CPU and GPU Accelerated Fully Homomorphic Encryption

by

Md Toufique Morshed Tamal

A thesis submitted to
The Faculty of Graduate Studies of
The University of Manitoba
in partial fulfillment of the requirements
of the degree of

Master of Science

Department of Computer Science
The University of Manitoba
Winnipeg, Manitoba, Canada
November 2019

Thesis advisor

Dr. Noman Mohammed

Author

Md Toufique Morshed Tamal

CPU and GPU Accelerated Fully Homomorphic Encryption

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 thesis, 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× for any 32-bit boolean operation and 14.5× for multiplications.

Contents

List of Figures

List of Tables

Acknowledgments

Foremost, I would like to express my gratitude to the mightiest God for granting me the opportunity to write this thesis.

I would like to thank my supervisor, Dr. Noman Mohammed, for giving me the opportunity to work under his supervision. I am very grateful for his guidance, suggestions, and feedback throughout the preparation of this thesis. I am thankful to the other members of my committee, Dr. Ruppa Thulasiram, and Dr. Robert McLeod, for the comment and advice they provided.

I thank my mentor Md Momin Al Aziz, and fellow labmates in Data Security and Privacy Group: Nazmus Sadat, Kazi Wasif Ahmed, Zahidul Hasan, Tanbir Ahmed, Safiur Rahman Mahdi, and Tasnia Faequa for their valuable consults and time. I am equally grateful to all the benevolent faculties of Computer Science, University of Manitoba.

Last but not the least, I would like to thank my family: Morsheda Begum and A T M Rafiqul Islam, for giving birth to me at the first place and supporting me throughout my life in every way, and Sadia Mehnaz for being with me throughout the journey.

This thesis is dedicated to Morsheda Begum and A T M Rafiqul Islam.

Chapter 1

Introduction

Fully Homomorphic Encryption (FHE) [1, 2, 3] have 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 [4, 5, 6] 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. The advent and proliferation of machine learning techniques and their applications in recent years have been remarkable. 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 [7], Microsoft Azure Machine Learning [8], GraphLab [9] etc.) 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 and thus susceptible to privacy attacks by the employee at the service provider [10]. 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.

1.1 Current Techniques

Based on the computational power, the homomorphic encryption schemes can be divided into three major categories: Partially, Somewhat, and Fully Homomorphic Encryption schemes. Partially Homomorphic schemes only support one type of operation (e.g., addition or multiplication) for any number of time. For example, RSA [11] is a multiplicative homomorphic scheme. While important, these schemes are not useful in performing arbitrary computations on encrypted data. Hence, the above mentioned motivating applications cannot be realized by partially homomorphic schemes.

Somewhat Homomorphic Encryption (SWHE) schemes are more powerful than partially homomorphic encryption schemes. These schemes support both addition and multiplication operations on encrypted data, but for a limited (pre-defined) number of times. In addition, these schemes are relatively efficient (see Table 1.1 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.

Fully Homomorphic Encryption 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 [12]. 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.

Table 1.1: 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 [11]	1978	Partial	×	×	128	0.9	×	5
Paillier [13]	1999	Partial	×	×	128	0.3	4	×
TFHE [1]	2016	Fully	Exact	AVX [14]	110	31.5	7044	4,89,938
HEEAN [2]	2018	Somewhat	Approximate	CPU	157	7,168	11.37	1,215
SEAL (BFV) [3]	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
TFHE (GPU)	Our proposal	Fully	Exact	GPU	110	31.5	1,991	33,930

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 [18, 19, 20]. 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 thesis, we assume that the computational entity (e.g., cloud server) is standalone, and we show that parallelism can be used to lower the cost of FHE instead of relaxing the security assumptions for the computation model.

1.1.1 Why TFHE?

Since the inception of FHE by Gentry [21], there has been notable advancements towards its asymptotic performance [22, 23, 24]. There have been several attempts on faster and more efficient FHE schemes [25, 26, 27], which are pivotal to this work (Section 3 for details). Among the schemes, Torus Fully Homomorphic Encryption (TFHE) [1] is one of the most renowned FHE scheme 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 1.1). 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.

1.1.2 Why GPU?

FHE schemes are based on the Learning With Error (LWE) paradigm and its variant. Each plaintext is encrypted using a polynomial, and it is represented by vectors. As a result, most of the computations are based on vector operations that are highly parallelizable.

Graphics Processing Units (GPUs) offer a large number of computing cores (in thousands compared to CPUs). These cores can be utilized to compute parallel vectors operations. However, GPUs have some limitations as their global memory is fixed (8 to 16 GB) and have a reduced computing power compared to any CPU core. Based on the computations in FHE and the working principle of GPU, our observation is that if the cores are utilized properly, we can execute FHE computations efficiently.

Due to the binary plaintext space of TFHE, the computations are basically boolean gate operations. Therefore, to take the advantage of GPU in higher level circuits (addition, multiplication, vector, matrix), it demands the implementation at the gate level using GPU first. Hence, our construction starts with the gate level parallelism in GPU. Additionally, the availability of a large number of cores facilitates parallel gate operations. That is, parallel execution of different gate operations. Thus, we move gradually to the parallelization of higher level computations like addition, multiplication, vector, and matrix operations. These operations consequently can be used in other higher level computations (e.g., logistic regression, deep learning).

Table 1.2: A comparison of the execution times (in seconds) of our CPU and GPU parallel framework for 32-bit numbers. The length of the vectors for vector addition is 32. The computation times for the matrix multiplication (16×16) are in minutes.

	Gate Op.	**Addition**		**Multiplication**	
		Regular	**Vector**	**Regular**	**Matrix**
TFHE [1]	1.40	7.04	224.31	489.93	8717.89
CPU-Parallel	0.50	7.04	77.18	174.54	2514.34
GPU-Parallel	0.07	1.99	11.22	33.93	186.23

1.2 Contributions

We summarize the contributions of this thesis below.

- The existing TFHE implementation offers encryption functions and boolean gate operations such as AND, OR, and XOR. In this thesis, we use these boolean gates to construct higher level algebraic circuits such as addition and multiplication. The circuits are constructed sequentially to measure the runtime of the existing method (TFHE).

- We utilize CPU-level parallelism in the circuit construction to exploit computational resources available in the multi-core machines. To take the advantage of the available resources (cores), we adapt the sequential circuits and incorporate parallelism at the CPU-level. Experimental results demonstrate the advantage of higher level circuit construction using multi-threading over the naive sequential implementation.

- The primary contribution of this thesis is to port TFHE homomorphic oper-

ations to GPU. We have re-implemented the basic gates (i.e., AND, XOR) using the GPU framework, and used novel optimization techniques such as bit coalescing (coalescing the ciphertexts in contiguous memory), compound gate (multiple gate computation in parallel), and tree-based vector addition to implement the higher level algebraic circuits (addition and multiplication). Note that without these optimization techniques, the GPU-based solution does not yield a faster solution. The code is readily available at GitHub (`https://github.com/tmp1370/tmpProject`).

- We have done extensive experiments to compare the computation time of the existing sequential TFHE [1] with the proposed CPU-based and GPU-based parallel implementations. As shown in the Table 1.2, our proposed GPU-based method is more than 14.4× and 46.81× faster than the existing technique for the multiplication and the matrix multiplication operations, respectively.

- We evaluated and analyzed the performance of our constructions with existing GPU-based TFHE frameworks, namely, cuFHE [15] and NuFHE [16]. We also benchmarked with Cingulata [17] (a CPU-based compiler toolchain for running C++ programs over encrypted data using TFHE), where our framework outperforms in arithmetic circuit computation as well.

- The approximation of non-linear functions (used in machine learning algorithms) over encrypted data is another novel contribution of the thesis. We compared the approximation error with existing proposals.

- We employed our GPU accelerated framework to carry out logistic regression

on Parkinson's disease data (available at UCI-repository [28]) and analyzed the execution time with [29]. Besides, we take the leverage of binary gate operations for binary inputs and discuss the construction as well.

1.3 Organization

The rest of the thesis is organized as follows.

- Chapter 2 discusses necessary background materials utilized in different methods proposed in this thesis.
- Chapter 3 presents some current works related to this thesis.
- Chapter 4 describes our proposed parallel frameworks using CPU and GPU.
- Chapter 5 presents the non-linear function approximation over encrypted data. Then, it describes the logistic regression as an application of our proposed GPU-parallel frameworks.
- Chapter 6 presents a detailed experimental as well as benchmarking analyses of the proposed framework.
- Finally, Chapter 7 concludes the thesis.

Chapter 2

Preliminaries

In this chapter, we describe the required background briefly. Table 2.1 lists the notations used in the thesis.

2.1 Torus FHE (TFHE)

In this work, we closely investigate a Fully Homomorphic Encryption (FHE) scheme, Torus FHE (TFHE) [1] (incremental to [25]). In TFHE, 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 on the Learning with Errors (LWE) [12] represented as Torus LWE (TLWE) where an error term (drawn from Gaussian distribution χ) is added with 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 $\mathbb{T}$. The other part B is

Table 2.1: Notations used throughout the thesis

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
$A \in \mathbb{Z} \subset \mathbb{B}^n$	Integer number of n-bits
$a_i \in \mathbb{B}$	Binary bit at position i
$\mathbb{L}^n$	LWE sample vector of length n
A'	One's complement of Integer A
n, m	Bit size and Secret key size
$\|\|$, $\ll$	Parallel and Left Shift operation
$\wedge$, $\|$, $\oplus$	Binary AND, OR and XOR operation

defined as:

$$B = \overrightarrow{A} \cdot \overrightarrow{S} + e \tag{2.1}$$

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 maintain the data integrity/proper decryption. Figure 4.2 illustrates a schematic diagram of a gate computation (e.g., $\wedge$) in TFHE.

TFHE considers 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 thesis, we use *bit* and *LWE sample* interchangeably.

TFHE Properties: For our parallel framework, we choose TFHE for the following reasons:

- **Fast and Exact Bootstrapping.** TFHE provided the fastest bootstrapping technique ($\approx 0.1s$) as claimed in [1] in 2016. Some recent encryption schemes [2, 30] (and their implementations) propose faster bootstrapping and FHE computations in general. However, they do not have exact bootstrapping and larger ciphertext size.

- **Encryption Size.** TFHE requires lower storage for the encrypted data compared to the other FHE schemes. For example, one 32-bit encrypted integer require storage in terms of Kilobytes, whereas the other schemes may need a few Megabytes. This difference is further discussed in Section 3.

- **Boolean Gate Operation.** TFHE also uses boolean/logical gates (`AND`, `XOR`, etc.), which can be employed to construct algebraic operations. Furthermore, these binary bits can 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) [14]. AVX (or more refined AVX2) is an extension to x86 instruction set from Intel. It facilitates parallel vector operations in CPU. For example, we need to compute $\overrightarrow{R} = \overrightarrow{P} + \overrightarrow{Q}$ where $\overrightarrow{P}$, $\overrightarrow{Q}$ and $\overrightarrow{R}$ are vectors of numbers ($\mathbb{Z}^{\ell}$). To compute $\overrightarrow{R}$ using x86 instruction set, we need ℓ additions one by one (sequentially). However, AVX2 facilitates instructions to execute ℓ additions in parallel, providing a noticeable speedup.

Furthermore, 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) [31] which inherently uses AVX (if available). Despite the AVX usage, the existing TFHE implementation computes multiple boolean gates in sequential manner.

2.2 Parallelism

In this work, we employ and analyze two structurally different but conceptually similar parallel frameworks described below:

2.2.1 Hardware Parallelism

Our **CPU Parallel** framework utilizes the cores and existing resources (i.e., Vector Extensions) readily 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

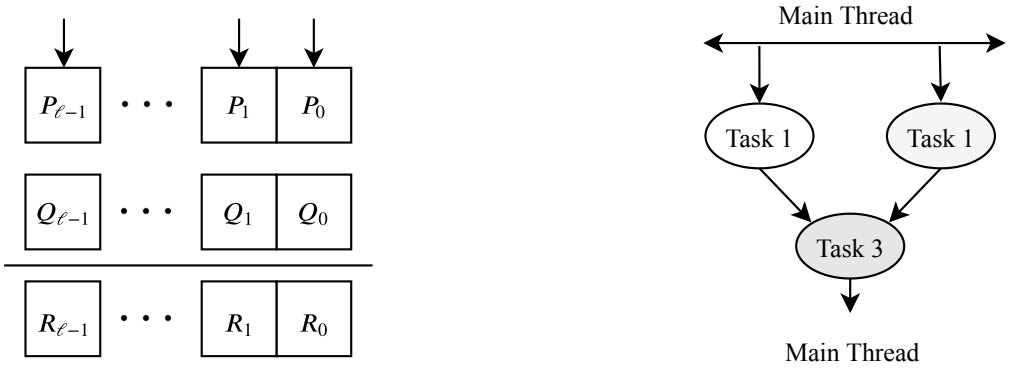
available cores. Contemporary desktop computers come with 4 to 8 cores containing a maximum of 16 threads. There have been multiple attempts [32] to use a large number of CPUs collectively for parallel operations, whereas we show that single GPU is equivalent (and better performing) for most FHE operations.

In contrast to CPUs, **GPU Parallel** framework offers a significant number of cores (/threads) available solely from the hardware. Thus, the expense of increasing cores with multiple CPUs is reduced by integrating one GPU. For example, our GPU hardware consisted $40,960$ cores [33] compared to a regular Intel i7 machine with 8 cores. More details on the GPU architecture are in Section 2.3.

This difference in available cores has granted much-needed parallelism in every prevailing deep learning frameworks, speeding up the training process for massive datasets. However, two major shortcomings in employing GPUs are—a) limited global memory and b) communication time through PCIe. For example, we employed a GPU with only 8 GB fixed global memory. Thus, we considered these issues along with some others when designing the proposed framework.

2.2.2 Software Parallelism

Data-level Parallelism traditionally means the execution of one operation over all existing data. Data parallel paradigm is also known as Single Instruction Multiple Data (SIMD) model [34] where multiple threads execute the same operations for different data. Figure 2.1a illustrates this data parallelism where each thread computes the same operation ($\overrightarrow{R} = \overrightarrow{P} + \overrightarrow{Q}$ where $\overrightarrow{P}, \overrightarrow{Q}, \overrightarrow{R} \in \mathbb{Z}^{\ell}$) on different data. It is important to note that both CPU and GPU support such parallelism.



(a) Data parallel

(b) Task parallel

Figure 2.1: Data-level and task-level parallelism architecture. Arrows and oval shapes represent threads and tasks, resp.

Task-level Parallelism, on the contrary, can handle different computations on the same/different data. In Figure 2.1b, we illustrate this paradigm where the computation of Task3 (t_3) depends on Task1 (t_1) and Task2 (t_2). However, t_1 and t_2 are independent of each other and can be executed in parallel. Hence, two threads are used to compute them in parallel and only then t_3 is executed. Although CPUs support task-level parallelism, GPUs lack this feature. Therefore, in our framework, we provide a construction of compound gate (Section 4.2.1) which is analogous to task-level parallelism.

2.3 GPU Architecture

2.3.1 Computational Hierarchy

A Graphics Processing Unit (GPU) consists of a scalable array of multithreaded streaming multiprocessors (SMs). For example, our GPU for the experiments, Nvidia

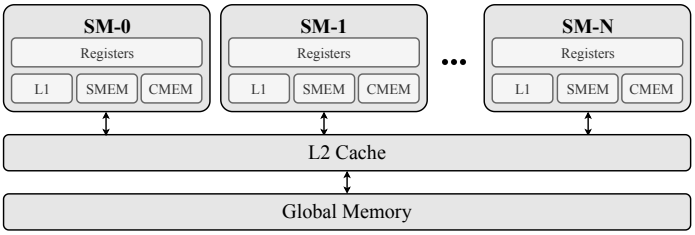


Figure 2.2: GPU memory hierarchy.

GTX 1080 has 20 SMs where SM contained 2048 individual threads. Threads are grouped into blocks, and the blocks are grouped to form Grids. Threads in the blocks are split into warps (32 threads) in the same SM.

For the computational unit, the GPU includes 128 CUDA (Compute Unified Device Architecture) cores per SM. Each core execution unit has one float, and one integer compute processor.

2.3.2 Memory Hierarchy

SMs can run in parallel with different instructions. However, all the threads of a respective SM execute the same instruction simultaneously. Therefore, GPUs are called Single Instruction Multiple Data (SIMD) machines. Besides having a large number of threads, the GPU memory system also consists of a wide variety of memories for the underlying computations. Architecturally, we divide the memory system into five categories: *a)* register, *b)* cache, *c)* shared, *d)* constant, and *e)* global. Figure 2.2 portrays the memory categories and their organization. We present a brief discussion on the memory categories.

Register. Registers are the fastest and smallest among all memories. Registers are private to the threads.

Cache. GPUs facilitate two levels of caches, namely: L1 cache and L2 cache. In terms of latency, the L1 cache is below the registers. Each SM is equipped with private L1 cache. On the contrary, the L2 cache is with latency more than the L1 cached and shared by all SMs.

Shared Memory. Being on the SM chip, shared memory has higher bandwidth and much lower latency than the global memory. It has much lower memory space and lacks volatility.Like L1 caches, shared memory is private to SMs as well, but public to the threads inside the respective SMs.

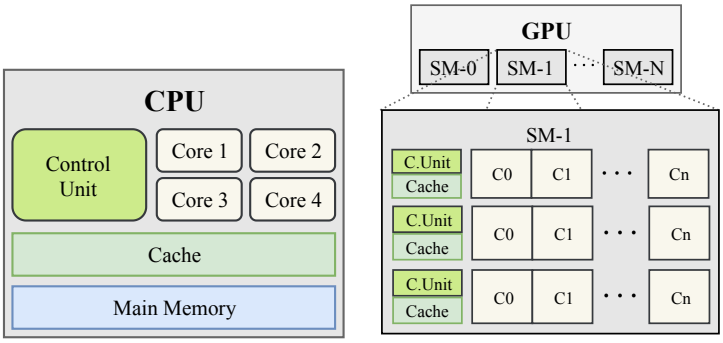
Constant Memory. The constant memory resides in the device memory and is cached in the constant cache. Each SM has its own constant memory. Constant memory increases the cache hit for constant variables.

Global Memory. Global memory is the largest (Table 2.2) among all memory categories, yet the slowest and non-persistent. One major limitation of the global memory is that it is fixed, while the main memory can be changed for the CPUs.

Computational and Memory Hierarchy Coordination

The coordination between computation and memory hierarchy is a crucial aspect to take the advantage of both faster memory and parallelism. Each thread has private local variable storage known as registers. Threads inside the same block can access the shared memory, constant memory, and L1 cache. The memories for one block is inaccessible by others inside the same SM. The number of grids can be at most the number of global memory, and the global memory is shareable from all SMs.

Bit Coalescing (Section 6.1.2) discusses the unification of LWE-samples. Hence,



(a) CPU architecture (b) GPU architecture

Figure 2.3: A schematic illustration of CPU (a) and GPU (b) architecture. Unit and C_n represents a control unit and a core in GPU, respectively. Figure (b) illustrates an SM construction

for a sufficiently large n-bit (LWE-sample) coalescing, the memory requirement exceeds the existing shared memory. Therefore, the current GPU || construction uses the global memory (the slowest).

The rest of the computations use registers to store the thread specific local variables, and shared memory to share the data among the threads.

2.3.3 Architectural Differences with CPU

We present the differences of CPU and GPU architecture in terms of following metrices:

Number of Cores Modern CPUs consist of a small number of independent cores and thus confine the scopes of parallelism. GPUs, on the other hand, have an array of SM, where each SM possesses a large number of cores. For example, in Figure 2.3, the CPU comprises of 4 independent cores, while the GPU consists of N SMs with

Table 2.2: A comparison between Intel(R) Core™ i7-2600 and Nvidia GTX 1080 configurations

	CPU	GPU
Clock Speed	3.40 GHz	1734 MHz
Main Memory	16 GB	8 GB
L1 Cache	256 KB	48 KB
L2 cache	256 KB	2048 KB
L3 cache	8192 KB	×
Physical Threads	8	40,960

n cuda cores in each SM. Thus, GPUs offer more parallel computing power for any computation.

Computation Complexity. Although GPUs provide more scopes of parallelism, GPU cores lack the computational power with respect to CPU. CPU cores have higher clock rate (3.40GHz) than GPU (1734 MHz) as showed Table 2.2. Moreover, CPU cores are capable of executing complex instruction of small data. On the contrary, GPU core is simple, typically consists of an execution unit of integers and float numbers [35].

Memory Space Table 2.2 provides the storage capacity of different types of memory in the machines. Additionally, a unique aspect of CPUs is that the main memory can be modified on the H/W. GPUs lack this facility as every device is shipped with fixed size memory. This creates additional complexities like memory exhaustion while computing with large dataset/models.

Number of Threads In modern desktop machines, the number of physical threads is equal to the number of cores. However, hyperthreading technology virtually doubles the number of threads. Thus, the CPUs can have virtual threads twice the number of cores. GPUs, on the contrary, provide thousands of cores. In GTX 1080, the total number of threads is $40,960$. Therefore, the GPU is faster in data parallel algorithms.

2.4 Sequential Framework

Arithmetic computations on ciphertexts (i.e., additions, multiplications) can be devised from boolean gates (i.e., $\wedge$, $\oplus$). Here, we present a brief overview of the sequential arithmetic circuit constructions using boolean gates.

2.4.1 Addition

A 1-bit full adder circuit takes two inputs along with a carry bit 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 serial dependency enforces the addition operation to be sequential for n-bit numbers [36, 37].

2.4.2 Multiplication

Algorithm 1 Naive Multiplication

Input: A, B $\in \mathbb{B}^n$

Output: C $\in \mathbb{B}^{2n}$

1: $C \leftarrow 0$

2: $temp \leftarrow 0$

3: **for** $i \leftarrow 0$ **to** $n-1$ **do**

4: **for** $j \leftarrow 0$ **to** $n-1$ **do**

5: $temp_j \leftarrow a_j \wedge b_i$ //subscripts represent bits

6: **end for**

7: $temp \leftarrow temp \ll i$

8: C $\leftarrow$ C $+temp$

9: **end for**

10: **return** C

Naive Approach

For two n-bit numbers $A, B \in \mathbb{Z} \subset \mathbb{B}^n$, we multiply (`AND`) the number A with each bit b_i of B, resulting in n numbers, where each number is of n-bits. Then, these numbers are `left shift`ed by i bits (multiplied by 2^i) individually resulting in $[n, 2n]$-bit numbers. Finally, we `accumulate` (reduce by addition) these n shifted numbers using the addition operation. Algorithm 1 demonstrates the textbook multiplication algorithm with $O(n^2)$ complexity, which can be performed in parallel (details on Section 4.1.2 and 4.2.2).

Karatsuba Algorithm

Algorithm 2 Karatsuba Multiplication [38]

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 if**

4: $k \leftarrow n/2$

5: $X_0 \leftarrow X \mod 2^k$

6: $Y_0 \leftarrow Y \mod 2^k$

7: $X_1 \leftarrow X/2^k$

8: $Y_1 \leftarrow Y/2^k$

9: $Z_0 \leftarrow$ `KaratsubaMultiply`(X_0, Y_0)

10: $Z_1 \leftarrow$ `KaratsubaMultiply`(X_1, Y_1)

11: $Z_2 \leftarrow$ `KaratsubaMultiply`($X_0 + Y_0$, $X_1 + Y_1$)

12: **return** $Z \leftarrow Z_0 + (Z_2 - Z_1 - Z_0)\ 2^n + (Z_1)2^{2n}$

We also consider the divide-and-conquer Karatsuba's algorithm for its time complexity $O(n^{\log^3})$ [38]. The idea relies on dividing the n-bit inputs and performing smaller multiplications. For example, for two n-bit inputs, Karatsuba's algorithm splits them into four smaller numbers ($n/2$-bit size) and replaces the original multiplication by additions and multiplications on the smaller numbers (Line 12 of Algorithm 2). It is noteworthy that we will utilize our parallel vector operations for further optimizations with respect to this algorithm.

Chapter 3

Related Works

The chapter discusses some of the related works on FHE and the non-linear function approximation.

3.1 Homomorphic Encryption

We now further discuss Table 1.1 that presents a comparative analysis of the different homomorphic crypto schemes and their features. These popular frameworks are relevant to our work, and we discuss the commonality and difference while seeking to compare them on the same settings. Evaluating all these frameworks are challenging due to their complex and dissimilar parameterization. Therefore, there are some recent developments towards standardizing the available HE schemes [39].

We are differentiating the Homomorphic Encryption (HE) schemes based on their number representation: *a)* bit-wise, *b)* modular and *c)* approximate. Note that these schemes can be categorized differently based on theoretical results [40, 41]. It is

noteworthy that such categorization is solely for analytical purpose whereas there are different theoretical categorization available [40, 41].

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 [42] around 2013, and it was later improved in subsequent works [25, 1, 43].

Modular Encryption schemes utilize a fixed modulus q which denotes the size of the ciphertexts. There have been many developments [44, 45] in this direction as they offer a reasonable execution time as shown in Table 1.1. The addition and multiplication execution times from FV-NFLlib [26] and SEAL [3] show the difference as they are much faster compared to our GPU-based parallel 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 via bootstrapping. Decryption is performed after the desired computation. Furthermore, the encrypted data evidently suffers from larger ciphertexts as the value of q is intentionally picked from large numbers.

For example, we selected the ciphertext modulus of 250 and 881 bits for FV-NFLlib [26] and SEAL [3], 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 1.1. 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 the developer to fix the number of homomorphic operations, which limits the use cases and incurs complex parameterization.

Approximate Number representations are recently proposed by Cheon *et al.* (CKKS [46]) in 2017. These schemes also provide efficient Single Instruction Multiple Data (SIMD) [34] 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 $d = 15$.

We did not discuss HELib [27], which is the first and cornerstone to all HE implementations. This is due to its cryptosystem BGV [45] that is enhanced and utilized by the other modular HE schemes (such as SEAL [3]). We enlist the available cryptosystems in Table 1.1.

One of the goals of this work is to parallelize an FHE scheme. Most of the HE schemes that follow modular encryption are either somewhat or adopt inexact bootstrapping. Additionally, their expansion after encryption also requires more memory. Therefore, 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 [21] with ideal lattices, most of the effi-

ciency improvements are considered from the standpoint of asymptotic runtimes. A few approaches considered the incorporation of existing multiprocessors (e.g., GPU) or FPGAs [47] to achieve faster homomorphic operation. Dai and Sunar ported another scheme LTV [48] to GPU-based implementation [49, 50]. LTV is a variant of HE that performs a limited number of operations on a given ciphertext. Moreover, in their implementation, they offload the vector operations in GPU incurring communication between CPU (host) and GPU (device). For faster polynomial multiplication they used SchnhageStrassen algorithm [51] which implicitly uses Fast Fourier transforms. None of these works address the problem of parallelizing the FHE schemes, which is the main theme of the thesis.

cuFHE, a CUDA-accelerated FHE Library, was released in 2018 [15]. cuFHE takes advantage of the streaming multiprocessors for gate computations. However, their implementation did not consider the sequential gate computations in arithmetic circuits, where the gate operations can not be parallelized. Soon after cuFHE, NuFHE, another GPU accelerated TFHE came out in 2018 [16]. Like cuFHE, NuFHE focuses on gate-level optimization, while our focus is on arithmetic circuit optimization.

3.2 Non-linear Function Computation

The non-linear function circuit computations are more complex than the arithmetic ones. Hence, it requires different measures. Here, we discuss some of the related works on approximations based on interaction among the servers and data owners.

- In **Interactive protocols**, the servers communicate among themselves, sometimes, even with the data owner/s, to compute the non-linear functions. Or-

landi *et al.* [52] proposed their model based on the interaction with the data owner to compute the non-linear functions. DeepSecure [53] and Gazzle [19] use two-party computation [54] architecture for non-linear function approximation.

- **Non-interactive Protocol**, on the contrary, approximates the non-linear functions in a standalone server. CryptoNet [55] discussed the concept of the approximation while computing the functions. Later, Hesamifard *et al.* proposed CryptoDL [20, 56] using Chebyshev Polynomial form of interpolation [57] to approximate the non-linear functions.

In this thesis, we propose an approximation of non-linear functions in a non-interactive fashion. Our novelty in the approximation is the integration of the encrypted comparison circuit that reduces the error to a great extent considering the error incurred by Hesamifard *et al.* in [20, 56].

Chapter 4

Proposed Frameworks

4.1 CPU-based Parallel Framework

We propose a CPU parallel framework (CPU ||) which utilizes multiple cores available in typical commodity computers. Notably, we utilize the existing TFHE implementation which comes with an extensive usage of AVX2 (discussed in Section 2.1) available in each core. Therefore, our multi-core implementations have inherently used such optimizations as we exhaustively employ every resource on a given CPU.

4.1.1 Addition

Figure 4.1 illustrates the bitwise addition operation considered in our CPU framework. Here, any resultant bit r_i depends on its previous c_{i-1} bit. In other words, there is a serial dependency between the adjacent bits, which restricts us from incorporating any data-level parallelism in the addition circuit construction.

However, it is possible to exploit some task-level parallelism where two different

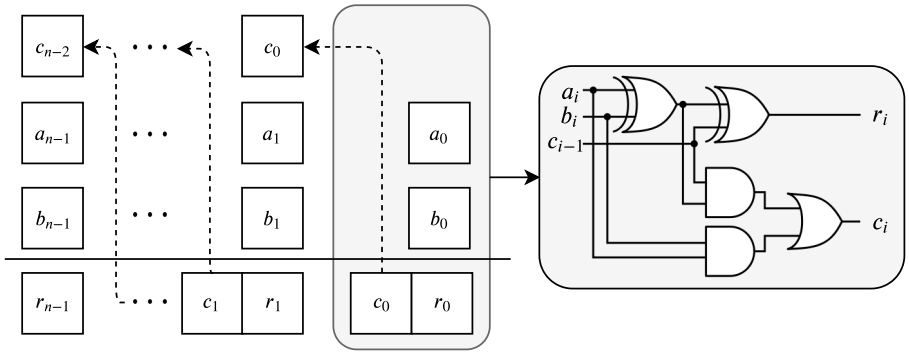


Figure 4.1: Bitwise addition of two n-bit numbers A and B. Each box represents a bit. a_i, b_i are i^{th}-bit of A and B, where c_i and r_i represent carry and resultant of i^{th}-bit, resp.

threads execute the `XOR` ($\oplus$) and `AND` ($\wedge$) operations (Figure 4.1), simultaneously. We observed that the time required to perform such task distribution between two threads is higher than executing them serially. This is partially due to the costly thread operations and eventual serial dependency of the results. Therefore, we did not employ this technique for CPUs whereas employed an alternative for GPUs (see Section 4.2).

4.1.2 Multiplication

Out of the three major operations (`AND`, `left shift`, and `accumulation` as mentioned in Section 2.4.2), the `AND` and left shifts (Line 3 – 6 in Algorithm 1) 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 different threads. That is, Thread $0, 1, 2$, and 3 compute simultaneously on the bit positions $0-3$, $4-7, 8-11, 12-15$, respectively. Such distribution of workload among different

threads is known as the work sharing principle.

The accumulation operation, on the contary, 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 [58]. Given that it is computationally expensive, we do not employ this technique in any parallel framework.

We utilized a custom reduction operation in OpenMP [58], which uses the global shared memory (in CPU) to store (and add) 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.

We utilized the reduction operation in OpenMP [58] which uses the global shared memory (in CPU) to store (and add) the in-between results. Since we are adding ciphertexts (LWE Samples), we implemented a custom reduction function solely to avoid the global thread synchronization. This resulted in much better performance compared to the naive approach of waiting on all threads to complete their individual tasks.

4.1.3 Vector operations

To compute the vector operations (addition, multiplication) efficiently, we distribute the work into multiple threads (work sharing). 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. Given that the computation of each position of $\overrightarrow{C}$ is

independent. Therefore, in a multi threaded machine, we can share the work among different threads. For vector multiplication, we take similar measure for work sharing as well. (see Section 6 for experimental results).

4.1.4 Matrix operations

Matrix Addition: Matrix addition is a series of addition operations between the elements (analogous to a number or an encrypted number) of two matrices. The elements have to be from the same position of the matrices, and the addition operations between them are independent of each other. Therefore, we divide the matrices row-wise and divide the additions to different threads, where each thread operates on different positions.

Matrix Multiplication: Multiplication over the matrices is a bit more complicated than addition. It consists of both multiplications and additions.

$$\mathbf{X} \times \mathbf{Y} = \begin{bmatrix} X_{00}Y_{00} + X_{01}Y_{10} & X_{00}Y_{01} + X_{01}Y_{11} \\ X_{10}Y_{00} + X_{11}Y_{10} & X_{10}Y_{00} + X_{11}Y_{10} \end{bmatrix} \tag{4.1}$$

Equation 4.1 provides a schematic computation of a 2×2 matrix multiplication. Inspecting the calculation of the resultant matrix, we highlight three major observations on the parallelism of matrix multiplication.

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

For example, the computations of the first element ($X_{00}Y_{00}+X_{01}Y_{10}$) do not depend on other elements of the resultant matrix. The multiplication operations ($X_{00}Y_{00}, X_{01}Y_{10}$) in the computation are independent as well (same for other indices). 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 4.2.2.

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. For example, the X_{0i} and Y_{i0} are sent to one core to compute the $(\mathbf{XY})_{00}$ of the resulting matrix.

4.2 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.

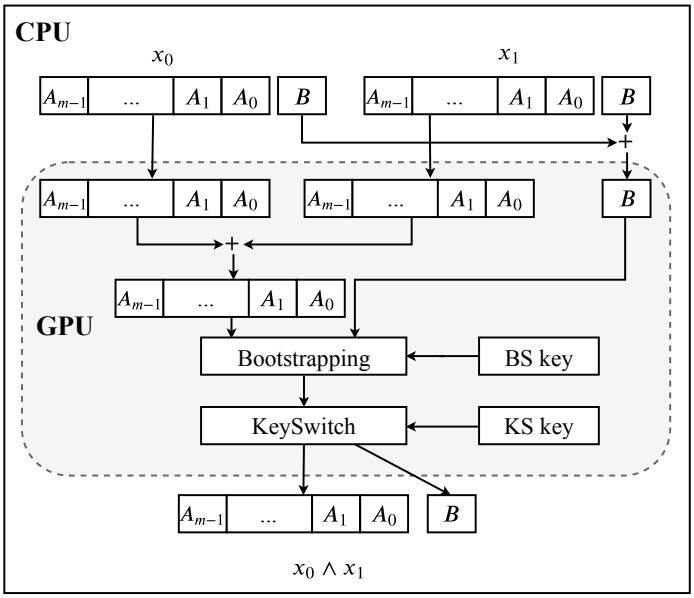


Figure 4.2: Arbitrary operation (e.g., `AND`) between two bits, where the vector operations are done inside the GPU. BS, KS key represent bootstrapping and key switching keys, respectively

4.2.1 Proposed Optimization Techniques

Parallel TFHE Construction

We depict the boolean circuit computation (gate level) in Figure 4.2. Here, each LWE sample (ciphertext) 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 vector of 32-bit integers with a length of the key size (m) allowing much lower memory size compared to other FHE implementations (further discussed in Section 3). In our parallel TFHE construction, we only store $\overrightarrow{A}$ on the GPU's global memory.

In addition to performing all vector operations inside the GPU, we use NVIDIA CUDA FFT library (cuFFT) for FFT operations instead of the existing libraries. We

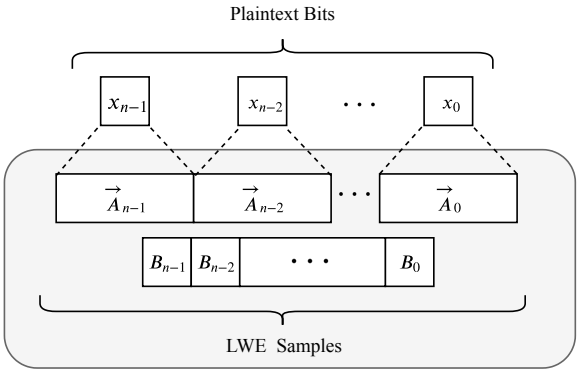


Figure 4.3: Coalescing n-LWE samples (ciphertexts) for n-bits (plaintext)

utilized the parallel batching technique from cuFFT to support multiple FFT operations 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].

Bit Coalescing (BC)

Bit Coalescing combines n LWE samples in a contiguous memory to represent the corresponding n-encrypted bits together. 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. In BC construction, instead of treating the vectors of ciphertexts separately, we coalesce them altogether (dimension $1 \times mn$) as illustrated in Figure 4.3.

Thus, instead of computing the same homomorphic operation n times sequentially, we compute them in parallel. Consequentially, we found that homomorphic computations using BC reduce the execution time for larger n.

The intuition behind such construction is to increase parallelism by increasing the

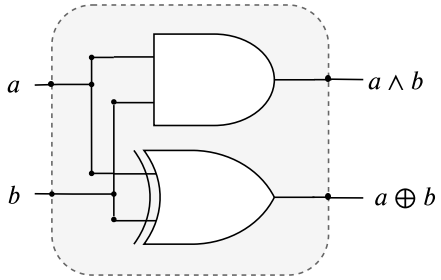


Figure 4.4: Compound gate construction with two input bits (a and b) and two output bits ($a \wedge b$ and $a \oplus b$)

vector length in a contiguous memory. Coalescing the vector increases vector length and thus, we incorporate more threads to achieve maximum parallelization and less execution time. The effect of such efficiency is evident in Section 6.2.

Compound Gate

While adding two numbers $A, B \in \mathbb{B}^n$, we can optimize the underlying boolean circuit further by proposing a new gate structure— Compound Gate. These gates are a hybrid of multiple gates, which takes the same two inputs as an ordinary boolean gate but outputs two different outputs based on the instructions. The motivation behind this novel gate structure comes from the addition circuit. For $R = A + B$ *s. t.* $R \in \mathbb{B}^{n+1}$, we compute r_i and c_i with the following equations:

$$r_i = a_i \oplus b_i \oplus c_{i-1} \tag{4.2}$$

$$c_i = a_i \wedge b_i \mid (a_i \oplus b_i) \wedge c_{i-1} \tag{4.3}$$

Here, r_i, a_i, b_i, and c_i denotes i^{th}-bit of R, A, B, and the carry bit, respectively. Figure 4.1 also illustrates this computation for an n-bit addition.

While computing the equations 4.2, and 4.3, we observe that `AND` ($\wedge$) and `XOR`

($\oplus$) are computed on the same input bits. These operations are independent of each other, and it is reasonable to combine them into a single gate, which then can be computed in parallel. We name these gates as *compound gate*. Figure 4.4 portrays the concept of a compound gate circuit for performing $a \wedge b$, $a \oplus b$, simultaneously. Thus, $a \oplus b$ and $a \wedge b$ from equation 4.2 and 4.3 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 gate construction is analogous to the task-level parallelism in CPU, where one thread performs $\wedge$, while another thread performs $\oplus$.

In GPU ||, the compound gate 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.

4.2.2 Algebraic Circuits on GPU

Addition

Bitwise Addition ($\mathbf{GPU_1}$): From the addition circuit in Section 4.1.1, 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 gate construction. Hence, we incorporated the compound gate to construct the bitwise addition circuit. We also implemented the vector addition circuits using GPU_1 to support complex circuits such as multiplications (Section 4.2.2).

Number-wise Addition ($\mathbf{GPU}_n$): In algorithm 3, we present another addition

Algorithm 3 Number-wise Addition

Input: A, B $\in \mathbb{B}^n$

Output: R $\in \mathbb{B}^{n+1}$

1: R $\leftarrow$ A
2: **for** $i \leftarrow 0$ **to** $n-1$ **do**
3: Carry $\leftarrow$ R $\wedge$ B
4: R $\leftarrow$ R $\oplus$ B
5: B $\leftarrow$ Carry $\ll 1$
6: **end for**
7: **return** R

technique that encapsulates the bitwise addition capturing the efficiency proposed by bit coalescing (Section 4.2.1). Here, we operate on the whole number instead of one bit (in bitwise addition) and reduce the number of operations from equation 4.2 and 4.3.

We can also utilize the compound gates to perform $R \wedge B$ and $R \oplus B$ in parallel. However, the number-wise representation will increase the ciphertext size for BC whereas bitwise operations will only consider a single bit. Therefore, operating on larger n-bit numbers ($n > 24$) can be detrimental as it will surpass such optimizations (more details on Section 6).

Multiplication

Naive Approach: The multiplication operation of the framework adopts the parallel computation of the first two operations: $\wedge$ and $\ll$, resulting in n-numbers (LWE

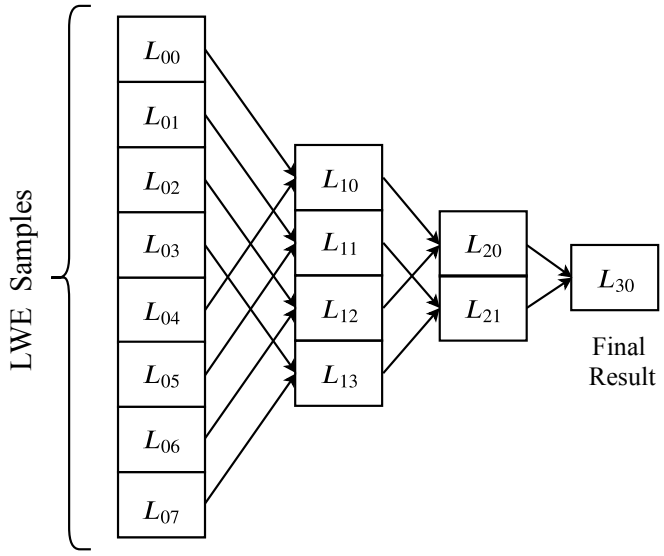


Figure 4.5: Accumulating LWE samples in parallel using a tree based reduction for $n = 8$ where L_{ij}s are added in parallel

sample vectors in ciphertext) ranging over $[n, 2n]$-bits. We need to accumulate these numbers which cannot be distributed among the GPU threads. Furthermore, it offers another sequential bottleneck while adding and storing the results in the same memory location (for $+=$ operation). 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 sample vectors) into two $n/2$ vectors and perform an addition operation that results in $n/2$ outputs. We again distribute the resultant vectors between two $n/4$ vectors and add them. The process repeats until we get a single value. Eventually, the tree-based approach requires $\log n$ steps for the accumulation.

We illustrate the tree-based accumulation approach in Figure 4.5 for $n = 8$, where

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 the framework. More interestingly, we used both vector additions and multiplications in Karatsuba's Algorithm which we describe next.

Karatsuba Multiplication: We include Karatsuba's Algorithm in our framework to achieve further efficiency while performing multiplications. However, this algorithm requires vector operations (addition and multiplication) and tests the efficacy of these components. We were required to modify the original Algorithm 2 to introduce the vector operations and rewrite the computations in Line 9 – 12 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, respectively.

It is noteworthy that in the CPU || framework, we utilized task-level parallelism to perform these vector operations as described in Section 4.1.

Vector and Matrix Operations

Vector and Matrix 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 in parallel.

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.

Vector and Matrix Multiplication: Analogous or 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 4.2.2).

Unlike matrix addition, its multiplication is more complicated as it requires more computations. Section 4.1.4 presents a schematic computation for a 2×2 matrix and discusses the scopes of parallelism while computing it. Notably, for two n-ranked *squared* (for brevity) matrix multiplication, we require n^3 multiplication operations. One way to do this is to separate all the multipliers and multiplicands into two vectors and perform parallel vector operation. 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 large vectors will require a hefty amount of GPU memory which we need to avoid. It will also require a large number of threads both for computing and storing the LWE samples. Although, the exiting thread can be reused sequentially, the problem is severe for the fixed GPU memory constraint. Therefore, matrix multiplications on larger dimensions can essentially run out of GPU memory.

Therefore, we consider a different technique to solve the issue— Cannon's Algorithm [59]. Algorithm 4 illustrates the computations involved in Cannon's algorithm. One disadvantage of the algorithm is the presence of a sequential block (Line 6 in Algorithm 4). Instead of computing all multiplications in parallel, it propagates in cycles consisting of multiplication, addition and shifting the matrix elements.

For each cycles of the algorithm, we first perform the multiplication operation in parallel (Line 9) exploiting the vector multiplications. Later, we compute the addition on the computed data, which also takes advantage of the parallel vector additions. Lastly, we shift the elements' positions for the next round, as mentioned in the algorithm.

Algorithm 4 Cannon's Algorithm [59]

Input: $\mathbf{X}, \mathbf{Y} \in \mathbb{Z}^{d \times d}$

Output: $\mathbf{Z} \in \mathbb{Z}^{d \times d}$

1: $\mathbf{Z} \leftarrow 0$
2: **for** $i \leftarrow 0$ **to** $d - 1$ **do**
3: left-rotate row i of $\mathbf{X}$ by i
4: up-rotate column i of $\mathbf{Y}$ by i
5: **end for**
6: **for** $k \leftarrow 0$ **to** $d - 1$ **do**
7: **for** $i \leftarrow 0$ **to** $d - 1$ **do**
8: **for** $j \leftarrow 0$ **to** $d - 1$ **do**
9: $\mathbf{Z}[i, j] \leftarrow \mathbf{Z}[i, j] + \mathbf{X}[i, j]\mathbf{Y}[i, j]$
10: **end for**
11: **end for**
12: left-rotate of each row of $\mathbf{X}$ by 1
13: up-rotate of each row of $\mathbf{Y}$ by 1
14: **end for**
15: **return** $\mathbf{Z}$

Chapter 5

Regression Analysis

5.1 Logistic Regression

Logistic Regression is a well known and widely used binary classifier algorithm. The algorithm takes attributes of an instance as input, computes their characteristics on a combination, and generates a probability of classification. In this thesis, we consider the Parkinson dataset(D) [60] with u-instances, where each instance $\overrightarrow{D_i}$ has v-attributes. Here, $\overrightarrow{D_i}$ is a vector of numbers and represented as,

$$\overrightarrow{D_i} = (D_i^0, D_i^1, D_i^2, \ldots, D_i^{v-1})$$

where D_i^j represents j^{th} attribute of i^{th} instance. G_i represents the ground truth of $\overrightarrow{D_i}$. The following equation represents the Logistic Regression,

$$Prob(G_i = 1 | \overrightarrow{D_i}^T \overrightarrow{W}) = \sigma(\overrightarrow{D_i}^T \overrightarrow{W}) = 1/(1 + e^{-\overrightarrow{D_i}^T \overrightarrow{W}}) \tag{5.1}$$

where, $\overrightarrow{W}$ is the weight vector and σ is the sigmoid function. While training the regression model, we learn and optimize the model parameters (weights) $\overrightarrow{W}$ from the

error. The error is the Euclidean distance between the ground truth and trained classification as follows.

$$E_i = (G_{train} - G_i)^2/2 \tag{5.2}$$

During training, we update the model parameters as follows,

$$W_i^j = W_i^j + \alpha \frac{dE}{dD_i} \tag{5.3}$$

where, α is the learning parameter, and $\frac{dE}{dD}$ is the derivative of the error (E) with respect to input instance (D_i).

Linear regression is another machine learning technique, widely studied in literature. Linear regression predicts continuous values (house rent, forecast), while the logistic regression predicts categories (class levels). However, they are very similar in terms of computations. Logistic regression uses the sigmoid ($\sigma(\overrightarrow{D_i}^T \overrightarrow{W})$) function in Equation 5.1 to convert the value between 0 and 1, while the linear regression does not use σ.

Modifications

The current TFHE framework [1] does not support floating-point numbers and division operations. Hence, we modify the regression computations and introduce the following changes.

- For a specific attribute we scale up the number by a factor of tens, to get integer values. We adapt such scaling up from [29].
- We scale up the class values by a factor of 1024. Therefore, the class labels become 0 or 1024. This also gives us a wide range of values similar to the

floating point number between 0 and 1. For the upper limit, we use 1024 to take advantage of bit manipulations.

- In Equation 5.3, we take the reciprocal of the learning rate (α) [29].
- Most importantly, we approximate the non-linear sigmoid function, as it can not be modified (details in Section 5.5).

The rest of the chapter discusses the building blocks (Lagrange Interpolation Form, MINMAX computation) for the proposed approximation techniques.

5.2 Mathematical Functions

The activation functions used in machine learning algorithms usually fall under the following categories.

a) **Non-linear Function**

Definition 5.2.1. A linear function (f) satisfies following criteria for $\forall x, y$ and $\beta \in \mathbb{R}$

$$f(x+y) = f(x) + f(y)$$

$$f(\beta x) = \beta f(x)$$

Non-linear functions, on the contrary, do not satisfy the mentioned criteria. Therefore, except the identity function ($f(x) = x$), most of the activation functions are non-linear functions.

b) **Continuous Functions**

Definition 5.2.2. A function f is said to be continuous at $x = a$ if $f(a)$ is defined and $\lim_{x \to a} f(x) = f(a)$. For example, sigmoid function,

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{5.4}$$

Other activation functions that fall in this categories are identity function and *tanh* function.

c) **Piecewise Functions**

Definition 5.2.3. A piecewise function has more than one formula to define output at different ranges, e.g., Rectified Linear Unit (ReLU) function.

$$ReLU(x) = \begin{cases} x & \text{if } x \geq 0 \\ 0 & \text{if } x < 0 \end{cases} \tag{5.5}$$

Besides ReLU, binary step function, min-max function are the well known piecewise activation functions in machine learning literature.

d) **Piecewise Linear Functions** The piecewise linear functions are a special kind of piecewise functions, where all the formula are linear functions.

5.3 Lagrange Interpolation Form

Polynomial interpolation is one of the applications of Lagrange Interpolating Polynomial. It approximates any polynomial $P(x)$ of degree $\leq (d-1)$ that passes through the d points [61].

Definition 5.3.1. For a given set S of d points,

$$S = \{(x_i, y_i) \mid i < d \wedge \forall\, i, j < d \wedge i \neq j,\ x_i \neq x_j\}$$

The Lagrange Polynomial Interpolation is defined as,

$$P(x) = \sum_{i=0}^{d-1} y_i P_i(x) \tag{5.6}$$

where,

$$P_i(x) = \prod_{\substack{j=0 \\ j \neq i}}^{d-1} \frac{x - x_j}{x_i - x_j}$$

Thus, for any d-degree non-linear function approximation, we choose $d+1$ points and compute the approximation using Equation 5.6.

In our construction, we first break the equation into pieces and approximate the non-linear parts among them to fit the equation curve. Section 5.5 discusses the piecewise approximation concept in detail.

5.4 MINMAX Computation and Comparators

Unlike other F/HE libraries, TFHE comes with an encrypted bitwise comparator: MUX. The MUX operator takes three binary inputs a, b and c, and outputs one bit (either b or c) based on the a.

$$MUX(a, b, c) = a\ ?\ b\ :\ c$$

The existing TFHE library provides an implementation of the MINMAX function for unsigned numbers. We extend the algorithm for signed MINMAX computation. Algorithm 5 presents our proposed signed MINMAX computation. We furthermore,

Algorithm 5 Minimum of two Signed Encrypted Numbers

Input: A, B $\in \mathbb{B}^n$

Output: R $\in \mathbb{B}^{n+1}$

1: C $\leftarrow$ A == B `// A XNOR B`

2: $t_0 \leftarrow 0$ `//`t_0`, a single bit`

3: **for** $i \leftarrow 0$ **to** $n-1$ **do**

4: $t_0 = c_i \; ? \; t_0 \; : \; a_i$ `//`i^{th}`-bit of C and A`

5: **end for**

6: $t_0 \leftarrow c_{n-1} \; ? \; t_0 \; : \sim t_0$ `//check for signed numbers`

7: R $\leftarrow t_0 \; ? \; B \; : \; A$

8: **return** R

incorporate two following types of vector operations. It is important to note that we implement MUX operation, MINMAX computation, along with the vector MINMAX computations employing GPU to maximize the use of hardware resources to reduce the computation time.

- **κ-output from κ-pairs:** First, we compute κ-MINMAX operations over the κ-pair of numbers in parallel. We assemble the bits that are subject to a single operation and execute them together. For example, to execute Line 1, 4, and 6 in Algorithm 5, we collate all the bits at that stages and execute them in parallel.

- **MINMAX of κ-numbers:** The second MINMAX vector operation is to find the minimum (maximum) from a vector of encrypted numbers.This operation usually requires κ-sequential comparisons. However, our construction performs

the k-comparison in $\log n$ sequential comparisons in a tree-based approach. First, we divide κ-encrypted numbers into two vectors of length $\kappa/2$ and find the vector of minimum/maximum encrypted numbers using the MINMAX operation with $\kappa/2$ output. We continue the same process of dividing and computing until we get one value. Figure 4.5 depicts similar tree-based approach. In the figure, instead of addition, we perform MINMAX operation.

5.5 Activation Function Approximation

The widely used activation functions in machine learning algorithms are non-linear, e.g., sigmoid, tanh. While approximating, we take the leverage of the comparator (available in TFHE) and the MINMAX function. In the construction, we first break the non-linear functions into piece-wise continuous functions on some interval and approximate each function separately. Below we present the proposed approximations of sigmoid and tanh as those are the most used activation functions.

5.5.1 Sigmoid Approximation

Figure 5.1 illustrates the sigmoid function that converts any input value to $[0, 1]$. From the plot, we observe that we can break the sigmoid curve into three separate continuous pieces. The ranges are *a)* $[-\infty, -4)$, *b)* $[-4, 4]$, and *c)* $(4, -\infty]$. Thus, the

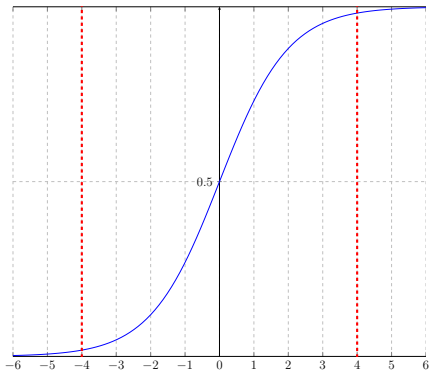


Figure 5.1: Sigmoid (σ) activation function. The dashed red lines indicates intervals of the divided segments

new piecewise sigmoid function (σ_p) from Equation 5.4 becomes,

$$\sigma_p(x) = \begin{cases} 0 & \text{if } x < -4 \\ \sigma(x) & \text{if } -4 \leq x \leq 4 \\ 1 & \text{if } x > 4 \end{cases} \tag{5.7}$$

The $\sigma_p(x)$ still has a complex construction of $\sigma(x)$ for the range $[-4, 4]$. Hence, we approximate the region using Lagrange Interpolation polynomial. For a given set of points,

$$S_\sigma = \{(x, \sigma_p(x)) \mid -4 \leq x \leq 4\}$$

we compute the interpolation polynomial in the Lagrange form for degree $1, 2$ and 3. We analyze the error for each degree of approximation in Section 6.8 and use the *Degree* 1 approximation because of comparatively equivalent error and less compu-

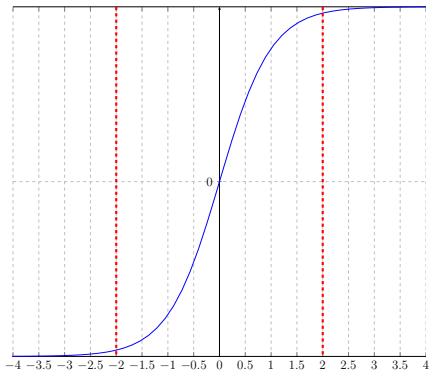


Figure 5.2: *tanh* activation function. The dashed red lines indicates intervals of the divided segments

tational depth. Therefore, the Equation 5.7 becomes,

$$\sigma_{p_L}(x) = \begin{cases} 0 & \text{if } x < -4 \\ 0.125x + 0.50 & \text{if } -4 \leq x \leq 4 \\ 1 & \text{if } x > 4 \end{cases} \tag{5.8}$$

Ciphertexts, unlike plaintext comparison, does not provide *true* (1) or, *false* (0). Hence, we need to generalize Equation 5.8 as,

$$\sigma_{p_L}(x) = min(1, max(0, x) \gg 3 + 0.5) \tag{5.9}$$

Here, $\gg$ denotes the right shift operation.

5.5.2 tanh Approximation

tanh function converts any input to $[-1, 1]$ and is defined as

$$tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{5.10}$$

Observing the plot in Figure 5.2 we divide the tanh function into three regions namely, *a)* $[-\infty, -2)$, *b)* $[-2, 2]$, and *c)* $(2, -\infty]$ from the . Thus, the tanh piecewise approximation formulation becomes,

$$tanh_p(x) = \begin{cases} -1 & \text{if } x < -2 \\ tanh(x) & \text{if } -2 \leq x \leq 2 \\ 1 & \text{if } x > 2 \end{cases} \tag{5.11}$$

Like σ_p, the approximation equation $tanh_p$ for the region [-2, 2] using Lagrange Interpolation Form becomes,

$$tanh_{p_L}(x) = \begin{cases} -1 & \text{if } x < -2 \\ x/2 & \text{if } -2 \leq x \leq 2 \\ 1 & \text{if } x > 2 \end{cases} \tag{5.12}$$

And the *tanh* approximation on ciphertexts is,

$$tanh_{p_L}(x) = min(1, max(-1, x) \gg 1) \tag{5.13}$$

5.6 Logistic Regression in GPU || framework

The logistic regression model generation (training) phase consists of dot products of each instance ($\overrightarrow{D_i}$) in the dataset and the weight vector ($\overrightarrow{W}$) (Equation 5.1). In a dot product, first, the vectors ($\overrightarrow{D_i}^T$ and $\overrightarrow{W}$) are multiplied component-wise (i.e., vector multiplication), and later, the multiplied outputs are accumulated (reduced by sum). For the component-wise multiplication, we use our proposed vector multiplication, and for the accumulation, we re-used the tree-based approach (discussed in Section 4.2.2). Figure 5.3 illustrates the parallel dot product approach.

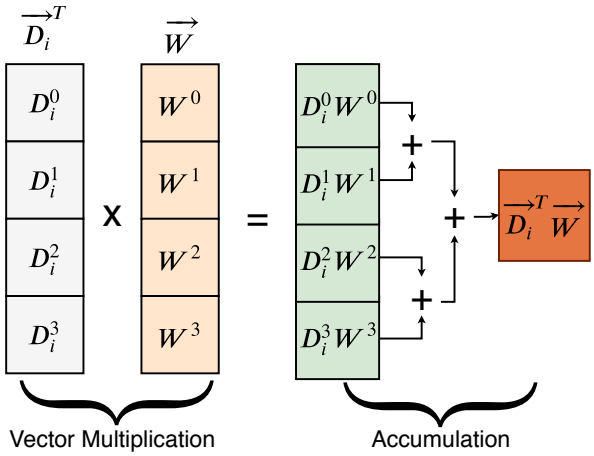


Figure 5.3: Parallel Dot Product computation ($\overrightarrow{D_i}^T \overrightarrow{W}$). The first computation is the parallel vector multiplication, and the second part is the tree-based accumulation

This computation is similar to Linear Regression techniques. However, the only difference is the sigmoid ($\sigma(\overrightarrow{D_i}^T \overrightarrow{W})$. We use the approximation technique discussed in the previous section. The current framework supports model training one instance at a time i.e., supported batch size is 1.

Chapter 6

Experimental Analysis

6.1 Comparison Metrics

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

We use two metrics for the comparison among the frameworks. One is the execution time (/wall clock) for any operations, and the other one is the speedup:

$$\text{Speedup} = \frac{T_{seq}}{T_{par}}$$

where, T_{seq} and T_{par} are the time taken to compute the sequential and the parallel algorithm, respectively. In the following discussions, we gradually analyze the complicated arithmetic circuits using the best results from the foregoing analysis.

For analyzing the approximation error, we used mean squared error (MSE). MSE

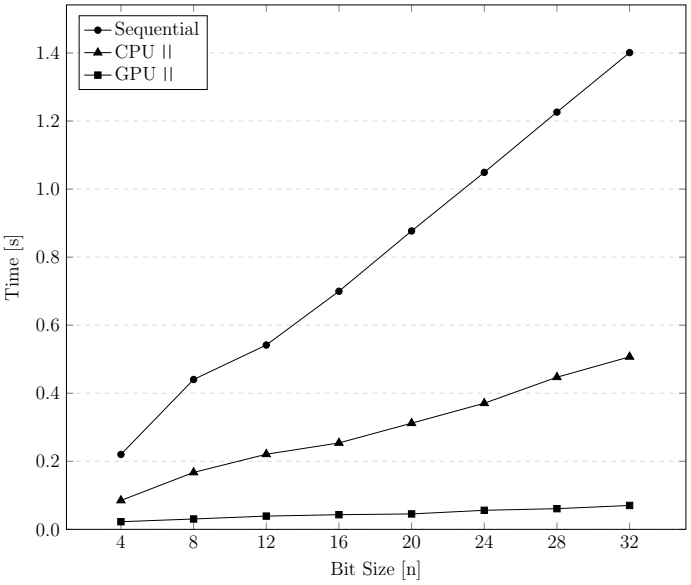


Figure 6.1: Execution time to compute n-bit boolean gate (`AND`) computation in three different frameworks

assesses the quality of approximation functions based on the mean of the squared error for each data point approximation. It is defined as,

$$MSE = \frac{1}{t}\sum_{i=0}^{t}(\hat{\theta} - \theta)^2$$

Here, $\hat{\theta}$, θ, and t are approximate value, ideal value, and sample size respectively.

6.2 GPU-accelerated TFHE

First, we discuss our performance over boolean gate operations, which are the building blocks of any computation under encryption. Figure 6.1 depicts the execution time difference between the sequential, CPU || and GPU || framework for $4, \ldots, 32$-bits. The sequential `AND` operation takes a minimum of 0.22s (4-bit), and the runtime linearly increases to 1.4s for 32-bits.

Table 6.1: Analysis of time (in milliseconds) taken by Bootstrapping, Key Switching and other operations for sequential and GPU framework w.r.t. different bit sizes (n)

	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

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 parallelism is available in the CPU || framework as we divide the number of bits to 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 in stead of CPU.

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 these three as they are the most time-consuming operations and fairly generalizable to other HE schemes. Table 6.1 shows the difference in execution time between the sequential and the GPU || for $\{2, 4 \ldots, 32\}$-bits.

We further investigated the performance of bootstrapping in GPU || while performing the gate operations. Our FFT library, cuFFT operates in batches for incorporating parallelism. However, the number of batches to be executed in parallel is limited, and solely depends on the hardware specification. It can operate a certain number of batches at once and other batches are kept in a queue. Hence, a sequential overhead is incurred for a large number of batches that eventually increases the execution time.

Under the same environmental settings, we benchmark the existing GPU-frameworks of TFHE, cuFHE, and NuFHE. Although the GPU || framework outperforms NuFHE for different bit sizes (Figure 6.2), the performance degrades for larger bit sizes with re-

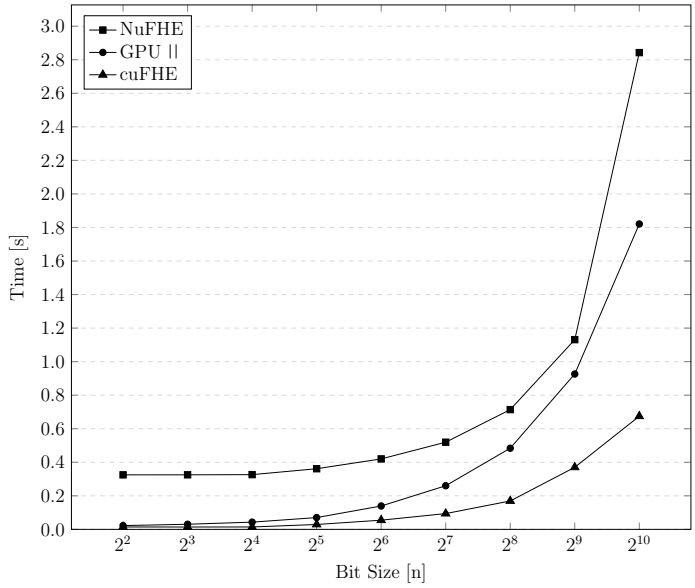


Figure 6.2: Execution time comparison for n-bit boolean gate (`AND`) with existing GPU-accelerated frameworks

spect to cuFHE. cuFHE implementation focuses more on the gate level optimization, while our focus lies on arithmetic circuit computations. Later, we provide performance analysis on the arithmetic circuits, where our framework outruns the existing ones.

6.3 Compound Gate Analysis

We utilized compound gates to improve the execution time for additions or multiplications as described in Section 4.2.1. Since, the other existing frameworks do not provide such optimization techniques (for arithmetic circuits), we benchmark with our single gate and compound gate computations. Figure 6.3 illustrates the performance of one compound gate over its counterpart, 2-single gates sequential computation.

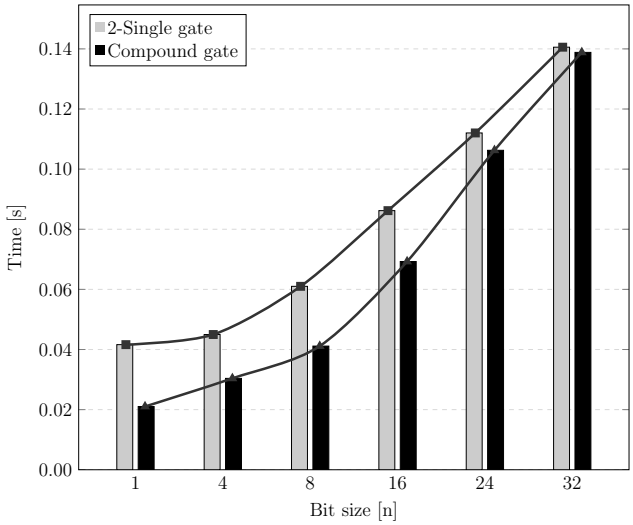


Figure 6.3: Performance of compound gates against 2-single gate operations for different bit sizes (n)

We performed several operations for different the number of bits $(1, 4, \ldots, 32)$ on the X-axis, and Y-axis represents the execution time. It is noteworthy that a 32-bit compound gate will have two 32-bit inputs and output two 32-bits as well. Here, bit coalescing is the reason behind the improvement as it takes only 0.02s for one compound gate, compared to 0.04s on performing 2-Single gates sequentially. However, Figure 6.3 shows an interesting trend in execution time between 2-single gate and a compound gate computation. The gap favoring the compound gates tends to get narrower for a higher number of bits. For example, the speedup for 1-bit happens to be $\dfrac{0.04}{0.02} = 2$. However, this speedup reduces to 1.01 for 32-bits. The reason behind this diminishing performance gap is the asynchronous launch queue of GPU.

As mentioned in Section 4.2.1, we use batch execution of cuFFT. The number of batches to execute in parallel depends on the asynchronous launch queue of GPU resulting in delaying FFT for large number batches (for larger bit sizes), ultimately

Table 6.2: Execution time (in seconds) required for adding two n-bit numbers, where GPU_n and GPU_1 represent the number-wise addition and bitwise addition, respectively

Frameworks	**16-bit**	**24-bit**	**32-bit**
Sequential	3.51	5.23	7.04
CPU ||	3.51	5.23	7.04
GPU_n||	0.94	2.55	4.44
GPU_1||	0.98	1.47	1.99
cuFHE	1.00	1.51	2.03
NuFHE	2.92	3.56	4.16
Cingulata	1.10	1.63	2.16

affecting the speedup. Nevertheless, the analysis shows that the 1-bit compound gate is the most efficient, and we employ in the following operations.

6.4 Addition

Table 6.2 presents a comparative analysis of the execution times (in seconds) for the addition operation of 16, 24, 32-bit encrypted numbers. In the analysis, we consider our three frameworks: sequential (Section 2.4.1), CPU || (Section 4.1.1), and GPU || (Section 4.2.2) and benchmark with cuFHE, NuFHE and Cingulata. Besides, we discuss the performance of two GPU-based addition operations for GPU: GPU_n|| (number-wise addition) and GPU_1|| (bit by bit addition).

Table 6.2 demonstrates that $\text{GPU}_n||$ performs better than the sequential and CPU $||$ circuit. The GPU_n provides a 3.72× speedup for 16-bits whereas for 32-bit it is 1.58×. However, with respect to $\text{GPU}_1||$, $\text{GPU}_n||$ performs better only for 16-bit addition, and declines for larger bits. For 24 and 32-bit addition, $\text{GPU}_1||$ performs around 2× better than $\text{GPU}_n||$. This is an important observation as it reveals which algorithm to choose between $\text{GPU}_1||$ and $\text{GPU}_n||$.

Although, both the addition operations ($\text{GPU}_n||$ and $\text{GPU}_1||$) utilize compound gate, they differ in the number of input bit sizes (n for $\text{GPU}_n||$ and 1 for $\text{GPU}_1||$). Since the compound gate performs better for smaller number of bits (Section 6.3), the bitwise addition performs better than the number-wise addition for 24/32-bit operations. Hence, we utilize bitwise addition for building other circuits.

cuFHE and NuFHE do not provide arithmetic circuits in their library. Therefore, we implemented addition circuits on their work and carried out experiments on them. Additionally, we considered Cingulata (a compiler toolchain for computing over encrypted data using TFHE) and compared the execution time. Table 6.2 summarizes all the results, where we found our proposed addition circuit ($\text{GPU}_1||$) outperforming others.

We further experimented on the vector addition operation adopting the bitwise addition and demonstrate the runtime analysis in Table 6.3. Analogous to the regular additions, the performance improvement on our 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 larger vectors as they involve more parallel bit computations, and consequently, increase the batch size of cuFFT operations.

Table 6.3: Execution time (in seconds) taken to add vectors of different length (ℓ)

	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

The difference is demonstrated on 32-bit vector additions with $\ell = 32$ which takes almost twice the time from $\ell = 16$. However, for $\ell \leq 8$, the executions times do not indicate such trend for both 16 or 32-bits.

In Section 6.3 we have discussed this issue and Figure 6.3 also aligns with evidence as we found out that the larger batch size for FFT on GPUs adversely effects the speedup, e.g.$\ell = 32$ will reqiure more FFT batches compared to $\ell = 16$ as we get a slight increase in execution time. We did not include other frameworks in Table 6.3, since the GPU || performed better comparing to the others in Table 6.2.

6.5 Multiplication

The multiplication operation uses a sequential accumulation (reduce by addition) operation. Instead, we use a tree-based vector addition approach (discussed in Section 4.2.2) and gain a significant speedup. Table 6.4 portrays the execution times for the multiplication operations using the frameworks.

Table 6.4: Naive and Karatsuba Multiplication runtime comparison (in seconds) for $16, 24$, and 32-bit numbers with existing frameworks

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	32.75	74.21	132.23
NuFHE	47.72	105.48	186.00
Cingulata	11.50	27.04	50.69
Karatsuba			
CPU	54.76	-	177.04
GPU ||	7.6708	-	24.62

For CPU || multiplications, 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.

Like the addition analysis, we implemented the multiplication circuit on cuFHE and NuFHE as well. Table 6.4 summarizes the result and comparison with respect to cuFHE, NuFHE, and Cingulata. The result provides evidence of the outstanding performance of our GPU || framework. It is important to note that the performance increase with the increase in the number of bits. The reason behind the improvement

Table 6.5: Execution time (in minutes) taken to multiply vectors of different length (ℓ)

	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

is the tree-based addition technique for the reduction operation and computing all the gate operations by coalescing the bits together.

Similar to the vector additions, we analyze vector multiplications available in our framework and present a comparison among the frameworks in Table 6.5. 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 as discussed in Section 6.4. Therefore, 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 existing framework surpassed their achievements for a single multiplications.

Table 6.6: Matrix multiplication execution time (in minutes)

Dimension	**Sequential**	**CPU ||**	**GPU ||**
2×2	17.07	10.62	0.86
4×4	136.68	47.78	5.90
8×8	1090.12	351.82	43.95
16×16	8717.89	2514.34	186.23

6.6 Karatsuba Multiplication

In Table 6.4, 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. Notably, Karatsuba multiplication can also be considered a complex and compound 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.

6.7 Matrix operations

From Section 4.2.2, it is evident that the vector addition represents matrix additions as both operations are done point-wise. Therefore, Table 6.3 can be extended to represent the execution time for the matrix additions, where ℓ becomes the number of

elements of the matrices. Table 6.6 enlists the matrix multiplication execution time for different dimensions using Algorithm 4. For a 16×16 matrix, GPU || achieves a $\approx 48\times$ and $\approx 15\times$ speedup compared to the sequential and CPU || approach, respectively.

6.8 Approximation Error

6.8.1 Sigmoid Approximation

The thesis discusses the approximation of non-linear activation functions using piecewise and Lagrange polynomial approximation (Section 5.5). In addition to the existing approach of Hesamifard *et al.* , we consider Taylor [62] and Bernstein [63] polynomial approximations as well. The experiment covers the domain $[-1000, 1000]$ and uses MSE for the error analysis. Here, the reference for the error calculation is the sigmoid function.

Table 6.7 exhibits a noticeable improvement of the proposed piecewise approximation comparing with [56] and Taylor approximation. Bernstein polynomial, on the other hand, incurs similar error to the Lagrange, except it possesses a larger computation depth.

We also analyze the approximation error incurred by the piecewise Lagrange polynomial for degrees $1, 2$ and 3 (Table 6.7). In terms of MSE, the order is *Degree* 3, *Degree* 1, and *Degree* 2. On the contrary, the computational depth of *Degree*1 is the lowest (one shift and one addition). Therefore, we choose the *Degree*1 form for sigmoid approximation.

Table 6.7: Sigmoid Approximation comparison on different approximation models. The experimental domain includes all the data points in $[-1000, 1000]$. The error metric is Mean Square Error (MSE)

Name	**Deg.**	**Polynomial**	**MSE**
Proposed Approximations (Piecewise)			
Lagrange	1	$0.125x + 0.50$	3.47e-05
	2	$0.02x^2 + 0.18x + 0.432$	1.58e-04
	3	$0.21x - 0.006x^3 + 0.5$	1.73e-06
Taylor	3	$- 0.0005x^3 + 0.25x + 0.5$	4.05e-04
Bernstein	3	$x^3/(exp(-1)+1) - (x-1)^3/2 +$ $(3x(x-1)^2)/(exp(-1/3)+1) -$ $(3x^2(t-1))/(exp(-2/3)+1)$	6.60e-05
Hesamifard *et al.* Approximation [20]			
Chebyshev	3	$-(8.60e-10)x^3 + (1.33e-17)x^2 + 0.001x + 0.499$	0.06
Method 1	5	$(2.07e-15)x^5 - (5.32e-23)x^4 + 0.001x + 0.499$	0.72
Chebyshev	3	$-(4.80e-10)x^3 - (7.08e-16)x^2 + 0.0009x + 0.5$	0.05
Method 2	5	$(6.65e-16)x^5 + (9.37e-21)x^4 + 0.001 * x + 0.5$	0.20

6.8.2 *tanh* Approximation

Table 6.8 presents an error analysis of the piecewise Lagrange approximation for *tanh*. We considered the approximation polynomials up to *Degree* 3. The order of the polynomials for the approximation error is *Degree* 1, *Degree* 2,and *Degree* 3.

Table 6.8: Comparison of Lagrange polynomial approximation for *tanh* function for different degrees

Deg.	**Polynomial**	**MSE**
1	$0.5x$	$6.97e-05$
2	$0.16x^2 + 0.74x - 0.13$	$5.35e-04$
3	$0.09x^3 - 0.85x$	0.0061

Since the order for the computational depth is the same as well, we use the *Degree* 1 polynomial for *tanh* approximation.

6.9 Logistic Regression

We use Parkinson's Disease data [28] to experiments on our logistic regression framework. The dataset consists of 195 observations, with 22 attributes for each observation. We compare our proposed linear and logistic regression with the proposed linear regression proposed in [29] and summarize the results in Table 6.9. The result portrays the superlative performance of FV-based CPU-parallel linear regression performance. The TFHE-based computation could not beat even using GPU, for binary data as well. The reason is the bootstrapping time at each gate operation in TFHE. Nevertheless, since TFHE can perform arbitrary depths of computations, such schemes are beneficial for heavily computing algorithms like Deep Learning.

Table 6.9: Regression Analysis comparison between FV and TFHE GPU || frameworks

Models	**Ciphertext Size (MB)**	**Execution Time (min)**	
		Numerical Attributes	**Binary Attributes**
Linear Regression (FV CPU ||)	587.90	10.01	10.01
Linear Regression (TFHE GPU||)	5.99	184.95	42.66
Logistic Regression (TFHE GPU||)	5.99	189.24	46.77

6.10 Summary of Experimental Results

The experimental analysis of the frameworks on arithmetic operations can be summarized as follows:

- Our proposed GPU || framework outperforms existing ones in arithmetic computation. However, it lags in gate level operations in comparison with cuFHE.
- By integrating the comparison circuit, we reduce the approximation error compared to the existing techniques.
- Finally, GPU-accelerated TFHE, as expected, could not beat the performance of Somewhat/leveled HE due to the bootstrapping operation.

Chapter 7

Conclusion

This chapter first summarizes the thesis and then discusses some frequently asked questions. Finally, it concludes with future directions.

7.1 Summary

In this thesis, 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 around 20 times faster than the existing technique for computing boolean gates and multiplications (Table 1.2).

7.2 Discussion

How does the CP/GPU hardware affect the performance?

The hardware specification plays a vital role in the execution time of the proposed frameworks. CP/GPUs with more cores and higher clock speeds are supposed to provide better speedups as they vary in floating point operation per cycle, memory size and access speeds. However, our experiments include machines with a modest configuration ($<$ 700\$ GPU).

Is the proposed framework sufficient to implement any machine learning model?

In this thesis, 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. Further investigations on complicated usecases such as *secure machine learning* [64, 65] were excluded as they can utilize our framework for computing encrypted data. Note that we have implemented a fully homomorphic encryption scheme. Hence, any user-defined computable function can be implemented using our framework.

How to achieve application level parallelism with the proposed framework?

We acknowledge that there are always scopes of application level parallelism which does not require parallel circuit constructions. For example, we can distribute the rows and columns of a matrix for parallel multiplications and discard the performance improvement from individual multiplications itself. However, our proposed frameworks do not discard but adheres such parallelism as it conjuncts any computations as long as they fit the H/W memory.

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.

Can we utilize the computation capacity of CPUs and GPUs concurrently?

In our approach, CPU and GPU || frameworks exhaust their resources (CP/GPU) separately and sit idle unless mentioned otherwise. Taking advantage of both resources can be an interesting future direction as it depends on the application. For example, with task-level parallelism in Figure 2.1, CPU computes Task 1 while GPU computes Task 2. Here, CP/GPU that finishes first needs to wait for the other before proceeding to Task 3. However, such waiting time will increase the overall execution time and needs to be proportionate for the optimal performance. It can also be employed for the aforementioned memory issue as we can split the computations between CPU and GPU for 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 (see Appendix for details). 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 [66] 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.

7.3 Future Works

The following are some interesting directions for future works.

- Adapting deep learning algorithms is an extension of the current framework. Moreover, the computational depth requirement of deep learning algorithms matches the features of TFHE. Therefore, we plan to implement deep learning algorithms using the proposed framework.

- The use of multiple GPUs can reduce the execution time in some cases, e.g., large vector and matrix operations. We can distribute the input among GPUs and collect the results at the end of computations. Such shared computing might introduce a delay for communications and waiting in the execution time.

Thus, for concurrent computations, we intend to introduce multiple GPUs and analyze the performance.

Bibliography

[1] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. Faster fully homomorphic encryption: Bootstrapping in less than 0.1 seconds. In *Advances in Cryptology–ASIACRYPT 2016: 22nd International Conference on the Theory and Application of Cryptology and Information Security, Hanoi, Vietnam, December 4-8, 2016, Proceedings, Part I 22*, pages 3–33. Springer, 2016.

[2] Jung Hee Cheon, Kyoohyung Han, Andrey Kim, Miran Kim, and Yongsoo Song. Bootstrapping for approximate homomorphic encryption. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 360–384. Springer, 2018.

[3] Microsoft SEAL (release 3.2). `https://github.com/Microsoft/SEAL`, February 2019. Microsoft Research, Redmond, WA.

[4] Anh Pham, Italo Dacosta, Guillaume Endignoux, Juan Ramon Troncoso Pastoriza, Kevin Huguenin, and Jean-Pierre Hubaux. Oride: A privacy-preserving yet accountable ride-hailing service. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 1235–1252, Vancouver, BC, 2017.

[5] Miran Kim, Yongsoo Song, and Jung Hee Cheon. Secure searching of biomark-

ers through hybrid homomorphic encryption scheme. *BMC medical genomics*, 10(2):42, 2017.

[6] Hao Chen, Ran Gilad-Bachrach, Kyoohyung Han, Zhicong Huang, Amir Jalali, Kim Laine, and Kristin Lauter. Logistic regression over encrypted data from fully homomorphic encryption. *BMC medical genomics*, 11(4):81, 2018.

[7] Google Prediction API. https://cloud.google.com/prediction/, Accessed on 21 May, 2019.

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

[9] GraphLab. https://turi.com/, Accessed on 21 May, 2019.

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

[11] Ronald L Rivest, Adi Shamir, and Leonard Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, 1978.

[12] Oded Regev. On lattices, learning with errors, random linear codes, and cryptography. *Journal of the ACM (JACM)*, 56(6):34, 2009.

[13] Pascal Paillier et al. Public-key cryptosystems based on composite degree residuosity classes. In *Eurocrypt*, volume 99, pages 223–238. Springer, 1999.

[14] Chris Lomont. Introduction to intel advanced vector extensions. *Intel White Paper*, pages 1–21, 2011.

[15] CUDA-accelerated Fully Homomorphic Encryption Library, September 2019.

[16] NuFHE, a GPU-powered Torus FHE implementation, September 2019.

[17] Cingulata, September 2019.

[18] Md Nazmus Sadat, Al Aziz, Md Momin, Noman Mohammed, Feng Chen, Xiaoqian Jiang, and Shuang Wang. Safety: secure gwas in federated environment through a hybrid solution. *IEEE/ACM Transactions on Computational Biology and Bioinformatics (TCBB)*, 16(1):93–102, 2019.

[19] Chiraag Juvekar, Vinod Vaikuntanathan, and Anantha Chandrakasan. GAZELLE: A low latency framework for secure neural network inference. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 1651–1669, 2018.

[20] Ehsan Hesamifard, Hassan Takabi, Mehdi Ghasemi, and Rebecca N Wright. Privacy-preserving machine learning as a service. *Proceedings on Privacy Enhancing Technologies*, 2018(3):123–142, 2018.

[21] Craig Gentry et al. Fully homomorphic encryption using ideal lattices. In *STOC*, volume 9, pages 169–178, 2009.

[22] Craig Gentry, Shai Halevi, and Nigel P Smart. Better bootstrapping in fully homomorphic encryption. In *International Workshop on Public Key Cryptography*, pages 1–16. Springer, 2012.

[23] Zvika Brakerski, Craig Gentry, and Shai Halevi. Packed ciphertexts in lwe-based homomorphic encryption. In *International Workshop on Public Key Cryptography*, pages 1–13. Springer, 2013.

[24] Jacob Alperin-Sheriff and Chris Peikert. Faster bootstrapping with polynomial error. In *International Cryptology Conference*, pages 297–314. Springer, 2014.

[25] Lo Ducas and Daniele Micciancio. Fhew: Bootstrapping homomorphic encryption in less than a second. Cryptology ePrint Archive, Report 2014/816, 2014. `https://eprint.iacr.org/2014/816`.

[26] Junfeng Fan and Frederik Vercauteren. Somewhat practical fully homomorphic encryption. *IACR Cryptology ePrint Archive*, 2012:144, 2012.

[27] Shai Halevi and Victor Shoup. Algorithms in helib. In *Annual Cryptology Conference*, pages 554–571. Springer, 2014.

[28] Max A Little, Patrick E McSharry, Stephen J Roberts, Declan AE Costello, and Irene M Moroz. Exploiting nonlinear recurrence and fractal scaling properties for voice disorder detection. *Biomedical engineering online*, 6(1):23, 2007.

[29] Toufique Morshed, Dima Alhadidi, and Noman Mohammed. Parallel linear regression on encrypted data. In *2018 16th Annual Conference on Privacy, Security and Trust (PST)*, pages 1–5. IEEE, 2018.

[30] Christina Boura, Nicolas Gama, and Mariya Georgieva. Chimera: a unified framework for b/fv, tfhe and heaan fully homomorphic encryption and predictions for deep learning. Technical report, Cryptology ePrint Archive, Report 2018/758, 2018.

[31] Matteo Frigo and Steven 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), volume 3, pages 1381–1384. IEEE, 1998.

[32] Ahmed Salem, Pascal Berrang, Mathias Humbert, and Michael Backes. Privacy-preserving similar patient queries for combined biomedical data. *Proceedings on Privacy Enhancing Technologies*, 2019(1):47–67, 2019.

[33] NVIDIA. Geforce gtx 1080 graphics cards from nvidia geforce.

[34] Michael J Flynn. Some computer organizations and their effectiveness. *IEEE transactions on computers*, 100(9):948–960, 1972.

[35] Fermi NVidia. Nvidias next generation cuda compute architecture. *NVidia, Santa Clara, Calif, USA*, 2009.

[36] Catherine C. McGeoch. Parallel addition. *The American Mathematical Monthly*, 100(9):867–871, 1993.

[37] M Lehman, D Senzig, and J Lee. Serial arithmetic techniques. In *Proceedings of the November 30–December 1, 1965, fall joint computer conference, part I*, pages 715–725. ACM, 1965.

[38] Anatolii Alekseevich Karatsuba and Yu P Ofman. Multiplication of many-digital numbers by automatic computers. In *Doklady Akademii Nauk*, volume 145, pages 293–294. Russian Academy of Sciences, 1962.

[39] Martin Albrecht, Melissa Chase, Hao Chen, Jintai Ding, Shafi Goldwasser, Sergey Gorbunov, Shai Halevi, Jeffrey Hoffstein, Kim Laine, Kristin Lauter, Satya Lokam, Daniele Micciancio, Dustin Moody, Travis Morrison, Amit Sahai,

and Vinod Vaikuntanathan. Homomorphic encryption security standard. Technical report, HomomorphicEncryption.org, Toronto, Canada, November 2018.

[40] Frederik Armknecht, Colin Boyd, Christopher Carr, Kristian Gjøsteen, Angela Jäschke, Christian A Reuter, and Martin Strand. A guide to fully homomorphic encryption. *IACR Cryptology ePrint Archive*, 2015:1192, 2015.

[41] Ciara Moore, Máire O'Neill, Elizabeth O'Sullivan, Yarkın Doröz, and Berk Sunar. Practical homomorphic encryption: A survey. In *2014 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 2792–2795. IEEE, 2014.

[42] Craig Gentry, Amit Sahai, and Brent Waters. Homomorphic encryption from learning with errors: Conceptually-simpler, asymptotically-faster, attribute-based. In *Advances in Cryptology–CRYPTO 2013*, pages 75–92. Springer, 2013.

[43] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. Faster packed homomorphic operations and efficient circuit bootstrapping for tfhe. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 377–408. Springer, 2017.

[44] Zvika Brakerski and Vinod Vaikuntanathan. Efficient fully homomorphic encryption from (standard) lwe. *SIAM Journal on Computing*, 43(2):831–871, 2014.

[45] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. (leveled) fully homomorphic encryption without bootstrapping. *ACM Transactions on Computation Theory (TOCT)*, 6(3):13, 2014.

[46] Jung Hee Cheon, Andrey Kim, Miran Kim, and Yongsoo Song. Homomorphic encryption for arithmetic of approximate numbers. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 409–437. Springer, 2017.

[47] Yarkın Doröz, Erdinç Öztürk, Erkay Savaş, and Berk Sunar. Accelerating ltv based homomorphic encryption in reconfigurable hardware. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 185–204. Springer, 2015.

[48] Adriana López-Alt, Eran Tromer, and Vinod 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*, pages 1219–1234. ACM, 2012.

[49] Wei Dai and Berk Sunar. cuhe: A homomorphic encryption accelerator library. In *International Conference on Cryptography and Information Security in the Balkans*, pages 169–186. Springer, 2015.

[50] Wei Dai, Yarkın Doröz, and Berk Sunar. Accelerating swhe based pirs using gpus. In *International Conference on Financial Cryptography and Data Security*, pages 160–171. Springer, 2015.

[51] Arnold Schönhage and Volker Strassen. Schnelle multiplikation grosser zahlen. *Computing*, 7(3-4):281–292, 1971.

[52] C. Orlandi, A. Piva, and M. Barni. Oblivious neural network computing

via homomorphic encryption. *EURASIP Journal on Information Security*, 2007(1):037343, Jul 2007.

[53] Bita Darvish Rouhani, M Sadegh Riazi, and Farinaz Koushanfar. Deepsecure: Scalable provably-secure deep learning. In *Proceedings of the 55th Annual Design Automation Conference*, page 2. ACM, 2018.

[54] Andrew Chi-Chih Yao. Protocols for secure computations. In *FOCS*, volume 82, pages 160–164, 1982.

[55] Ran Gilad-Bachrach, Nathan Dowlin, Kim Laine, Kristin Lauter, Michael Naehrig, and John Wernsing. Cryptonets: Applying neural networks to encrypted data with high throughput and accuracy. In *International Conference on Machine Learning*, pages 201–210, 2016.

[56] Ehsan Hesamifard, Hassan Takabi, and Mehdi Ghasemi. Cryptodl: Deep neural networks over encrypted data. *arXiv preprint arXiv:1711.05189*, 2017.

[57] Weimin Han and Kendall E Atkinson. *Theoretical Numerical Analysis: A Functional Analysis Framework*. Springer, 2009.

[58] Rohit Chandra, Leo Dagum, David Kohr, Ramesh Menon, Dror Maydan, and Jeff McDonald. *Parallel programming in OpenMP*. Morgan kaufmann, 2001.

[59] Lynn Elliot Cannon. *A cellular computer to implement the Kalman filter algorithm*. PhD thesis, Montana State University-Bozeman, College of Engineering, 1969.

[60] Max A Little, Patrick E McSharry, Eric J Hunter, Jennifer Spielman, Lorraine O Ramig, et al. Suitability of dysphonia measurements for telemonitoring of parkinson's disease. *IEEE transactions on biomedical engineering*, 56(4):1015–1022, 2009.

[61] Edward Waring. Vii. problems concerning interpolations. *Philosophical transactions of the royal society of London*, (69):59–67, 1779.

[62] Brook Taylor. *Methodus incrementorum directa & inversa*. Inny, 1717.

[63] Philip J Davis. *Interpolation and approximation*. Courier Corporation, 1975.

[64] Pengtao Xie, Misha Bilenko, Tom Finley, Ran Gilad-Bachrach, Kristin Lauter, and Michael Naehrig. Crypto-nets: Neural networks over encrypted data. *arXiv preprint arXiv:1412.6181*, 2014.

[65] Hassan Takabi, Ehsan Hesamifard, and Mehdi Ghasemi. Privacy preserving multi-party machine learning with homomorphic encryption. In *29th Annual Conference on Neural Information Processing Systems (NIPS)*, 2016.

[66] Ilaria Chillotti Snowman Nicolas Gama, SC. Tfhe: Fast fully homomorphic encryption library over the torus. `https://github.com/tfhe/tfhe/tree/v1.0.1`, 2017.