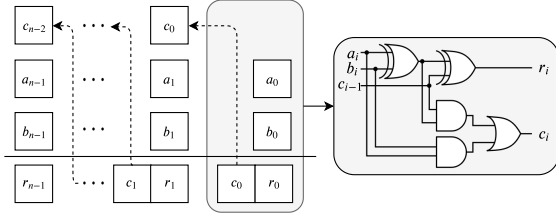11 **return** $Z \leftarrow Z_0 + (Z_2 - Z_1 - Z_0) \, 2^n + (Z_1)2^{2n}$

Fig. 1: Bitwise addition of two n-bit numbers A and B. a_i, b_i, c_i, r_i are i^{th}-bit of $A, B, carry$, and the $result$

thread operations and eventual serial dependency of the results. Hence, we did not employ this technique for CPUs.

B. Multiplication

Out of the three major operations (AND, left shift, and accumulation in multiplications, the AND and left shifts can be executed in parallel. For example, for any two 16-bit numbers A, B ($\in \mathbb{B}^{16}$), and four available threads, we divide the AND and left shift operation among four threads.

On the other hand, the accumulation operation is demanding as it requires n (for n-bit multiplication) additions. The accumulation operation adds and stores values to the same variable, which makes it atomic. Therefore, all threads performing the previous AND and left shift have to wait for such accumulation which is termed as global thread synchronization [32]. Given that it is computationally expensive, we do not employ this technique in any parallel framework.

We utilized a custom reduction operation in OpenMP [32], which uses the global shared memory (CPU) to store the in-between results. This customized reduction foresees additions of any results upon completion and facilitates a performance gain by avoiding the global thread synchronization.

C. Vector Operations

To compute the vector operations (addition, multiplication) efficiently, we distribute the work into multiple threads. For example, $\overrightarrow{A}$, $\overrightarrow{B}$, and $\overrightarrow{C} \in \mathbb{Z}^\ell$ are three vectors of length ℓ, where $\overrightarrow{C} = \overrightarrow{A} + \overrightarrow{B}$, and each element A_i, B_i, or C_i are n-bit integers. The computation of each position of $\overrightarrow{C}$ is

independent. Hence, we can share the work among different threads. We take similar measures for multiplication as well.

D. Matrix Operations

Matrix Addition is a series of addition operations between the elements of two matrices. The addition operations between them are independent of each other. Therefore, we divide the matrices row-wise and distribute the additions among threads. *Matrix Multiplication* is a bit more complicated than addition. It consists of both multiplications and additions.

$$\mathbf{X} \times \mathbf{Y} = \begin{bmatrix} X_{00}Y_{00} + X_{01}Y_{10} & X_{00}Y_{01} + X_{01}Y_{11} \\ X_{10}Y_{00} + X_{11}Y_{10} & X_{10}Y_{00} + X_{11}Y_{10} \end{bmatrix}$$

Inspecting the calculations above, in a 2x2 matrix multiplication, we highlight three major observations.

- Each element of the resultant matrix is independent.
- All multiplication operations are independent.
- The addition operation is accumulation operation.

Here, the computation for rank 2×2 end with an addition operation. However, for rank 3 (> 2), the computation for the first index becomes $[\mathbf{X} \times \mathbf{Y}]_{00} = X_{00}Y_{00} + X_{01}Y_{10} + X_{02}Y_{20}$, where all the multiplication results are accumulated (reduced by addition). We address the incorporation of parallelism in such accumulation in Section V-B2.

The small and limited number of cores is a limitation while distributing such matrix operation for CPU ||. Hence, we only take the first observation into account and employ each core to compute the results for the CPU-based parallel framework.

V. GPU-BASED PARALLEL FRAMEWORK

In this section, we first present three generalized techniques to introduce GPU parallelism (GPU ||) for any FHE computations. Then, we adopt them to implement and optimize the arithmetic operations.

A. Proposed Techniques

1) Parallel TFHE Construction: We depict the boolean circuit computation in Figure 2. Here, each LWE sample comprises of two variables namely $\overrightarrow{A}$ and B, where $\overrightarrow{A}$ is defined as a vector. It is noteworthy that $\overrightarrow{A}$ is a 32-bit integer vector defined by the secret key size (m) which has a lower memory requirement compared to other FHE implementations (Section VIII). In our parallel TFHE construction, we only store the vector $\overrightarrow{A}$ on the GPU's global memory.

In addition to all vector operations inside the GPU, we also employ the native cuda enabled FFT library (cuFFT) which uses the parallel cuda cores for FFT operations. Here, the parallel batching technique from cuFFT supports multiple FFT operations to be executed simultaneously. However, cuFFT also limits such parallel number of batches. It keeps the batches in an asynchronous launch queue, and processes a certain number of batches in parallel. This number of parallel batches solely depends on the hardware capacity and specifications [33].
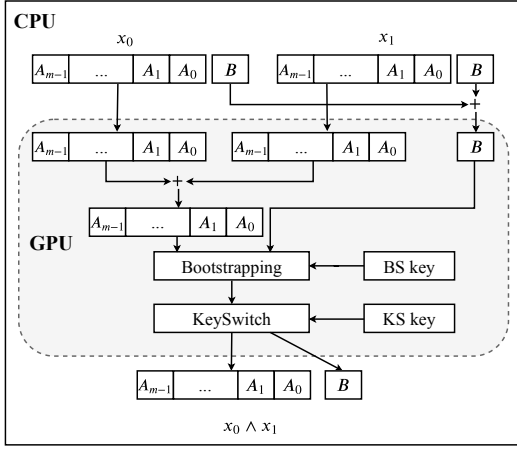
Fig. 2: Arbitrary operation between two bits where BS, KS key represents bootstrapping and key switching keys, respectively
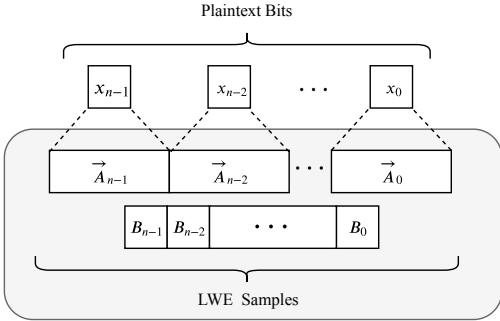

Fig. 3: Coalescing n-LWE samples (ciphertexts) for n-bits

2) Bit Coalescing (BC): Bit Coalescing combines n-LWE samples in a contiguous memory to represent n-encrypted bits. The encryption of a n-bit number, $X \in \mathbb{B}^n$ requires n-LWE samples (ciphertext), and each sample contains a vector of length m. Instead of treating the vectors of ciphertexts separately, we coalesce them altogether (dimension $1 \times mn$) as illustrated in Figure 3.

The intuition behind such construction is to increase parallelism by extending the vector length in a contiguous memory. Coalescing the vectors increases the vector length but we incorporate more threads to maximize parallelization and reduce the execution time.

3) Compound Gate: Since addition is used in most arithmetic circuits, we propose a new gate structure, *Compound Gates* which allows further parallel operations among encrypted bits. These gates are a hybrid of two gates, which takes two 1-bit inputs as an ordinary boolean gate but gives two different outputs. The motivation behind this novel gate structure comes from the addition circuit. For $R = A + B$, we compute r_i and c_i with the following equations:

$$r_i = a_i \oplus b_i \oplus c_{i-1} \quad (1)$$
$$c_i = a_i \wedge b_i \mid (a_i \oplus b_i) \wedge c_{i-1} \quad (2)$$

Here, r_i, a_i, b_i, and c_i denotes i^{th}-bit of R, A, B, and the carry, respectively. Figure 1 illustrates this computation for an n-bit addition.

While computing the equations 1, and 2, we observe that AND ($\wedge$) and XOR ($\oplus$) are computed on the same input bits. As these operations are independent, they can be combined into a single gate, which then can be computed in parallel. We name these gates as *compound gates*. Thus, $a \oplus b$ and $a \wedge b$ from Equation 1 and 2 can be computed as,

$$s, c = \underbrace{a \oplus b, a \wedge b}_{CONCAT}$$

Here, the outputs of $s = a \wedge b$ and $c = a \oplus b$ are concatenated. The compound gates construction is analogous to the task-level parallelism in CPU, where one thread performs $\wedge$, while another thread performs $\oplus$.

In GPU ||, the compound gates operations are flexible as $\wedge$ or $\oplus$ can be replaced with any other logic gates. Furthermore, the structure is extensible up to n-bits input and $2n$-bits output.

B. Algebraic Circuits on GPU

1) Addition: Bitwise Addition (GPU$_1$): From the addition circuit in Section IV-A, we did not find any data-level parallelism. However, we noticed the presence of task-level parallelism for AND and XOR as mentioned in the compound gates construction. Hence, we incorporated the compound gates to construct the bitwise addition circuit. We also implemented the vector addition circuits using GPU$_1$ to support complex circuits such as multiplications (Section V-B2).

Number-wise Addition (GPU$_n$): We consider another addition technique to benefit from bit coalescing. Here, we operate on all n-bits together. For $R = A + B$, we first store A in R ($R = A$). Then we compute, $Carry = R \wedge B$, $R = R \oplus B$, and $B = Carry \ll 1$, for n times.

Here, we utilize compound gates to perform $R \wedge B$ and $R \oplus B$ in parallel. Thus, in each iteration, the input becomes two n-bit numbers, while in bitwise computation the input was two single bits. On the contrary, even after using compound gates, the bitwise addition (Equations 1 and 2) has more sequential blocks (3) than the number-wise addition (0). We analyze both in Section VI-C.

2) Multiplication: Naive Approach: According to Section III-B1, multiplications have $\wedge$ and $\ll$ operations which can be executed in parallel. It will result in n-numbers where each number will have $[n, 2n]$-bits due to the $\ll$. We need to accumulate these uneven sized numbers which cannot be distributed among the GPU threads. Furthermore, the addition presents another sequential bottleneck while adding and storing (+ =) the results in the same memory location. Therefore, this serial addition will increase the execution time. In the framework, we optimize the operation by introducing a tree-based approach.

In this approach, we divide n-numbers (LWE vectors) into two $n/2$ vectors. This two $n/2$ vectors are added in parallel. We repeat the process as we divide the resultant vectors int two $n/4$ vectors and add them in parallel. The process continues
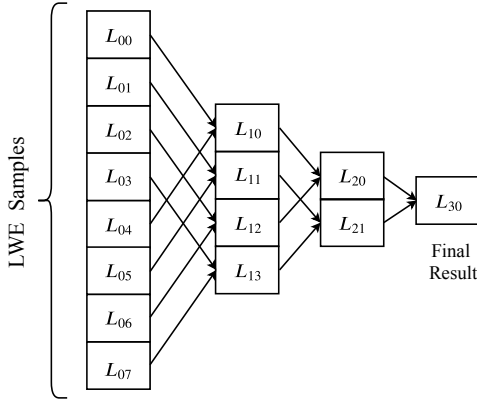
Fig. 4: Accumulating $n = 8$ LWE samples (L_{ij}) in parallel using a tree-based reduction

until we get the final result. Notably, the tree-based approach requires $\log n$ steps for the accumulation. In Figure 4 for $n = 8$, all the ciphertexts underwent $\wedge$ and $\ll$ in parallel, and waited for addition. Here, L_{ij} represents the LWE samples (encrypted numbers), i is the level, and j denotes the position.

Likewise vector additions, we integrated vector multiplications in similar fashion on our framework. Interestingly, we used both vector additions and multiplications in Karatsuba's algorithm which we describe next.

Karatsuba Multiplication: We used Karatsuba's algorithm with some modifications in our framework to achieve further efficiency while performing multiplications. However, this algorithm requires both addition and multiplication vector operations which tested the efficacy of these components as well. We modified the original Algorithm 1 to introduce the vector operations and rewrite the computations in Line 8-11 as:

$$\langle Temp_0, \ Temp_1 \rangle = \langle X_0, \ X_1 \rangle + \langle Y_0, \ Y_1 \rangle$$
$$\langle Z_0, \ Z_1, \ Z_2 \rangle = \langle X_0, \ X_1, \ Temp_0 \rangle \cdot \langle Y_0, \ Y_1, \ Temp_1 \rangle$$
$$\langle Temp_0, \ Temp_1 \rangle = \langle Z_2, \ Z_1 \rangle + \langle 1, \ Z_0 \rangle$$
$$Z_2 = Temp_0 + (Temp_1)'$$

In the above equations, X_0, X_1, Y_0, Y_1, Z_0, Z_1, and Z_2 are taken from the algorithm. $\langle \ldots \rangle$ and $\cdot$ are used to denote concatenated vectors and dot product, respectively. For example, in the first equation, $Temp_0$ and $Temp_1$ store the addition of X_0, Y_0 and X_1, Y_1. It is noteworthy that in the CPU ‖ framework, we utilized task-level parallelism to perform these vector operations as described in Section IV.

3) Vector and Matrix Operations:

Addition: The vector addition is a pointwise addition of the elements at their respective position. The underlying addition operation incorporates bitwise addition. Since the operation propagates bit by bit, we combine the bit from the numbers (to be added) and compute them in parallel. For example, in a vector addition of length ℓ, we combine all the bits for the required bit position and compute the result.

Matrix addition also performs pointwise additions between the matrix elements. Hence, matrices represented in a row-major vector format corresponds to a vector addition operation. Therefore, we simply convert the matrices into row-major and add them utilizing the parallel vector addition.

Multiplication: Analogous to additions, vector multiplications are also a pointwise operation. Here, we compute the AND and `left shift` operations in parallel and accumulate the values in a tree-based approach as described in regular multiplications (Section V-B2).

Unlike addition, matrix multiplication is more complicated as it requires more computations. Section IV-D presents a schematic computation for a 2×2 matrix along with the possible parallel computations. Notably, for two n-ranked *squared* (for brevity) matrix multiplication, we require n^3 multiplication operations. Here, we separate all the multipliers and multiplicands into two vectors and perform parallel vector operations. For example, in a square matrix of rank 16, each vector length for multiplication will be 4096. Furthermore, for each 16-bit vector components (matrix elements), the computation raises to $4096 \times 16 \times 16$-bits computation for parallel multiplication.

Therefore, multiplying these vectors require larger GPU memory which we wanted to avoid. It also required a large number of threads for computing and storing the LWE samples as well. Although, the threads can be reused sequentially, this problem is more severe due to the fixed GPU memory constraint. Hence, matrix multiplications on larger dimensions can essentially run out of GPU memory.

Therefore, we consider a different technique named Cannon's Algorithm [34]. The algorithm consists of cycles containing multiplication, addition, and shifting the matrix elements. Each cycle also depends on the values generated by the previous cycle, incurring a sequential bottleneck. Nevertheless, the algorithm provides a much needed scalability for large-scale multiplication instead of using a considerable amount of GPU memory which is often not present.

In each cycle, we first perform the multiplication using the vector operations. Then, we add on the multiplied data using the parallel vector additions. Lastly, we shift the elements' positions for the next round.

VI. EXPERIMENTAL ANALYSIS

The experimental environment included an Intel(R) Core™ i7-2600 CPU having 16 GB system memory with a NVIDIA GeForce GTX 1080 GPU with 8 GB memory [33]. The CPU and GPU contained 8 and $40,960$ hardware threads, respectively. We used the same setup to analyze all three frameworks: sequential, CPU ‖, GPU ‖.

We use two metrics for the comparison: a) execution time and b) speedup $= \frac{T_{seq}}{T_{par}}$. Here, T_{seq} and T_{par} are the time for computing the sequential and the parallel algorithm. In the following sections, we gradually analyze the complicated arithmetic circuits using the best results from the foregoing analysis.
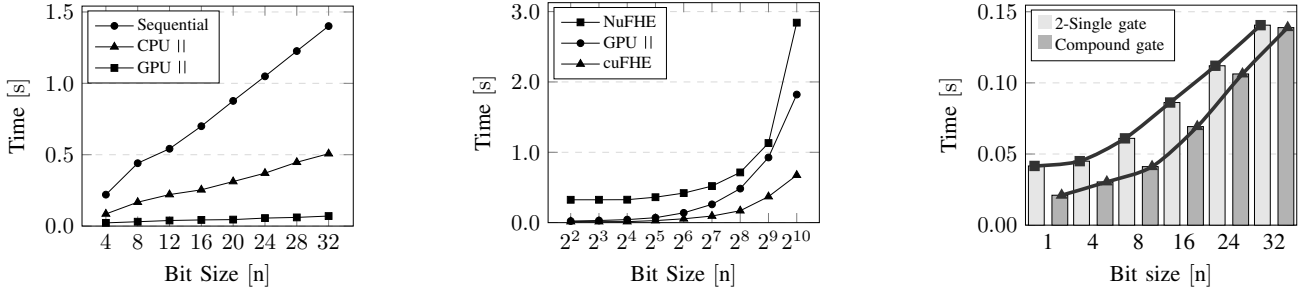
(a) Comparison of Seq., CPU ||, and GPU || (b) Analysis with GPU-assisted frameworks   (c) Compound gate against 2-single gates

Fig. 5: Performance analysis of GPU-accelerated TFHE with the sequential and CPU || frameworks (5a), and comparison with the existing GPU-assisted libraries (5b). Figure 5c presents the performance of compound gates against 2-single gate operations

TABLE IV: Computation time (ms) for Bootstrapping, Key Switching and Misc. for sequential and GPU framework

Bit Size	Sequential					GPU			
n	Bootstrapping	Key Switch	Misc.	Total		Bootstrapping	Key Switch	Misc.	Total
2	68.89	17.13	27.04	113.05		19.64	2.65	0.45	22.74
4	138.02	34.18	47.97	220.17		18.86	2.69	0.08	21.63
8	275.67	68.31	96.48	440.46		27.83	2.69	0.06	30.58
16	137.25	137.25	425.22	699.72		40.70	2.91	0.44	44.06
32	274.3	274.30	852.51	1401.10		66.74	3.34	0.42	70.50

A. GPU-accelerated TFHE

Initially, we discuss our performance over boolean gate operations, which is deemed as a building blocks of any computation. Figure 5a depicts the execution time difference among the sequential, CPU || and GPU || framework for $[4, 32]$-bits. The sequential AND operation takes a minimum of 0.22s (4-bit) while the runtime increases to 1.4s for 32-bits.

In the GPU || framework, bit coalescing facilitates storing LWE samples in contiguous memory and takes advantage of available vector operations. Thus, it helps to reduce the execution time from $0.22 - 1.4s$ to $0.02 - 0.06s$ for 4 to 32-bits. Here, for 32-bits, our techniques provide a $20\times$ speedup. Similar improvement is foreseen in the CPU || framework as we divide the number of bits by the available threads. However, the execution time increases for CPU framework since there is only a limited number of available threads. This limited number of threads is one of the primary motivations behind utilizing GPU.

Then, we further scrutinize the execution time by dividing gate operations into three major components—*a)* Bootstrapping, *b)* Key Switching, and *c)* Miscellaneous. We selected the first two as they are the most time-consuming operations and fairly generalizable to other HE schemes. Table IV shows the difference in execution time between the sequential and the GPU || for $\{2, \ldots, 32\}$-bits. We show that the execution time increment is less compared to the sequential approach.

We further investigated the bootstrapping performance in GPU || framework for the boolean gate operations. Our cuda enabled FFT library takes the LWE samples in batches and performs the FFT in parallel. However, due to the h/w limitations, the number of batches to be executed in parallel is limited. It can only operate on a certain number of batches at once and next batches are kept in a queue. Hence, a sequential overhead occurs for a large number of batches that can increase the execution time.

Under the same h/w setting, we benchmark our proposed framework with the existing GPU-based libraries (cuFHE and NuFHE). Although our GPU || framework outperforms NuFHE for different bit sizes (Figure 5b), the performance degrades for larger bit sizes w.r.t. cuFHE. As the cuFHE implementation focuses more on the gate level optimization, we focus on the arithmetic circuit computations. In Section VI-C, we analyze our arithmetic circuits where our framework outperforms the existing GPU libraries.

B. Compound Gate Analysis

According to Section V-A3, the compound gates are used to improve the execution time for additions or multiplications. Since, the existing frameworks do not provide these optimizations, we benchmark the compound gates with the proposed single gate computations. Figure 5c illustrates the performance of one compound gate over 2-single gates computed sequentially. We performed several iterations for different number of bits $(1, \ldots, 32)$ as shown on the X-axis while the Y-axis represents the execution time. Notably, a 32-bit compound gates will have two 32-bit inputs and output two 32-bits.

Here, bit coalescing improves the execution time as it takes only 0.02s for one compound gates evaluation, compared to 0.04s on performing 2-single gates sequentially. However, Figure 5c shows an interesting trend in the execution time between 2-single gates and one compound gates evaluation. The gap favoring the compound ones tends to get narrower for higher number of bits. For example, the speedup for 1-bit happens to be $0.04/0.02 = 2$ times whereas it reduces to 1.01

TABLE V: Execution time (sec) for the n-bit addition

Frameworks	16-bit	24-bit	32-bit
Sequential	3.51	5.23	7.04
cuFHE [15]	1.00	1.51	2.03
NuFHE [16]	2.92	3.56	4.16
Cingulata [17]	1.10	1.63	2.16
Our Methods			
CPU ||	3.51	5.23	7.04
GPU$_n$||	0.94	2.55	4.44
GPU$_1$||	0.98	1.47	1.99

TABLE VI: Execution time (sec) for vector addition

Length	16-bit				32-bit		
ℓ	Seq.	CPU ||	GPU ||		Seq.	CPU ||	GPU ||
4	13.98	5.07	1.27		28.05	10.02	2.56
8	27.86	9.96	1.78		56.01	19.29	3.58
16	55.66	19.65	2.82		111.3	38.77	5.70
32	111.32	38.99	5.41		224.31	77.18	11.22

TABLE VII: Multiplication execution time (sec) comparison

Frameworks	16-bit	24-bit	32-bit
Naive			
Sequential	120.64	273.82	489.94
CPU ||	52.77	101.22	174.54
GPU ||	11.16	22.08	33.99
cuFHE [15]	32.75	74.21	132.23
NuFHE [16]	47.72	105.48	186.00
Cingulata [17]	11.50	27.04	50.69
Karatsuba			
CPU	54.76	-	177.04
GPU ||	7.6708	-	24.62

TABLE VIII: Execution time (min) for vector multiplication

Length	16-bit				32-bit		
ℓ	Seq.	CPU ||	GPU ||		Seq.	CPU ||	GPU ||
4	8.13	3.25	0.41		32.56	12.15	1.61
8	16.29	6.17	0.75		65.12	23.48	2.96
16	32.62	11.93	1.40		130.31	46.39	5.62
32	65.15	23.58	2.68		260.52	92.44	10.79

for 32-bits. The reason behind this diminishing performance is the *asynchronous launch queue* of GPUs.

As mentioned in Section V-A1, we use batch execution for the FFT operations. Hence, the number of parallel batches depends on the asynchronous launch queue size of the underlying GPU which can delay the FFT operations for a large number batches. This ultimately adversely affects the speedup for large LWE sample vectors. Nevertheless, the analysis shows that the 1-bit compound gates is the most efficient, and we employ it in the following arithmetic operations.

C. Addition

Table V presents a comparative analysis of the addition operation for 16, 24, 32-bit encrypted numbers. We consider our proposed frameworks: sequential, CPU \|\|, and GPU \|\|, and benchmark them with cuFHE [15], NuFHE [16] and Cingulata [17]. Furthermore, we present the performance of two variants of addition operation: GPU$_n$\|\| (number-wise) and GPU$_1$\|\| (bitwise) as discussed in Section V-B1.

Table V demonstrates that GPU$_n$\|\| performs better than the sequential and CPU \|\| circuits. The GPU$_n$ provides a $3.72\times$ speedup for 16-bits whereas $1.58\times$ for 32-bit. However, GPU$_n$\|\| performs better only for 16-bit additions compared to GPU$_1$\|\|. For 24 and 32-bit additions, GPU$_1$\|\| performs around $2\times$ better than GPU$_n$\|\|. This improvement in essential as it reveals the algorithm to choose between GPU$_1$\|\| and GPU$_n$\|\|.

Although, both addition operations (GPU$_n$\|\| and GPU$_1$\|\|) utilize compound gates, they differ in the number of input bits (n and 1 for GPU$_n$\|\| and GPU$_1$\|\|, respectively). Since the compound gates performs better for smaller bits (Section VI-B), the bitwise addition performs better than the number-wise addition for 24/32-bit operations. Hence, we utilize bitwise addition for building other circuits.

NuFHE and cuFHE do not provide any arithmetic circuits in their library. Therefore, we implemented such circuits on their library and performed the same experiments. Additionally, we considered Cingulata [17] (a compiler toolchain for TFHE) and compared the execution time. Table V summarizes all

the results, where we found our proposed addition circuit (GPU$_1$\|\|) outperforms the other approaches.

We further experimented on the vector additions adopting the bitwise addition and showed the analysis in Table VI. Like addition, the performance improvement on the vector addition is also noticeable. The framework scales by taking similar execution time for smaller vector lengths $\ell \leq 8$. However, the execution time increases for longer vectors as they involve more parallel bit computations, and consequently, increase the batch size of FFT operations. The difference is clearer on 32-bit vector additions with $\ell = 32$ which takes almost twice the time of $\ell = 16$. However, for $\ell \leq 8$, the executions times are almost similar due to the parallel computations. In Section VI-B we have discussed this issue which relies on the FFT batch size. Notably, Figure 5c also aligns with this evidence as the larger batch size for FFT on GPUs affects the speedup. For example, $\ell = 32$ will require more FFT batches compared to $\ell = 16$ which requires more time to finish the addition operation. We did not include other frameworks in Table VI, since our GPU \|\| performed better comparing to the others in Table V.

D. Multiplication

The multiplication operation uses a sequential accumulation (reduce by addition) operation. Instead, we use a tree-based vector addition approach (discussed in Section V-B2) and gain a significant speedup. Table VII portrays the execution times for the multiplication operations using the frameworks. Here, we employed all available threads on the machine. Like the addition circuit performance, here GPU \|\| outperforms the sequential circuits and CPU \|\| operations by a factor of ≈ 11 and ≈ 14.5, respectively for 32-bit multiplication.

We further implemented the multiplication circuit on cuFHE and NuFHE. Table VII summarizes the results comparing our proposed framework with cuFHE, NuFHE, and Cingulata. Our GPU \|\| framework is faster in execution time than the other techniques. Notably, the performance improvement is scalable

TABLE IX: Matrix multiplication execution time (min)

Dimension	Sequential	CPU ‖	GPU ‖
2×2	17.07	10.62	0.86
4×4	136.68	47.78	5.90
8×8	1,090.12	351.82	43.95
16×16	8,717.89	2,514.34	186.23

TABLE X: Execution time (min) for Linear Regression

Datasets	#Rows	#Attributes	Data Type	Time
Dataset 1	200	10	Numerical	163.38
			Binary	53.91
Dataset 2		20	Numerical	268.86
			Binary	67.88
Dataset 3	300	10	Numerical	245.38
			Binary	80.91
Dataset 4		20	Numerical	403.85
			Binary	95.88

with the increasing number of bits. This is due to tree-based additions following the reduction operations and computing all boolean gate operations by coalescing the bits altogether.

Besides, we also analyze vector multiplications available in our framework and present a comparison among the frameworks in Table VIII. We found out an increase in execution time for a certain length (e.g., $\ell = 32$ on 16-bit or $\ell = 4$ on 32-bit), which is similar to the issue in vector addition (Section VI-C). Hence, the vector operations from $\ell \leq 16$ can be sequentially added to compute arbitrary vector operations. For example, we can use two $\ell = 16$ vector multiplication to compute $\ell = 32$ multiplication resulting around 11 mins. In the vector analysis, we did not add the computations over the other frameworks since our framework surpassed their achievements for a single multiplications.

E. Karatsuba Multiplication

In Table VII, we provide execution time for 16 and 24-bit Karatsuba multiplication over encrypted numbers as well. In the CPU ‖ construction of the algorithm, the execution time does not improvement, rather it increases slightly. We observed that for both 16 and 32-bit multiplication, Karatsuba outperforms naive GPU‖ multiplication algorithm on GPU by 1.50 times. Karatsuba multiplication can also be considered a complex arithmetic operation as it comprises of both addition, multiplication, and vector operations. However, the CPU ‖ framework did not provide such difference in performance as it took more time for the `fork-and-join` threads required by the divide and conquer algorithm.

F. Matrix Operations

From Section V-B3, it is evident that the vector addition represents matrix additions as both operations are done point-wise. Therefore, Table VI can be extended to represent the execution time for the matrix additions, where ℓ becomes the number of elements of the matrices. Table IX enlists the matrix multiplication execution time for different dimensions using Cannon's algorithm [34]. For a 16×16 matrix, GPU ‖ achieves a ≈ 48 and ≈ 15 times speedup compared to the sequential and CPU ‖ approach, respectively.

G. Application: Linear Regression

We employed the arithmetic operations (vector/matrix addition, multiplication) to compute linear regression models as an application to test the efficacy of the framework. We produced four synthetic dataset with different number of instances (rows) and attributes (columns), and tabulated the execution times in Table X. Each datasets had two variants: binary and numeric values. As the multiplications are essentially AND operations for binary values, it takes less time compared to the numeric dataset.

VII. DISCUSSION

In this section, we provide answers to the following questions about our proposed framework:

Is the proposed framework sufficient to implement any computations? In this article, we show how to implement boolean gates properly using GPUs to gain performance improvement. We then show how to compute addition, multiplication, and matrix operations using the proposed framework. Implementing more complex algorithms such as *secure machine learning* [35, 36] are beyond the scope of this paper. In future work, we will investigate how to further optimize the framework for machine learning algorithms. Note that we have implemented a FHE scheme. Hence, any computable function can be implemented using our framework.

For GPU ‖ framework, how do we compute on encrypted data larger than the fixed GPU memory? The fixed GPU memories and their variations in access speeds are limitations for any GPU ‖ application. Similar problems also occur in deep learning while handling larger datasets. The solution includes batching the data or using multiple GPUs. Our proposed framework can also avail such solutions as it can easily be extended to accommodate larger ciphertexts.

How can we achieve further speedup on both frameworks? On the CPU ‖ framework, we have attempted most H/W or S/W level optimizations to the best of our knowledge. However, our GPU ‖ framework partially relied on the global GPU memory, which is slower than its counterparts. This is critical as different device memories offer variant read/write speeds. Notably, shared memory (L1) is the fastest memory after register. Our implementation uses a combination of shared and global memory due to the ciphertext size. In the future, we would like to utilize only the shared memory, which is much smaller but should provide better speedup compared to the current approach.

How the bit security level would affect the reported speedup? The current framework is analogous to the existing implementation of TFHE [37] providing 110-bit security which might not be sufficient for some applications. However, our GPU ‖ framework can accommodate any change for the desired bit security level. Nevertheless, such change will change the execution times as well. For example, any less security level than 110-bits will result in faster execution and

likewise for a higher bit security. We will include and analyze the speedup for the dynamic bit security levels in future.

VIII. Related Works

In this section, we discuss the other HE schemes from Table I and categorize schemes based on their number representation: *a)* bit-wise, *b)* modular and *c)* approximate.

Bitwise Encryption usually takes the bit representation of any number and encrypts accordingly. The computations are also done bit-wise as each bit can be considered independent from another. This bit-wise representation is crucial for our parallel framework as it offers less dependency between bits which we can operate in parallel. Furthermore, it provides faster bootstrapping and smaller ciphertext size, which can be easily tailored for the fixed memory GPUs. This concept is formalized and named as GSW [38] around 2013, and it was later improved in subsequent works [23, 1, 25].

Modular Encryption schemes utilize a fixed modulus q which denotes the size of the ciphertexts. There have been many developments [39, 40] in this direction as they offer a reasonable execution time (Table I). The addition and multiplication times from FV [24] and SEAL [14] show the difference as they are much faster compared to our GPU-based framework.

However, these schemes do a trade-off between the bootstrapping and the efficiency as they are often designated as somewhat homomorphic encryption. Here, in most cases, the number of computations or the level of multiplications are predefined as there is no procedure for noise reduction Decryption is performed after the desired computation. Furthermore, the encrypted data evidently suffers from larger ciphertexts as the value of q is picked from large numbers.

For example, we selected the ciphertext modulus of 250 and 881 bits for FV-NFLlib [24] and SEAL [14], respectively. The polynomial degrees (d) were chosen 13 and 15 for the two frameworks as it was required to comply with the targeted bit security to populate Table I. It is noteworthy that smaller q and d will result in faster runtime and smaller ciphertexts, but they will limit the number of computations as well. Therefore, this modular representation requires to fix the number of homomorphic operations limiting the use cases.

Approximate Number representations are recently proposed by Cheon *et al.* (CKKS [41]) in 2017. These schemes also provide efficient Single Instruction Multiple Data (SIMD) [42] operations similar to the modular representations as mentioned above. However, they have an inexact but efficient bootstrapping mechanism which can be applied in less precision-demanding applications. The cryptosystem also incurs larger ciphertexts (7MB) similar to the modular approach as we tested it for $q = 1050$ and $d = 15$. Here, we did not discuss HELib [43], the first cornerstone of all HE implementations since its cryptosystem BGV [40] is enhanced and utilized by the other modular HE schemes (such as SEAL [14]).

The goal of this work is to parallelize an FHE scheme. Most HE schemes that follow modular encryption are either somewhat or adopt inexact bootstrapping. Besides, their expansion after encryption requires more memory. Hence, we choose the bitwise and bootstrappable encryption scheme: TFHE.

Hardware Solutions are less studied and employed to increase the efficiency of FHE computations. Since the formulation of FHE [2] with ideal lattices, most of the efficiency improvements are considered from the standpoint of asymptotic runtimes. A few approaches considered the incorporation of existing multiprocessors (e.g., GPU) or FPGAs [44] to achieve faster homomorphic operation. Dai and Sunar ported another scheme LTV [45] to GPU-based implementation [46, 47]. LTV is a variant of HE that performs a limited number of operations on a given ciphertext.

Lei *et al.* ported FHEW-V2 [23] to GPU [48] and extended the boolean implementation to 30-bit addition and 6-bit multiplication with a speed up ≈ 2.5. Since TFHE extends FHEW and performs better than it predecessor, we consider TFHE as our baseline framework.

In 2015, a GPU based HE scheme CuHE [46] was proposed. However, it was not fully homomorphic as it did not have bootstrapping, hence we do not include it in our analyses. Later in 2018, two GPU FHE libraries cuFHE [15] and NuFHE [16] were released. Both the libraries focused on optimizing of the boolean gate operations. Recently, Yang *et al.* [49] benchmarked cuFHE and its predecessor TFHE, and analyzed the speedup which we also discuss in our paper (Table I).

Our experimental analysis shows that only performing the boolean gates in parallel is not sufficient to reduce the execution time of higher level circuit (i.e., multiplication). Hence, besides employing GPU for homomorphic gate operations, we focus on arithmetic circuit. For example, we are 3.9 times faster than cuFHE in 32-bit multiplications.

Recently, Zhou *et al.* improved TFHE by reducing and performing the serial operations of bootstrapping in parallel [50]. However, they did use any hardware acceleration to the existing FHE operations. We consider this work as an essential future direction that can be integrated to our framework for better executing times.

Acknowledgments

IX. Conclusion

In this paper, we constructed the algebraic circuits for FHE, which can be utilized by arbitrary complex operations. Furthermore, we explored the CPU-level parallelism for improving the execution time of the underlying FHE computations. Our notable contribution is the proposed GPU-level parallel framework that utilizes novel optimizations such as bit coalescing, compound gate, and tree-based vector accumulation. Experimental results show that the proposed method is $20\times$ and $14.5\times$ faster than the existing technique for computing boolean gates and multiplications respectively (Table II).

REFERENCES

[1] I. Chillotti, N. Gama, M. Georgieva, and M. Izabachene, "Faster fully homomorphic encryption: Bootstrapping in less than 0.1 seconds," in *International Conference on the Theory and Application of Cryptology and Information Security*. Springer, 2016, pp. 3–33.

[2] C. Gentry *et al.*, "Fully homomorphic encryption using ideal lattices." in *Stoc*, vol. 9, no. 2009, 2009, pp. 169–178.

[3] A. Pham, I. Dacosta, G. Endignoux, J. R. T. Pastoriza, K. Huguenin, and J.-P. Hubaux, "Oride: A privacy-preserving yet accountable ride-hailing service," in *26th USENIX Security Symposium (USENIX Security 17)*, Vancouver, BC, 2017, pp. 1235–1252.

[4] M. Kim, Y. Song, and J. H. Cheon, "Secure searching of biomarkers through hybrid homomorphic encryption scheme," *BMC medical genomics*, vol. 10, no. 2, p. 42, 2017.

[5] H. Chen, R. Gilad-Bachrach, K. Han, Z. Huang, A. Jalali, K. Laine, and K. Lauter, "Logistic regression over encrypted data from fully homomorphic encryption," *BMC medical genomics*, vol. 11, no. 4, p. 81, 2018.

[6] Google Prediction API. Https://cloud.google.com/prediction/, Accessed on 21 May, 2019. [Online]. Available: https://cloud.google.com/prediction/

[7] Microsoft Azure Machine Learning. Https://azure.microsoft.com/en-us/services/machine-learning-studio/, Accessed 21 May, 2019. [Online]. Available: https://azure.microsoft.com/en-us/services/machine-learning-studio/

[8] M. N. Sadat, M. M. Al Aziz, N. Mohammed, S. Pakhomov, H. Liu, and X. Jiang, "A privacy-preserving distributed filtering framework for nlp artifacts," *BMC medical informatics and decision making*, vol. 19, no. 1, pp. 1–10, 2019.

[9] "Uber employees' spied on ex-partners, politicians and Beyonce."

[10] R. L. Rivest, A. Shamir, and L. Adleman, "A method for obtaining digital signatures and public-key cryptosystems," *Communications of the ACM*, vol. 21, no. 2, pp. 120–126, 1978.

[11] P. Paillier, "Public-key cryptosystems based on composite degree residuosity classes," in *International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 1999, pp. 223–238.

[12] C. Lomont, "Introduction to intel advanced vector extensions," *Intel White Paper*, pp. 1–21, 2011.

[13] J. H. Cheon, K. Han, A. Kim, M. Kim, and Y. Song, "Bootstrapping for approximate homomorphic encryption," in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 2018, pp. 360–384.

[14] "Microsoft SEAL (release 3.2)," https://github.com/Microsoft/SEAL, Feb. 2019, microsoft Research, Redmond, WA.

[15] "CUDA-accelerated Fully Homomorphic Encryption Library," Sep. 2019. [Online]. Available: https://github.com/vernamlab/cuFHE

[16] "NuFHE, a GPU-powered Torus FHE implementation," Sep. 2019. [Online]. Available: https://github.com/nucypher/nufhe

[17] "Cingulata," Sep. 2019. [Online]. Available: https://github.com/CEA-LIST/Cingulata

[18] O. Regev, "On lattices, learning with errors, random linear codes, and cryptography," *Journal of the ACM (JACM)*, vol. 56, no. 6, p. 34, 2009.

[19] M. N. Sadat, A. Aziz, M. Momin, N. Mohammed, F. Chen, X. Jiang, and S. Wang, "Safety: secure gwas in federated environment through a hybrid solution," *IEEE/ACM Transactions on Computational Biology and Bioinformatics (TCBB)*, vol. 16, no. 1, pp. 93–102, 2019.

[20] C. Juvekar, V. Vaikuntanathan, and A. Chandrakasan, "GAZELLE: A low latency framework for secure neural network inference," in *27th USENIX Security Symposium (USENIX Security 18)*, 2018, pp. 1651–1669.

[21] E. Hesamifard, H. Takabi, M. Ghasemi, and R. N. Wright, "Privacy-preserving machine learning as a service," *Proceedings on Privacy Enhancing Technologies*, vol. 2018, no. 3, pp. 123–142, 2018.

[22] Z. Brakerski, C. Gentry, and S. Halevi, "Packed ciphertexts in lwe-based homomorphic encryption," in *International Workshop on Public Key Cryptography*. Springer, 2013, pp. 1–13.

[23] L. Ducas and D. Micciancio, "Fhew: Bootstrapping homomorphic encryption in less than a second," Cryptology ePrint Archive, Report 2014/816, 2014, https://eprint.iacr.org/2014/816.

[24] J. Fan and F. Vercauteren, "Somewhat practical fully homomorphic encryption." *IACR Cryptology ePrint Archive*, vol. 2012, p. 144, 2012.

[25] I. Chillotti, N. Gama, M. Georgieva, and M. Izabachène, "Faster packed homomorphic operations and efficient circuit bootstrapping for tfhe," in *International Conference on the Theory and Application of Cryptology and Information Security*. Springer, 2017, pp. 377–408.

[26] M. N. Sadat, X. Jiang, M. M. Al Aziz, S. Wang, and N. Mohammed, "Secure and efficient regression analysis using a hybrid cryptographic framework: Development and evaluation," *JMIR medical informatics*, vol. 6, no. 1, p. e14, 2018.

[27] C. Boura, N. Gama, and M. Georgieva, "Chimera: a unified framework for b/fv, tfhe and heaan fully homomorphic encryption and predictions for deep learning," Cryptology ePrint Archive, Report 2018/758, Tech. Rep., 2018.

[28] M. Frigo and S. G. Johnson, "Fftw: An adaptive software architecture for the fft," in *Proceedings of the 1998 IEEE International Conference on Acoustics, Speech and Signal Processing, ICASSP'98 (Cat. No. 98CH36181)*, vol. 3. IEEE, 1998, pp. 1381–1384.

[29] A. Salem, P. Berrang, M. Humbert, and M. Backes, "Privacy-preserving similar patient queries for combined biomedical data," *Proceedings on Privacy Enhancing Technologies*, vol. 2019, no. 1, pp. 47–67, 2019.

[30] C. C. McGeoch, "Parallel addition," *The American Mathematical Monthly*, vol. 100, no. 9, pp. 867–871, 1993. [Online]. Available: http://www.jstor.org/stable/2324666

[31] A. A. Karatsuba and Y. P. Ofman, "Multiplication of many-digital numbers by automatic computers," in *Doklady Akademii Nauk*, vol. 145, no. 2. Russian Academy of Sciences, 1962, pp. 293–294.

[32] R. Chandra, L. Dagum, D. Kohr, R. Menon, D. Maydan, and J. McDonald, *Parallel programming in OpenMP*. Morgan kaufmann, 2001.

[33] NVIDIA, "Geforce gtx 1080 graphics cards from nvidia geforce." [Online]. Available: https://www.nvidia.com/en-us/geforce/products/10series/geforce-gtx-1080/

[34] L. E. Cannon, "A cellular computer to implement the kalman filter algorithm," Ph.D. dissertation, Montana State University-Bozeman, College of Engineering, 1969.

[35] P. Xie, M. Bilenko, T. Finley, R. Gilad-Bachrach, K. Lauter, and M. Naehrig, "Crypto-nets: Neural networks over encrypted data," *arXiv preprint arXiv:1412.6181*, 2014.

[36] H. Takabi, E. Hesamifard, and M. Ghasemi, "Privacy preserving multi-party machine learning with homomorphic encryption," in *29th Annual Conference on Neural Information Processing Systems (NIPS)*, 2016.

[37] I. C. S. Nicolas Gama, SC, "Tfhe: Fast fully homomorphic encryption library over the torus," https://github.com/tfhe/tfhe/tree/v1.0.1, 2017.

[38] C. Gentry, A. Sahai, and B. Waters, "Homomorphic encryption from learning with errors: Conceptually-simpler, asymptotically-faster, attribute-based," in *Annual Cryptology Conference*. Springer, 2013, pp. 75–92.

[39] Z. Brakerski and V. Vaikuntanathan, "Efficient fully homomorphic encryption from (standard) lwe," *SIAM Journal on Computing*, vol. 43, no. 2, pp. 831–871, 2014.

[40] Z. Brakerski, C. Gentry, and V. Vaikuntanathan, "(leveled) fully homomorphic encryption without boot-strapping," *ACM Transactions on Computation Theory (TOCT)*, vol. 6, no. 3, p. 13, 2014.

[41] 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.

[42] M. J. Flynn, "Some computer organizations and their effectiveness," *IEEE transactions on computers*, vol. 100, no. 9, pp. 948–960, 1972.

[43] S. Halevi and V. Shoup, "Algorithms in helib," in *Annual Cryptology Conference*. Springer, 2014, pp. 554–571.

[44] Y. Doröz, E. Öztürk, E. Savaş, and B. Sunar, "Accelerating ltv based homomorphic encryption in reconfigurable hardware," in *International Workshop on Cryptographic Hardware and Embedded Systems*. Springer, 2015, pp. 185–204.

[45] A. López-Alt, E. Tromer, and V. Vaikuntanathan, "On-the-fly multiparty computation on the cloud via multikey fully homomorphic encryption," in *Proceedings of the forty-fourth annual ACM symposium on Theory of computing*. ACM, 2012, pp. 1219–1234.

[46] W. Dai and B. Sunar, "cuhe: A homomorphic encryption accelerator library," in *International Conference on Cryptography and Information Security in the Balkans*. Springer, 2015, pp. 169–186.

[47] W. Dai, Y. Doröz, and B. Sunar, "Accelerating swhe based pirs using gpus," in *International Conference on Financial Cryptography and Data Security*. Springer, 2015, pp. 160–171.

[48] X. Lei, R. Guo, F. Zhang, L. Wang, R. Xu, and G. Qu, "Accelerating homomorphic full adder based on fhew using multicore cpu and gpus," *IEEE Transactions on Industrial Informatics*, 2019.

[49] H.-b. Yang, W.-j. Yao, W.-c. Liu, and B. Wei, "Efficiency analysis of tfhe fully homomorphic encryption software library based on gpu," in *Workshops of the International Conference on Advanced Information Networking and Applications*. Springer, 2019, pp. 93–102.

[50] T. Zhou, X. Yang, L. Liu, W. Zhang, and N. Li, "Faster bootstrapping with multiple addends," *IEEE Access*, vol. 6, pp. 49 868–49 876, 2018.