information

*Article*

# Secure Genomic String Search with Parallel Homomorphic Encryption

**Md Momin Al Aziz *, Md Toufique Morshed Tamal and Noman Mohammed**

Department of Computer Science, University of Manitoba, Winnipeg , MB R3T 5V6, Canada; morshed@cs.umanitoba.ca (M.T.M.T.); noman@cs.umanitoba.ca(N.M.)
* Correspondence: azizmma@cs.umanitoba.ca

**Abstract:** Fully homomorphic encryption (FHE) cryptographic systems enable limitless computations over encrypted data, providing solutions to many of today's data security problems. While effective FHE platforms can address modern data security concerns in unsecure environments, the extended execution time for these platforms hinders their broader application. This project aims to enhance FHE systems through an efficient parallel framework, specifically building upon the existing torus FHE (TFHE) system chillotti2016faster. The TFHE system was chosen for its superior bootstrapping computations and precise results for countless Boolean gate evaluations, such as AND and XOR. Our first approach was to expand upon the gate operations within the current system, shifting towards algebraic circuits, and using graphics processing units (GPUs) to manage cryptographic operations in parallel. Then, we implemented this GPU-parallel FHE framework into a needed genomic data operation, specifically string search. We utilized popular string distance metrics (hamming distance, edit distance, set maximal matches) to ascertain the disparities between multiple genomic sequences in a secure context with all data and operations occurring under encryption. Our experimental data revealed that our GPU implementation vastly outperforms the former method, providing a 20-fold speedup for any 32-bit Boolean operation and a 14.5-fold increase for multiplications.This paper introduces unique enhancements to existing FHE cryptographic systems using GPUs and additional algorithms to quicken fundamental computations. Looking ahead, the presented framework can be further developed to accommodate more complex, real-world applications.

**Keywords:** fully homomorphic encryption; GPU parallel operations; secure computation on GPU; parallel FHE framework; secure string search using FHE



**Citation:** Aziz, M.M.A.; Tamal, M.T.M.; Mohammed, N. Secure Genomic String Search with Parallel Homomorphic Encryption. *Information* **2024**, *15*, 40. https://doi.org/10.3390/info15010040

Academic Editors: Jose de Vasconcelos, Hugo Barbosa and Carla Cordeiro

Received: 8 November 2023
Revised: 28 December 2023
Accepted: 29 December 2023
Published: 11 January 2024



## 1. Introduction

In recent times, the study of fully homomorphic encryption (FHE) [1] has been a significant area of cryptographic research. FHE cryptosystems, renowned for their robust security assurances, are characterized by their ability to carry out numerous operations on encrypted data. As there is a growing demand for data-oriented applications designed to manage confidential human data, the notion of computing within the scope of encryption emerges as increasingly prevalent [2–4]. As such, FHE presents a perfect cryptographic solution to these privacy issues by facilitating arbitrary operations on encrypted data within an untrusted computational setting.

Despite the security provided by the cryptosystem, FHE's performance speed is a drawback for routine computations, leading to its limited adoption and scant real-world applications. For instance, a roughly 7-s duration is required to add two 32-bit encrypted numbers, while multiplication operations are notably slower, taking about 8 min (Table 1). To encourage the broader utilization of FHE in real-world applications, improvements to its speed are imperative—this can be achieved either through theoretical techniques to cut back on computational complexity or through simultaneous operations.

Our study suggests a parallel framework for executing FHE computations using graphics processing units (GPUs). Over the years, GPU technology has significantly contributed to various machine learning algorithms by expediting model training processes over extensive datasets. Following a similar approach, we tap into the multicore features of GPUs and propose a parallel FHE framework that uses the torus FHE (TFHE) cryptosystem [5].

Our goal is to apply the proposed FHE operations to genomic data and assess the framework's efficiency. Previously, when genomic data were processed in plain text without any protective measures, numerous safety issues surfaced [6,7]. Therefore, encrypting data during storage or computation should enhance security in case of a data breach or compromised system. Furthermore, we plan to enhance the proposed framework to include three common string search functions: hamming distance, edit distance, and set-maximal matches. These search functions hold crucial significance for applications such as ancestry search [8] and similar patients query [9,10], often involving confidential personal data. The effectiveness of these search operations is critically important, illustrated by their recent use in solving a high-profile crime case, the 'Golden State Killer' [11].

*Contributions*

Our study can be principally divided into two phases: (a) the development of a parallel fully homomorphic encryption (FHE) computation framework and (b) the execution of string search operations using our proposed framework. We outline our chief contributions below:

- We primarily expand Boolean gates (i.e., XOR, AND, etc.) from an existing FHE framework [12] to secure algebraic circuits comprising addition and multiplication.
- Taking full advantage of the latest enhancements in GPU architecture, we introduce parallel FHE operations. We further propose several enhancement methods, like bit coalescing, compound gates, and tree-based additions, for the execution of the secure algebraic circuits.
- We conducted a series of experiments to contrast the execution time of the sequential TFHE [12] with our proposed GPU parallel framework. Data from Table 1 demonstrate that our proposed GPU parallel method is 14.4 and 46.81 times quicker than the existing technique for standard and matrix multiplications, respectively. We also compared our performance with existing GPU-based TFHE frameworks, such as cuFHE [13], NuFHE [14], and Cingulata [15].
- Lastly, we focused on different string search operations in the genomic dataset (hamming distance, edit distance, and set-maximal matches) and executed them under encryption. Experimental outcomes reveal that the framework requires approximately 12 min to execute hamming distance and set-maximal matching on two genomic sequences with 128 genomes. In addition, for 8 genomes, the framework takes 11 min for an edit distance operation, significantly improving from the previous 5 h attempt by Cheon et al. [16].

**Table 1.** Comparison of the execution times (seconds) of our CPU and GPU framework for 32-bit numbers with TFHE [17], cuFHE [13], NuFHE [14], and Cingulata [15] (vector/matrix length of 32).

| | Gate Op. | Addition | | Multiplication | |
|---|---|---|---|---|---|
| | | Regular | Vector | Regular | Matrix (min) |
| GPU-parallel | 0.07 | 1.99 | 11.22 | 33.93 | 186.23 |
| CPU-parallel | 0.50 | 7.04 | 77.18 | 174.54 | 2514.34 |
| TFHE [12] | 1.40 | 7.04 | 224.31 | 489.93 | 8717.89 |
| cuFHE [13] | - | 2.03 | - | 132.23 | - |
| NuFHE [14] | - | 4.16 | - | 186 | - |
| Cingulata [15] | - | 2.16 | - | 50.69 | - |

In this work, we extend our previous work [18] on a CPU–GPU-parallel FHE framework. Notably, existing GPU-enabled TFHE libraries, cuFHE [13] and NuFHE [14], have implemented TFHE Boolean gates using GPUs, whereas our goal was to construct an optimized arithmetic circuit framework. Our design choices and algorithms reflect this improvement, and as a result, our multiplications are around 3.9 and 4.5 times faster than cuFHE and NuFHE, respectively. The code is readily available at https://github.com/UofM-DSP/CPU-GPU-TFHE (accessed on 4 January 2024).

The rest of this work is organized as follows: We discuss the required background of the work in Section 2. Section 3 discusses the underlying methods including the GPU-parallel framework and the string search operations using such parallel operations. In Section 4, we show the experimental analysis, whereas Section 5 discusses it in detail. Section 6 presents the related works, and finally, this work is concluded in Section 7.

## 2. Background

In this section, we describe the employed cryptographic scheme, TFHE [12], and later define the string search problem.

### *2.1. Torus FHE (TFHE)*

In this study, we utilize torus fully homomorphic encryption (TFHE) [12], where plaintexts and ciphertexts are defined over a real torus, $\mathbb{T} = \mathbb{R}/\mathbb{Z}$, a set of real numbers called modulo 1. Ciphertexts are built through learning with errors (LWE) [19] and are expressed as torus LWE (TLWE). Here, an error term, taken from a Gaussian distribution, $\chi$, is integrated into each ciphertext. When we consider a key size (dimension) of $m \geq 1$, a secret key present in an $m$-bit binary vector, and an error part of the chi distribution, an LWE sample is denoted as $(\mathbf{a}, b)$. Here, $\mathbf{a}$ signifies a vector of torus coefficients of length $m$ (key size), with each element $\mathbf{a}_i$ derived from the uniform distribution over the real torus, and $b = \mathbf{a} \cdot \mathbf{s} + e$.

The error term ($e$) in the LWE sample increases and proliferates with the number of operations (for instance, addition, multiplication). Therefore, the bootstrapping technique is used to decrypt and renew the ciphertext's encryption to mitigate the noise.

TFHE views binary bits as plaintext, producing LWE samples as ciphertexts. Consequently, computations of LWE samples ($\mathbb{L}^n$) in ciphertext are equivalent to binary bit computations in plaintext. By translating binary vector representations of integer numbers into LWE sample vectors, we can represent encrypted integers. For instance, an encrypted version of an $n$-bit integer would be an $n$-LWE sample. Hence, the operations of a binary addition circuit between two $n$-bit numbers can parallel the equivalent operations on LWE samples of encrypted integers. In this paper, we use the terms *bit* and *LWE sample* interchangeably, and we select TFHE for the following reasons:

- Fast and Exact Bootstrapping: TFHE provides the fastest exact bootstrapping requiring around 0.1 s. Some recent encryption schemes [20,21] also propose faster bootstrapping and homomorphic computations in general. However, they do not perform exact bootstrapping and are erroneous after successive computations on the same ciphertexts.
- Ciphertext Size: Compared with the other HE schemes, TFHE offers a smaller ciphertext size as it operates on binary plaintexts (only 32 kb compared to 8mb for one 32-bit number). Nevertheless, this minimal storage advantage allows us to utilize the limited and fixed memory of GPU when we optimize the gate structures.
- Boolean Operations: TFHE also supports Boolean operations that can be extended to construct arbitrary functions. These binary bits can then be operated in parallel if their computations are independent of each other.

*Current Approach:* The existing application of TFHE includes foundational cryptographic functions like encryption, decryption, and all binary gate operations [12]. Notably, despite the somewhat sequential calculation of gates in the original application, the base architecture employs advanced vector extensions (AVXs) [22]. AVX, an enhancement to Intel's

x86 instruction set, supports parallel vector operations. The bootstrapping process calls for substantial fast Fourier transform (FFT) operations that grow in complexity in O(n log n). The current model uses the Fastest Fourier Transform in the West (FFTW) [23], which inherently incorporates AVX.

*Why TFHE?* Several attempts have been made to enhance the performance and numerical operations of FHE [24–26], which are critical to our work (refer to Section 6 for details). Among the most prominent FHE schemes, torus FHE (TFHE) successfully delivers an arbitrary depth of circuits with a faster bootstrapping technique. Furthermore, it demands less storage compared with other encryption models (Table 8, available in related works for comparison). With TFHE, the plaintext message space is binary, which means that computations are entirely based on Boolean gates. Every gate operation necessitates a bootstrapping procedure in gate bootstrapping mode.

*Why GPU?* Most FHE schemes build upon the learning with errors (LWE) principle. In this context, plaintexts encrypted with polynomials can be portrayed using vectors. Hence, most calculations operate on vectors, making them highly parallelizable. In contrast, graphics processing units (GPUs) offer a vast number of computing cores, more so than CPUs. These cores can therefore effectively compute parallel vectors operations. Thus, these cores can be employed to parallelize FHE computations. It is crucial, however, to take into account the fixed and limited memory capacity of GPUs (8–16 GB) and their relative computing power compared with a CPU core. For in-depth comparisons, readers may refer to the Appendix A of this paper.

### *2.2. Sequential Framework*

In this section, we present a brief overview of the sequential arithmetic circuit constructions using Boolean gates as background, which we extend later.

#### 2.2.1. Addition

A carry-ahead 1-bit full adder circuit takes two input bits along with a carry to compute the sum and a new carry that propagates to the next bit's addition. Therefore, in a full adder, we have three inputs as $a_i$, $b_i$, and $c_{i-1}$, where $i$ denotes the bit position. Here, the addition of bit $a_1$ and $b_1$ in $A, B \in \mathbb{B}^n$ requires the carry bit from $a_0$ and $b_0$. This dependency enforces the addition operation to be sequential for $n$-bit numbers [27]. In this work, we also used half adders for numeric increments and decrements.

#### 2.2.2. Multiplication

Naive Approach

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

Karatsuba Algorithm

We consider the divide-and-conquer Karatsuba algorithm for its improved time complexity $O(n^{\log^3})$ [28]. It relies on dividing the large input numbers and performing smaller multiplications. For $n$-bit inputs, the Karatsuba algorithm splits them into smaller numbers of $n/2$-bit size and replaces the multiplication with additions and subsequent multiplications (Line 12 of Algorithm 1). Later, we introduce parallel vector operations for further optimizations.

**Algorithm 1:** Karatsuba Multiplication [28]

**Input:** $X, Y \in \mathbb{B}^n$
**Output:** $Z \in \mathbb{B}^{2n}$
**if** $n < n_0$ **then**
  **return** `BaseMultiplication(X, Y)`
**end**
$X_0 \leftarrow X \mod 2^{n/2}$
$Y_0 \leftarrow Y \mod 2^{n/2}$
$X_1 \leftarrow X/2^{n/2}$
$Y_1 \leftarrow Y/2^{n/2}$
$Z_0 \leftarrow$ `KaratsubaMultiply`$(X_0, Y_0)$
$Z_1 \leftarrow$ `KaratsubaMultiply`$(X_1, Y_1)$
$Z_2 \leftarrow$ `KaratsubaMultiply`$(X_0 + Y_0, X_1 + Y_1)$
**return** $Z_0 + (Z_2 - Z_1 - Z_0)2^n + (Z_1)2^{2n}$

### *2.3. CPU-Based Parallel Framework*

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

#### 2.3.1. Addition

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

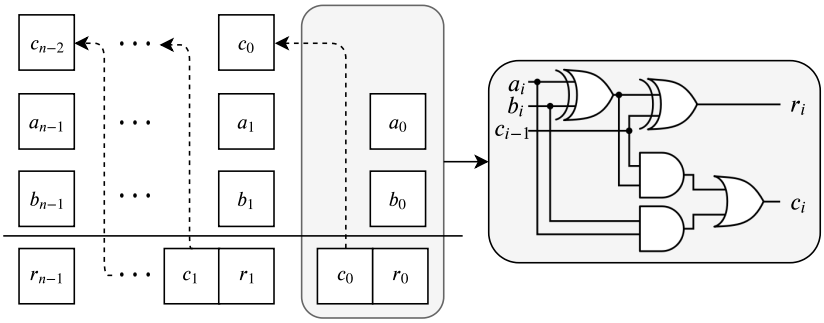



**Figure 1.** Bitwise addition of two $n$-bit numbers, $A$ and $B$. $a_i$, $b_i$, $c_i$, and $r_i$ are the $i$th-bit of $A$, $B$, *carry*, and the *result*.

Here, it is possible to exploit task-level parallelism where two threads execute the `XOR` and `AND` operations (Figure 1) simultaneously. We observed that the time required to perform such `fork-and-join` between two threads is higher than that when executing them serially. This is partially due to the costly thread operations and eventual serial dependency of the results. Hence, we did not employ this technique for CPUs.

#### 2.3.2. Multiplication

Among the three key operations, `AND`, `left shift`, and `accumulation` (addition), used in multiplications, `AND` and `left shift` can be carried out concurrently. For instance, when we have two 16-bit numbers, A and B ($\in \mathbb{B}^{16}$), and four functional threads, we distribute the `AND` and `left shift` operations among these threads.

Conversely, the accumulation operation is more demanding as it requires the execution of $n$ additions for $n$-bit multiplication. This operation of accumulating values necessitates adding and storing values to the same variable, thereby rendering the operation atomic. As a result, all threads that were engaged in performing the previous `AND` and `left shift` operations must wait for the accumulation to complete. This is known as global thread

synchronization [29]. However, due to its computational expensiveness, we avoid using this approach in any parallel framework.

Instead, we adopted a customized reduction operation within OpenMP [29], which exploits the globally shared memory (CPU) to store interim results. This approach predicts the addition of any results upon completion, thus avoiding the need for global thread synchronization and ultimately enhancing performance. We found that this custom reduction greatly improved performance when compared with the traditional approach of waiting for threads to complete their tasks, otherwise known as global thread synchronization.

### *2.4. String Search: Problem Definition*

We are proposing privacy-preserving methods to measure string distances using hamming, edit distance, and set-maximal matching. We define the query string as $q$, whereas the target genomic sequence is denoted as $y$. For simplicity, we assume that all sequences have an equal number of $m$ genes, where each gene is biallelic. Biallelic genes are represented as $\{0,1\}$, resulting in a query to be a bit vector where $q = [q_1, q_2, \ldots, q_m]$ as $q_i \in \{0,1\}$. On the other hand, any target sequence is defined as $y = [y_1, y_2, \ldots, y_m]$ as $y_i \in \{0,1\}$.

In this problem, the query $q$ and data $y$ are encrypted with a fully homomorphic encryption (FHE) scheme [12]. Upon encryption, we denote the query as a vector of encrypted bits and is represented with $\mathbf{q}$. The encrypted data $\mathbf{y}$ is hosted in a cloud environment where a researcher is sending his/her encrypted query. Notably, the target can be a set of genomic sequences, denoted as $\mathbf{Y}$ . The target is to exactly calculate or approximate a string distance score for $\mathbf{q}$ against $\mathbf{y}$ under FHE with a certain algorithm, such as hamming or edit distance. Since it is an asymmetric encryption scheme, we assume that the cloud server only has access to the public key. On the other hand, the researcher and data owner have the private key to decrypt the result and encrypt the genomic data, respectively. The targeted string distance metrics are formally defined below:

**Definition 1** (*Hamming distance*)**.** *The hamming distance* $hd(\mathbf{q}, \mathbf{y})$ *measures the difference or number of genes that are different in two sequences,* $\boldsymbol{q}$ *and* $\mathbf{y_i}$*:* $hd(\boldsymbol{q}, \mathbf{y}) = \sum_{k \in [1,m]} (\boldsymbol{q}[k] \neq \mathbf{y}[k])$.

**Definition 2** (*Edit distance*)**.** *The edit distance* $ed(\mathbf{q}, \mathbf{y})$ *between two sequences* $(\mathbf{q}, \mathbf{y})$ *is defined as the minimum cost taken over all edit sequences that transform query* $\mathbf{q}$ *into* $\mathbf{y}$*. That is,* $ed(\mathbf{x}, \mathbf{y}) = min\{C(s) | s$ *is a sequence of edit operations (insert, update, or delete) transforming* $\mathbf{q}$ *into* $\mathbf{y}\}$.

**Definition 3** (*Set-maximal distance*)**.** *A set-maximal score or distance* $sd(\mathbf{q}, \mathbf{y})$ *denotes the maximum number of consecutive matching genes between* $\mathbf{q}$ *and* $\mathbf{y}$*, which have the following conditions:*

1. *There exists some index* $k_2 > k_1$ *such that* $\boldsymbol{q}[k_1, k_2] = \mathbf{y}[k_1, k_2]$ *(same substring);*
2. $\boldsymbol{q}[k_1 - 1, k_2] \neq \mathbf{y_i}[k_1 - 1, j_2]$ *and* $\mathbf{q}[k_1, k_2 + 1] \neq \mathbf{y}[j_1, j_2 + 1]$*; and*
3. *For all other genes,* $k' \neq k$ *and* $k' \in [1, m]$*, if there exist* $k'_2 > k'_1$ - $\mathbf{q}[k'_1, k'_2] = \mathbf{y}[k'_1, k'_2]$ *then it must be* $k'_2 - k'_1 < k_2 - k_1$.

*The set-maximal distance is defined as* $sd(\mathbf{q}, \mathbf{y}) = k_2 - k_1$.

## 3. Methods

In this section, we outline our proposed solutions to compute the string distance metrics for the targeted algorithms. First, we propose the GPU-parallel FHE framework on top of which we build the string search operations described later.

### *3.1. GPU-Based Parallel Framework*

In this section, we present three generalized techniques to introduce GPU parallelism (GPU ||) for any FHE computations. Then, we adopt them to implement and optimize the arithmetic operations. Notably, our CPU-parallel (CPU ||) framework is also described in Section 2.3.

#### 3.1.1. Proposed Techniques for Parallel HE Operations

This section introduces general techniques adopted for the GPU-based parallel framework.

Parallel TFHE Construction

Figure 2 refers to the depiction of Boolean circuit computation. In this computation, every LWE sample is composed of two elements: **a** and $b$. **a** is a 32-bit integer vector determined by the secret key size ($m$), and its memory needs are less than those in other FHE implementations (Section 6). In this parallel TFHE construction, only the vector **a** is stored in the GPU's global memory.

Moreover, this setup uses the native CUDA-enabled FFT library (cuFFT). This library employs the parallel CUDA cores for FFT operations, with a batching technique that allows for many FFT operations to be carried out concurrently. However, the same cuFFT tool also sets a limit to the parallel number of batches. It arranges these batches in an asynchronous launch queue and processes a specific number of these batches simultaneously. This number is strictly dependent on the hardware capacity and specifications [30].

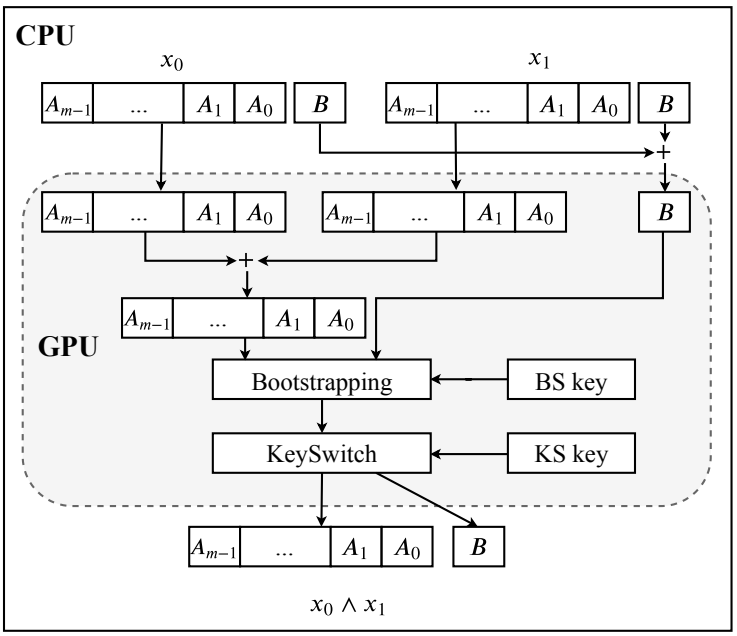



**Figure 2.** Arbitrary operation between two bits where BS and KS key represent bootstrapping and key switching keys, respectively.

Bit Coalescing (BC)

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

The main idea of such a structure is to boost parallel processes by extending the length of the vector in sequential memory. While the length of the vector is increased through vector coalescing, we incorporate additional threads to optimize parallelization and decrease execution time.

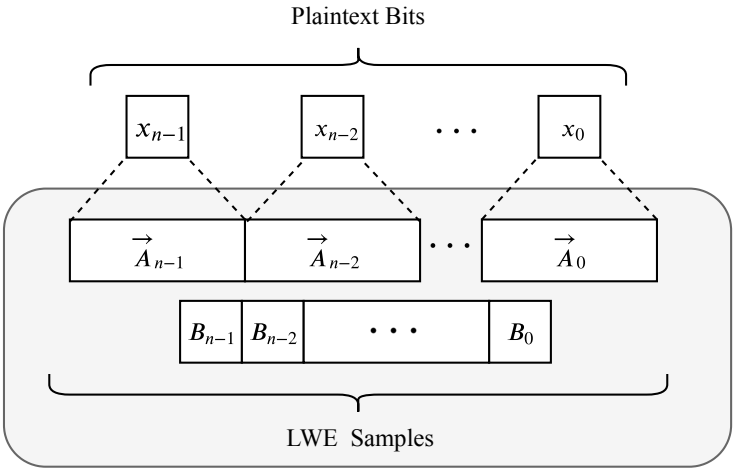



**Figure 3.** Coalescing $n$-LWE samples (ciphertexts) for $n$-bits where $\mathbf{A}_i$ vectors are contiguously located in GPU memory. $B$ values for all $n$ bits are also located togethar as the GPU memory is marked in gray.

Compound Gate

As addition is a crucial component in many arithmetic circuits, we suggest a unique gate structure known as *compound gates*. These gates provide more opportunities for parallel processes among encrypted bits. They constitute a blend of two gates and work similarly to an ordinary Boolean gate, accepting two 1-bit inputs but delivering two distinct outputs. This innovative gate structure's inspiration comes from the addition circuit. For the sum $R = A + B$, we calculate $r_i$ and $c_i$ using the given equations.

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

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

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

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

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

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

3.1.2. Algebraic Circuits on GPU

This section presents different algebraic circuit constructions in a GPU-based parallel framework using the general techniques.

**Addition:** *Bitwise addition ($GPU_1$):* From the addition circuit in Section 2.3.1, 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 gates' construction. Hence, we incorporated the compound gates to construct the bitwise addition circuit. We also implemented the

vector addition circuits using $GPU_1$ to support complex circuits, such as multiplications (Section 2.2.2).

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

Here, we utilize compound gates to perform $R \wedge B$ and $R \oplus B$ in parallel. Thus, in each iteration, the input becomes two $n$-bit numbers, while in bitwise computation, the input was two single bits. On the contrary, even after using compound gates, the bitwise addition (Equations (1) and (2)) has more sequential blocks (3) than the numberwise addition (0). We analyze both in Section 4.3.

*Numeric increments and decrements:* We also propose half adders for numeric increments and decrements required for several operations in string search. For example, in Algorithms 3–5, we need to perform increments and decrements of an encrypted number. We use half adders and subtractors to perform the operations. In Figure 4, we show the difference of the operations. For the half adder, we perform the XOR and AND operations for all input bits for the encrypted number, while the other input is set to 1 (or 0) under encryption. The only difference for the half subtractor is that the input bit is inverted before the AND operation, which represents the carry bit.


(**a**) Half Adder (**b**) Half Subtractor

**Figure 4.** One-bit increment and decrement using the half adder or subtractor where $\mathbf{x}_i$ is the input bit and the carry bit is propagated into the next bit's operation.

The sign bit for these encrypted numbers (most significant bit) also goes through the same operation as the rest of the bits. However, in this work, we cannot protect the increment against overflow as the number of bits for each encrypted number is set prior to the execution. For example, if we are incrementing a 16-bit encrypted number, and it obtains a value of $2^{15} + 1$ (1 bit reserved for the sign bit), it will not obtain a correct decrypted value. On the other hand, while decrementing by 1 for Algorithms 4 and 5, we will eventually get into negative numbers, represented by the sign bit. Therefore, we perform an OR operation in Algorithm 2 on Line 10.

**Algorithm 2:** Determine if input number is greater than zero

**Input:** Encrypted number **x** with $|\mathbf{x}|$ bits, Boolean flag *hasSign* if **x** has sign bit
**Output:** One bit representing whether **x** is greater than 0, *result*

```
2  Procedure greaterThanZero(x,hasSign)
3      i ← 0
4      result ← E(0)
5      while i < |x| − 1 do
6          result ← result OR x[i]
7          i ← i + 1
8      end
9      if hasSign then
10         result ← result AND (NOT x[|x|])
11     end
12     return result
```

**Multiplication**

*Naive Approach:* According to Section 2.2.2, multiplications have $\wedge$ and $\ll$ operations that can be executed in parallel. It will result in $n$-numbers where each number will have $[n, 2n]$-bits due to $\ll$. We need to accumulate these uneven-sized numbers, which cannot be distributed among the GPU threads. Furthermore, the addition presents another sequential bottleneck while adding and storing $(+=)$ the results in the same memory location. Therefore, this serial addition will increase the execution time. In the framework, we optimize the operation by introducing a tree-based approach.

In this approach, we divide $n$-numbers (LWE vectors) into two $n/2$ vectors. These two $n/2$ vectors are added in parallel. We repeat the process as we divide the resultant vectors into two $n/4$ vectors and add them in parallel. The process continues until we obtain the final result. Notably, the tree-based approach requires $\log n$ steps for the accumulation. In Figure 5 for $n = 8$, all the ciphertexts underwent $\wedge$ and $\ll$ in parallel, and waited for addition. Here, $L_{ij}$ represents the LWE samples (encrypted numbers), $i$ is the level, and $j$ denotes the position.

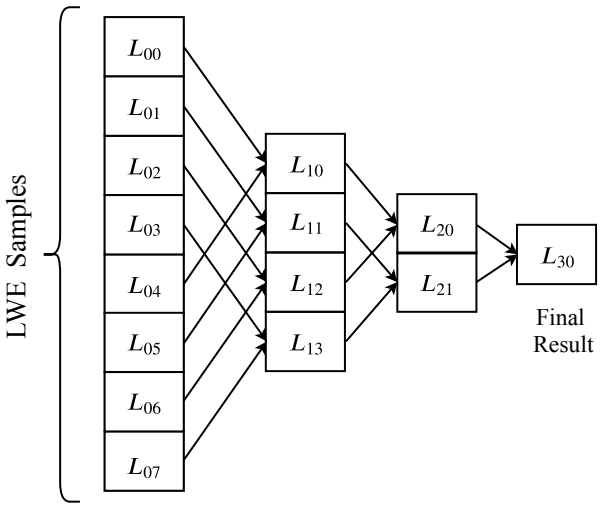



**Figure 5.** Accumulating $n = 8$ LWE samples ($L_{ij}$) in parallel using a tree-based reduction.

*Karatsuba multiplication:* We used the Karatsuba algorithm with some modifications in our framework to achieve further efficiency while performing multiplications. However, this algorithm requires both addition and multiplication vector operations, which tested the efficacy of these components as well. We modified the original Algorithm 1 to introduce the vector operations and rewrite the computations in Lines 9–12 as

$$\begin{aligned} \langle Temp_0,\ Temp_1 \rangle &= \langle X_0,\ X_1 \rangle + \langle Y_0,\ Y_1 \rangle \\ \langle Z_0,\ Z_1,\ Z_2 \rangle &= \langle X_0,\ X_1,\ Temp_0 \rangle \cdot \langle Y_0,\ Y_1,\ Temp_1 \rangle \\ \langle Temp_0,\ Temp_1 \rangle &= \langle Z_2,\ Z_1 \rangle + \langle 1,\ Z_0 \rangle \\ Z_2 &= Temp_0 + (Temp_1)' \end{aligned}$$

In the above equations, $X_0$, $X_1$, $Y_0$, $Y_1$, $Z_0$, $Z_1$, and $Z_2$ are taken from the algorithm, and $\langle \ldots \rangle$ and $\cdot$ are used to denote concatenated vectors and dot product, respectively. For example, in the first equation, $Temp_0$ and $Temp_1$ store the addition of $X_0$, $Y_0$ and $X_1$, $Y_1$. It is noteworthy that in the CPU || framework, we utilized task-level parallelism to perform these vector operations as described in Section 2.3.

3.1.3. Bitwise Operations

In this section, the general bitwise operations required for determining the string distances are discussed. These algorithms will inherit the aforementioned algorithms and extend them accordingly based on the corresponding use cases:

Greater Than Zero

Our string distance methods on encrypted data rely on Algorithm 2 to check whether $input > 0$. Here, the algorithm takes an encrypted number as an input and checks whether it is greater than zero. This allows us to judge whether there are any set bits on the encrypted version of the number. In order to output that result, in Line 6, an encrypted bitwise OR ($\bar{OR}$) operation is performed between an encrypted bit $X[i]$ and the current *result*.

The final result also considers the sign bit as the number can be negative. Here, the sign bit is set as the most significant bit (MSB) or $X[|X|]$, which is inverted and placed on another OR operation with the *result* variable. Determining whether the input is less than 0 can also be achieved by this bit. Notably, the value of the *result* is kept encrypted throughout the computations, which is utilized in the upcoming algorithms.

Longest Consecutive Ones

Algorithm 3 is specifically designed to find the most extended series of consecutive 1 value bits in an encrypted number or bit stream. The encrypted number input, denoted as $X$, is left shifted in each repetition until we exhaust the bit stream, which happens after $|X|$ repetitions.

The crucial operation of this method occurs on Line 6, where $X$ undergoes a left shift by a single unit. After this shift, an encrypted bitwise $A\bar{N}D$ operation is also performed with the preceding value of $X$. One check is then performed to see if the newly created $X$ contains any 1 bit (or whether $X > 0$), and if so, a counter is incremented. This counter is linked to the *result* variable, as indicated in Algorithm 2, and increments by one each time $X > 0$.

It is crucial to underline that this algorithm is particularly applicable for set-maximal distance calculations, where encrypted haplotypes are used as the $X$ input. This further notes that `greaterThanZero` accounts for the sign bit, which is not essential in set-maximal operations. As such, Line 10 is not considered in this specific context.

**Algorithm 3:** Find longest consecutive ones

**Input:** Encrypted number $\mathbf{x}$
**Output:** *result* representing the number of the longest consecutive ones
2 **Procedure** `maxConsecutiveOnes(`$X$`)`
3 $numbits \leftarrow |\mathbf{x}|$
4 $result \leftarrow$ `greaterThanZero`$(\mathbf{x}, false)$
5 **while** $numbits > 0$ **do**
6 $\mathbf{x} \leftarrow \mathbf{x}$ A$\bar{\text{N}}$D $(\mathbf{x} \ll 1)$
7 $result \leftarrow result +$ `greaterThanZero`$(\mathbf{x}, false)$
8 $numbits \leftarrow numbits - 1$
9 **end**
10 **return** *result*

Let us assume that we have an encrypted number $\mathbf{x} = \mathcal{E}(011101)$ with the sign bit $\mathcal{E}(0)$, and it contains 3 consecutive ones. Here, the result bit is set to 1 since $\mathbf{x} > 0$. In the first iteration, we perform an encrypted AND operation of $\mathbf{x}$ (011101) and $\mathbf{x} << 1$ (111010). Since the resulting $\mathbf{x}$ is greater than 0, the encrypted *result* number is incremented. In the following iteration, $\mathbf{x} = \mathcal{E}(011000)$ is multiplied (AND) with $\mathcal{E}(110000)$, which results in 010000. The result number is incremented again. However, in the subsequent iterations ($|\mathbf{x}|$ many times), the $\mathbf{x}$ values are set to 0 and result is not incremented anymore. Finally, the result from Algorithm 3 is retrieved as $\mathcal{E}(3)$.

Finding Minimum and Maximum Number

The pair of Algorithms 4 and 5 employs a method to target and identify the smallest and biggest numbers among a set of $n$ numbers, respectively. Each encrypted number in the set, denoted as $\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n$, undergoes a decrement operation for every bit ($|\mathbf{x}_i|$ bit size).

The algorithms then scan to determine whether the processed number arrived at zero or if it still holds any set bit. In the process to locate the smallest number, the algorithm increments the encrypted result variable only if all numbers are beyond zero. On the contrary, to find the maximum number, the algorithm utilizes an encrypted $\bar{OR}$ operation to verify if even a single number is greater than zero.

**Algorithm 4:** Get minimum number among $\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n$ encrypted positive numbers ($x_i \geq 0$)

**Input:** Positive Numbers $\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n$
**Output:** Minimum encrypted number *result*
**Procedure** `getMin`$(\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n)$
  $numinter \leftarrow 2^{|\mathbf{x}_0|}$
  $result \leftarrow 0$
  **while** $numinter > 0$ **do**
    $gtZero \leftarrow \mathcal{E}(1)$
    **foreach** $x_i \in \{x_1, \ldots, x_n\}$ **do**
      $gtZero \leftarrow gtZero$ $\bar{\text{AND}}$ `greaterThanZero`$(\mathbf{x}_i)$
      $\mathbf{x}_i \leftarrow \mathbf{x}_i - 1$
    **end**
    $result \leftarrow result + gtZero$
    $numinter \leftarrow numinter - 1$
  **end**
  **return** *result*

**Algorithm 5:** Get maximum number among $\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n$ encrypted positive numbers $x_i \geq 0$

**Input:** Positive numbers $\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n$
**Output:** Maximum encrypted number *result*
**Procedure** `getMax`$(\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n)$
  $numiter \leftarrow 2^{|\mathbf{x}_i|}$
  $result \leftarrow \mathcal{E}(0)$
  **while** $numiter > 0$ **do**
    $gtZero \leftarrow \mathcal{E}(0)$
    **foreach** $x_i \in \{x_1, \ldots, x_n\}$ **do**
      $gtZero \leftarrow gtZero$ $\bar{\text{OR}}$ `greaterThanZero`$(\mathbf{x}_i)$
      $\mathbf{x}_i \leftarrow \mathbf{x}_i - 1$
    **end**
    $result \leftarrow result + gtZero$
    $numiter \leftarrow numiter - 1$
  **end**
  **return** *result*

In both of these algorithms, a numerical decrement process is included, represented as $\mathbf{x}_i \leftarrow \mathbf{x}_i - 1$. A binary half subtractor is employed for this task. However, these decrement operations may induce an underflow, considering that the input numbers xi could turn negative in any given iteration. To deal with this, we use an operation in Algorithm 2 having MSB as `greaterThanZero`$(\mathbf{x}_i)$, which emits a single bit, indicating $\mathbf{x}_i > 0$. For identifying the smallest or largest among $n$ numbers, this single bit (denoted as *gtZero*) is added to the result for all *numbits* instances by Algorithms 4 and 5. The final result is under encryption and utilized in edit distance approximation (Section 3.2.2) and set-maximal matches (Section 3.2.3).

Alternative Approach

As an alternative strategy, full adders can also be utilized to derive the smallest or largest numbers. For instance, deducing the maximum between $x$ and $y$ can be attained by calculating $x - y$. If the sign bit is unset, it implies that $x$ is larger. Similarly, the smallest between two numbers can also be discerned by evaluating the sign bit. This method is weighed up due to the potential exponential number of iterations, considering the total bit count in Line 3 within Algorithms 5 and 4. Considering that $|x| > 16$ would necessitate numerous rounds of computations that are under encryption, for the case of $|x| \leq 16$, we resort to the previously discussed algorithms.

### *3.2. Secure String Search Operations*

In this section, we discuss the string search operations over encrypted data utilizing the earlier algorithms.

#### 3.2.1. Hamming Distance

Hamming distance $hd(\mathbf{q}, \mathbf{y})$ represents the bitwise difference of the input query $\mathbf{q}$ and stored sequence $\mathbf{y}$. Therefore, we perform an encrypted XŌR operation between $\mathbf{q}$ and $\mathbf{y}$ where the result will have set bits (value of 1) on all occasions of mismatches. Now, we need to perform a summation of all these bits on the XOR result to obtain the hamming distance (Definition 1). Notably, we assume that the query $\mathbf{q}$ and the encrypted sequences are of the same length.

In Algorithm 6, we outline the mechanism to generate the hamming distance *hd*, where it contains an encrypted distance value for one target sequence $\mathbf{y}$ and query $\mathbf{q}$. This can be iterated through all sequences and performs the XOR operation between the query $\mathbf{q}$ and $\mathbf{y}_i$ sequence. Subsequently, it also adds the bits to formulate the hamming distance in Line 5. Since the *result* variable is under encryption, the addition (or increment) is oblivious as we perform the operation for every encrypted bit in *result*.

**Algorithm 6:** Hamming distances between a query and encrypted sequences

**Input:** Encrypted target sequence $\mathbf{y}$ and query $\mathbf{q}$
**Output:** Encrypted distances between $\mathbf{y}$ and $\mathbf{q}$, $\mathbf{hd}$
2 **Procedure** `HammingDistance(`$\mathbf{q}, \mathbf{y}$`)`
3 $result \leftarrow \mathbf{q}$ XŌR $\mathbf{y}$
4 **foreach** *bit* $r \in result$ **do**
5 $\mathbf{hd} \leftarrow \mathbf{hd} + r$
6 **end**
7 **return** $\mathbf{hd}$

#### 3.2.2. Edit Distance Approximation

Edit distance is more complicated than hamming distance as it considers more than the bitwise difference (insertion, deletion, and subtraction). Furthermore, under plaintext, it has a $O$ ($m^2$) complexity, where $m$ is the length of the sequence. Therefore, to reduce the complexity, we opt for the banded edit distance [10,31], where we only compute on a band of fixed size.

Algorithm 7 outlines the proposed method, where we set a fixed parameter $b$ along with the encrypted input sequences $\mathbf{q}$ and $\mathbf{y}_i$. Apart from the initialization, we also calculate the variables *low* and *high* dictating the number of expensive operations in Line 11. Here, we calculate whether the $\mathbf{q}[i]$ and $\mathbf{y}_i[k]$ bits are the same or not using an encrypted XNOR gate. If they are not the same, then the encrypted number $d[i-1, k-1]$ needs to be incremented, which is performed with the half adder. Since we do not know the output of *same_bit*, we push that bit as carry and initialize the substitution variable. Similarly, the insertion and deletion values are set from the existing distance matrix. Finally, we calculate the minimum `getMin`$(ins, del, sub)$ to predict the distance at that specific position. This

is set as the new value of $d[i,k]$. Here, the three half-adder operations are run in parallel before the minimum operation.

**Algorithm 7:** Banded edit distance on encrypted sequence

**Data:** query $\mathbf{q}$, sequence $\mathbf{y}$, and band length $b$
**Result:** $b$-banded Edit Distance $d(\mathbf{q},\mathbf{y})$ [31]

$m \leftarrow |\mathbf{q}| + 1$
set each element of matrix $d_{m\times m}$ to $\mathcal{E}(0)$
**for** $i \leftarrow 1$ **to** $m$ **do**
    $d[i,0] \leftarrow \mathcal{E}(i)$;
    $d[0,i] \leftarrow \mathcal{E}(i)$;
**end**
**for** $i \leftarrow 1$ **to** $m$ **do**
    **if** $i - b < 1$ **then** $low \leftarrow 1$;
    **else** $low \leftarrow i - b$;
    **if** $i + b > m$ **then** $high \leftarrow m$;
    **else** $high \leftarrow i + b$;
    **for** $k \leftarrow$ $low$ **to** $high$ **do**
        $same\_bit \leftarrow \mathbf{q}[i-1]$ XNOR $\mathbf{y}[k-1]$
        $sub \leftarrow d[i-1,k-1] + same\_bit$
        $ins \leftarrow d[i,k-1] + 1$
        $del \leftarrow d[i-1,k] + 1$
        $d[i,k] \leftarrow \texttt{getMin}(sub, ins, del)$
    **end**
**end**
**return** $d[m,m]$;

#### 3.2.3. Set-Maximal Distance

The set-maximal distance or match (SMM) represents the length of the longest matching substring in two sequences [32]. This allows a health-care researcher to identify genomic sequences that have more genes in common and probably are identical in their physical attributes. The distance also has applications over similar patient queries [9], secure positional Burrows–Wheeler transformation [33,34], etc.

The proposed secure set-maximal match using homomorphic encryption operation depends on Algorithm 3, `maxConsecutiveOnes`. Initially, we perform an encrypted XNOR between two sequences, $\mathbf{y}_i$ and query $\mathbf{q}$. Here, the XNOR operation (NOT XOR) sets a value of 1 to the positions where the sequences are matching. Now, from this XNOR result, we can perform the `maxConsecutiveOnes` algorithm and obtain the highest number of set bits that are grouped together.

Suppose for a query $\mathbf{q} = 01100111$ and some input sequence $\mathbf{y}_i = 10000110$, where $\mathbf{y}_i \in \mathbf{Y}$, then $\mathbf{q}$ XNOR $\mathbf{y}_i$ will be 00011110. Now, if we perform $\texttt{maxConsecutiveOnes}(\mathbf{q}$ XNOR $\mathbf{y}_i, false)$, then the output should provide us with the encrypted result of 3. This result denotes the number of set bits on the encrypted XNOR operation, hence the set-maximal distance between $\mathbf{q}$ and $\mathbf{y}_i$.

##### Threshold SMM

In a threshold version of this match, we need to output only the distances beyond an input threshold $t$. Here, an extra operation proceeding the `maxConsecutiveOnes` is required, where a simple numeric comparison with threshold $t$ would output the result. Therefore, we can use an encrypted MUX operation [5] for this comparison. However, encrypted MUX is an expensive operation, and we can replace it with a subtraction. Therefore, we negate the $\mathcal{E}(t)$ value from the resulting $\texttt{maxConsecutiveOnes}(\mathbf{q}$ XNOR $\mathbf{y}_i, false)$. Then,

we use the `greaterThanZero` algorithm on the result, which represents if the SMM distance is beyond the threshold $t$.

We outline the algorithm in Algorithm 8, where the threshold value is encrypted at first. The matching bits of the query and the sequence are calculated next with the XNOR operation. Subsequently, we perform the comparison operation with a maximum between *result* and $\mathcal{E}(t)$. If $\mathcal{E}(t) \geq result$, then *gt_threshold* is set as the *OR* of all bits among the **smm_distance** XN̄OR **enc_t** bits. Here, XNOR represents whether two vectors are the same or not and performing another logical OR among them. Lastly, we perform the AND operation with the distance. If the value of *gt_threshold* is 0, then we obtain all unset bits on the output, with set-maximal distance in the other case.

**Algorithm 8:** Thresholded set-maximal matching

**Input:** Encrypted query $\mathbf{q}$, encrypted sequence $\mathbf{y}_i \in \{\mathbf{y}_1, \ldots, \mathbf{y}_n\}$ and threshold $t$
**Output:** Encrypted SMM distance between $\mathbf{q}$ and $\mathbf{y}_i$ if it is greater than some value $t$

**Procedure** `SMMDistance`$(\mathbf{q}, \mathbf{y}_i, t)$
  **enc_t** $\leftarrow \mathcal{E}(t)$
  **result** $\leftarrow$ `maxConsecutiveOnes`(**q** XN̄OR $\mathbf{y}_i, false$)
  **smm_distance** $\leftarrow$ `getMax`(**result**, **enc_t**)
  *gt_threshold* $\leftarrow$ AND all bits in(**smm_distance** XN̄OR **enc_t**)
  **smm_distance** $\leftarrow$!*gt_threshold* AN̄D **smm_distance**
  **return** **smm_distance**

## 4. Experimental Analysis

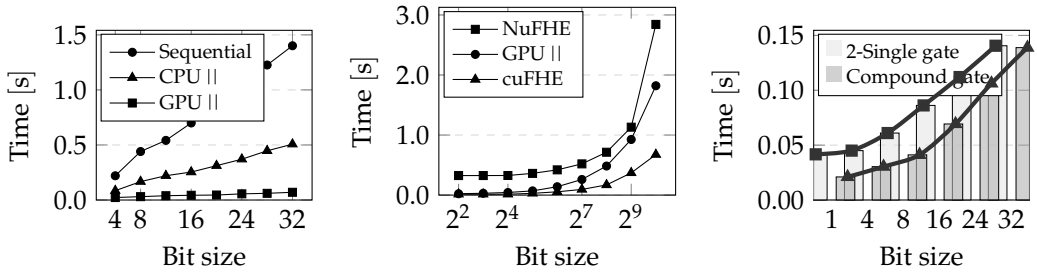
The experimental environment included an Intel(R) Core™ i7-2600 CPU having 16 GB system memory with an NVIDIA GeForce GTX 1080 GPU (check GPU details in Appendix A) with 8 GB memory [30]. The CPU and GPU contained 8 and 40,960 hardware threads, respectively. We used the same setup to analyze all three frameworks: sequential, CPU ||, and GPU ||.

We use two metrics for the comparison: (a) execution time and (b) speedup $= \frac{T_{seq}}{T_{par}}$. Here, $T_{seq}$ and $T_{par}$ are the time for computing the sequential and the parallel algorithm. In the following sections, we gradually analyze the complicated arithmetic circuits using the best results from the foregoing analysis.

### *4.1. GPU-Accelerated TFHE*

Initially, we discuss our performance over Boolean gate operations, deemed as building blocks of any computation. Figure 6a depicts the execution time difference among the sequential, CPU ||, and GPU || frameworks for $[4, 32]$ bits. The sequential `AND` operation takes a minimum of 0.22 s (4 bits), while the runtime increases to 1.4 s for 32 bits.

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




(**a**) Comparison of Seq., CPU, and GPU parallel framework

(**b**) Analysis with GPU-assisted frameworks

(**c**) Compound gate against 2 single gates

**Figure 6.** Performance analysis of GPU-accelerated TFHE with the sequential and CPU || frameworks (**a**) and comparison with the existing GPU-assisted libraries (**b**). (**c**) presents the performance of compound gates against 2 single-gate operations, while x-axis and y-axis represent bit size and time in seconds.

Then, we further scrutinize the execution time by dividing gate operations into three major components—(a) bootstrapping, (b) key switching, and (c) miscellaneous. We select the first two as they are the most time-consuming operations and fairly generalizable to other HE schemes. Table 2 shows the difference in execution time between the sequential and the GPU parallel for $\{2, \ldots, 32\}$ bits. We show that the execution time increment is less compared with the sequential approach.

**Table 2.** Computation time (ms) for bootstrapping, key switching, and misc. for sequential and GPU frameworks.

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

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

Under the same h/w setting, we benchmark our proposed framework with the existing GPU-based libraries (cuFHE and NuFHE). Although our GPU-parallel framework outperforms NuFHE for different bit sizes (Figure 6b), the performance degrades for larger bit sizes w.r.t. cuFHE. As the cuFHE implementation focuses more on the gate level optimization, we focus on the arithmetic circuit computations. In Section 4.3, we analyze our arithmetic circuits where our framework outperforms the existing GPU libraries.

### *4.2. Compound Gate Analysis*

According to Section 3.1.1, the compound gates are used to improve the execution time for additions or multiplications. Since the existing frameworks do not provide these optimizations, we benchmark the compound gates with the proposed single-gate computations. Figure 6c illustrates the performance of 1 compound gate over 2 single gates computed sequentially. We performed several iterations for a different number of bits

$(1, \ldots, 32)$, as shown on the x-axis while the y-axis represents the execution time. Notably, 32-bit compound gates will have two 32-bit inputs and output two 32 bits.

Here, bit coalescing improves the execution time as it takes only 0.02 s for one compound gate evaluation, compared with 0.04 s on performing 2 single gates sequentially. However, Figure 6c shows an interesting trend in the execution time between 2 single gates and 1 compound gate evaluation. The gap favoring the compound ones tends to get narrower for a higher number of bits. For example, the speedup for 1 bit happens to be $0.04/0.02 = 2$ times, whereas it reduces to 1.01 for 32 bits. The reason behind this diminishing performance is the *asynchronous launch queue* of GPUs.

As mentioned in Section 3.1.1, we use batch execution for the FFT operations. Hence, the number of parallel batches depends on the asynchronous launch queue size of the underlying GPU, which can delay the FFT operations for a large number batches. This ultimately adversely affects the speedup for large LWE sample vectors. Nevertheless, the analysis shows that the 1-bit compound gates are the most efficient, and we employ them in the following arithmetic operations.

### *4.3. Addition*

Table 3 presents a comparative analysis of the addition operation for 16-, 24-, and 32-bit encrypted numbers. We consider our proposed frameworks: sequential, CPU ||, and GPU ||, and benchmark them with cuFHE [13], NuFHE [14], and Cingulata [15]. Furthermore, we present the performance of two variants of addition operation: $\text{GPU}_n||$ (numberwise) and $\text{GPU}_1||$ (bitwise), as discussed in Section 3.1.2.

**Table 3.** Execution time (s) for the $n$-bit addition.

| **Frameworks** | **16-Bit** | **24-Bit** | **32-Bit** |
|---|---|---|---|
| Sequential | 3.51 | 5.23 | 7.04 |
| cuFHE [13] | 1.00 | 1.51 | 2.03 |
| NuFHE [14] | 2.92 | 3.56 | 4.16 |
| Cingulata [15] | 1.10 | 1.63 | 2.16 |
| **Our Methods** | | | |
| CPU || | 3.51 | 5.23 | 7.04 |
| $\text{GPU}_n||$ | 0.94 | 2.55 | 4.44 |
| $\text{GPU}_1||$ | 0.98 | 1.47 | 1.99 |

Table 3 demonstrates that $\text{GPU}_n||$ performs better than the sequential and CPU || circuits. $\text{GPU}_n$ provides a 3.72× speedup for 16 bits, whereas 1.58× for 32 bits. However, $\text{GPU}_n||$ performs better only for 16-bit additions compared with $\text{GPU}_1||$. For 24- and 32-bit additions, $\text{GPU}_1||$ performs around 2× better than $\text{GPU}_n||$. This improvement in essential as it reveals the algorithm to choose between $\text{GPU}_1||$ and $\text{GPU}_n||$.

Although both addition operations ($\text{GPU}_n||$ and $\text{GPU}_1||$) utilize compound gates, they differ in the number of input bits ($n$ and 1 for $\text{GPU}_n||$ and $\text{GPU}_1||$, respectively). Since the compound gates perform better for smaller bits (Section 4.2), the bitwise addition performs better than the numberwise addition for 24/32-bit operations. Hence, we utilize bitwise addition for building other circuits.

NuFHE and cuFHE do not provide any arithmetic circuits in their library. Therefore, we implemented such circuits on their library and performed the same experiments. Additionally, we considered Cingulata [15] (a compiler toolchain for TFHE) and compared the execution time. Table 3 summarizes all the results, where we found that our proposed addition circuit ($\text{GPU}_1||$) outperforms the other approaches.

We further experimented on the vector additions adopting the bitwise addition and showed the analysis in Table 4. Like addition, the performance improvement on the vector addition is also noticeable. The framework scales by taking a similar execution time for smaller vector lengths $\ell \leq 8$. However, the execution time increases for longer vectors as they involve more parallel bit computations and, consequently, increase the batch size of FFT operations. The difference is clearer on 32-bit vector additions with $\ell = 32$ which takes almost twice the time of $\ell = 16$. However, for $\ell \leq 8$, the execution times are almost similar due to the parallel computations. In Section 4.2, we have discussed this issue, which relies on the FFT batch size. Notably, Figure 6c also aligns with this evidence as the larger batch size for FFT on GPUs affects the speedup. For example, $\ell = 32$ will require more FFT batches compared with $\ell = 16$, which requires more time to finish the addition operation. We did not include other frameworks in Table 4 since our GPU || performed better compared with the others in Table 3.

**Table 4.** Execution time (s) for vector addition.

| **Length** | **16-Bit** | | | **32-Bit** | | |
|---|---|---|---|---|---|---|
| $\ell$ | **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 |

### *4.4. Multiplication*

The multiplication operation uses a sequential accumulation (reduce by addition) operation. Instead, we use a tree-based vector addition approach (discussed in Section 4.4) and gain a significant speedup. Table 5 portrays the execution times for the multiplication operations using the frameworks. Here, we employed all available threads on the machine. Like the addition circuit performance, here, GPU || outperforms the sequential circuits and CPU || operations by a factor of ≈11 and ≈14.5, respectively, for 32-bit multiplication.

**Table 5.** Multiplication execution time (s) comparison.

| **Frameworks** | **16-Bit** | **24-Bit** | **32-Bit** |
|---|---|---|---|
| | | **Naive** | |
| Sequential | 120.64 | 273.82 | 489.94 |
| CPU \|\| | 52.77 | 101.22 | 174.54 |
| GPU \|\| | 11.16 | 22.08 | 33.99 |
| cuFHE [13] | 32.75 | 74.21 | 132.23 |
| NuFHE [14] | 47.72 | 105.48 | 186.00 |
| Cingulata [15] | 11.50 | 27.04 | 50.69 |
| | | Karatsuba | |
| CPU \|\| | 54.76 | - | 177.04 |
| GPU \|\| | 7.6708 | - | 24.62 |

We further implemented the multiplication circuit on cuFHE and NuFHE. Table 5 summarizes the results comparing our proposed framework with cuFHE, NuFHE, and Cingulata. Our GPU || framework is faster in execution time than the other techniques.

Notably, the performance improvement is scalable with the increasing number of bits. This is due to tree-based additions following the reduction operations and computing all Boolean gate operations by coalescing the bits altogether.

Additionally, we analyze vector multiplications available in our framework and present a comparison among the frameworks in Table 6. We found out an increase in execution time for a certain length (e.g., $\ell = 32$ on 16-bit or $\ell = 4$ on 32-bit), which is similar to the issue in vector addition (Section 4.3). Hence, the vector operations from $\ell \leq 16$ can be sequentially added to compute arbitrary vector operations. For example, we can use two $\ell = 16$ vector multiplications to compute $\ell = 32$ multiplication, resulting in around 11 min. In the vector analysis, we did not add the computations over the other frameworks since our framework surpassed their achievements for single multiplications.

**Table 6.** Execution time (in minutes) for vector multiplication.

| **Length** | **16-Bit** | | | **32-Bit** | | |
|---|---|---|---|---|---|---|
| $\ell$ | **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 |

### *4.5. Karatsuba Multiplication*

In Table 5, we provide execution time for 16- and 24-bit Karatsuba multiplications over encrypted numbers as well. In the CPU || construction of the algorithm, the execution time does not improve; rather, it increases slightly. We observed that for both 16- and 32-bit multiplications, Karatsuba outperforms the naive GPU|| multiplication algorithm on GPU by 1.50 times. Karatsuba multiplication can also be considered a complex arithmetic operation as it comprises 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.

### *4.6. String Search Operations*

In Table 7, we report the execution time for the three string search operations. Here, we report the execution time in seconds, where we change the size of the genomic data. The values of $m = \{8, 16, \ldots, 256\}$ denote the number of genes for the query $q$ and target $y$.

**Table 7.** Execution time (in seconds) for variable size query and target sequence $m$ for different distance metrics.

| **Method** | **m** | | | | | |
|---|---|---|---|---|---|---|
| | **8** | **16** | **32** | **64** | **128** | **256** |
| Hamming distance | 2.89 | 11.84 | 47.95 | 189.81 | 758.73 | 3035.0 |
| Set-maximal | 3.76 | 13.3 | 51.24 | 195.72 | 771.08 | 3061.48 |
| Set-maximal (with $t$) | 7.15 | 20.67 | 64.43 | 223.14 | 827.76 | 3173.34 |
| Edit distance | 662 | 2577 | 9989 | 39,022 | 154,194 | 612,435 |

The results show that hamming distance requires the least amount of time. It is also clear from Definition 1 as it requires an XOR operation. The set-maximal matches (Definition 3) need more operations as `maxConsecutiveOnes` in Algorithm 3 employs the half adder for all bits. Furthermore, the threshold version of SMM takes more time since

we need to perform the `getMax` operation. For example, for a target and query sequence size of $m = 128$, it takes around 14 and 13 min with and without an existing threshold. However, for a smaller size of $m$, we can see that the time difference is more significant as it takes 3.76 s to perform SMM, compared with 7.15 s.

Edit Distance (2) takes the highest amount of time for the same genome size $m$. For example, for a sequence of size $m = 32$, edit distance under FHE takes around 2 h, whereas hamming or set-maximal matches take less than a minute. Notably, in these methods, we use Algorithms 5 and 4 for $m < 32$, whereas we use the alternative (subtraction) method for larger sequences.

## 5. Discussion

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

**Is the proposed framework sufficient to implement any computations?** This paper discusses in detail the proper implementation of Boolean gates using GPUs to enhance performance. It further explains the process of conducting basic arithmetic computations like addition, multiplication, and matrix operations using the suggested model. However, the implementation of more sophisticated algorithms like secure machine learning, as referenced in sources [35,36], is not within the scope of this paper. Future research will dive into the potential to optimize this model further for machine learning algorithms.

**For the GPU || framework, how do we compute on encrypted data larger than the fixed GPU memory?**

Limitations like fixed GPU memory sizes and varying access speeds are common to all GPU || applications. These issues also arise in deep learning when managing larger datasets. The resolution lies in segmenting the data or utilizing multiple GPUs. The model we are proposing can also apply these solutions as it is flexible enough to manage larger ciphertexts.

**How can we achieve further speedup on both frameworks?** In the case of the CPU || model, we have tried to implement as many hardware and software level optimizations as we were able to. Nonetheless, our GPU || model partially relies on the slower global GPU memory. The memory speed is crucial as different device memories have various read/write speeds. `L1`, or shared memory, is the fastest after the register. We used a combination of shared and global memory due to the size of the ciphertext. Going forward, we plan to use only shared memory, which is smaller but is expected to enhance the speed compared with the present method.

**How would the bit security level affect the reported speedup?** Currently, our model is comparable to the TFHE implementation [37], offering a security level of 110 bits, which may not be enough for certain applications. That being said, our GPU || model is adaptable to any desired bit security level. However, any changes will also alter the execution times. For instance, security levels lower than 110 bits will result in faster execution and vice versa for higher bit security. Future research will incorporate and evaluate the speedup for evolving bit security levels.

**Impact on computational accuracy while computing in GPUs.** In the earlier GPU architecture, GPUs offered lower precision than the IEEE-754 (Available online: https://ieeexplore.ieee.org/document/8766229, accessed on 8 January 2024) floating point standard. However, the GPUs we used for testing did not have such issues as they offered double precision. Nevertheless, it is important to understand that, during FHE encryption and operations, we tend to lose precision as encrypted bits get noisier with each gate operation. Therefore, any result from FHE computation does not offer the IEEE floating point standard, to the best of our knowledge. Consequently, our framework also suffers from the same lower precision inherited by FHE limitations.

## 6. Related Works

### *6.1. Parallel Frameworks for FHE*

In this section, we discuss the other HE schemes from Table 8 and categorize schemes based on their number representation:

1. Bitwise;
2. modular; and
3. approximate.

**Table 8.** A comparative analysis of existing homomorphic encryption schemes for different parameters on a 32-bit number.

| | Year | Homomorphism | Bootstrapping | Parallelism | Bit security | Size (kb) | Add. (ms) | Mult. (ms) |
|---|---|---|---|---|---|---|---|---|
| RSA [38] | 1978 | Partial | × | × | 128 | 0.9 | × | 5 |
| Paillier [39] | 1999 | Partial | × | × | 128 | 0.3 | 4 | × |
| TFHE [12] | 2016 | Fully | Exact | AVX [22] | 110 | 31.5 | 7044 | 489,938 |
| HEEAN [20] | 2018 | Somewhat | Approximate | CPU | 157 | 7168 | 11.37 | 1215 |
| SEAL (BFV) [40] | 2019 | Somewhat | × | × | 157 | 8806 | 4237 | 23,954 |
| cuFHE [13] | 2018 | Fully | Exact | GPU | 110 | 31.5 | 2032 | 132,231 |
| NuFHE [14] | 2018 | Fully | Exact | GPU | 110 | 31.5 | 4162 | 186,011 |
| Cigulata [15] | 2018 | Fully | Exact | × | 110 | 31.5 | 2160 | 50,690 |
| Our Method | - | Fully | Exact | GPU | 110 | 31.5 | 1991 | 33,930 |

**Bitwise encryption** works by encrypting the bit representation of a number. Its computation is also performed bit by bit, with each bit being handled independently of others. This characteristic is particularly beneficial for our parallel framework as the bit independence allows for parallel operation and reduces dependencies. The advantages of this process include increased speed in bootstrapping and a reduction in ciphertext size, suitable attributes considering the constraints of fixed-memory GPUs. This concept was first formalized and dubbed as GSW in 2013 [41], and has since been advanced over time [12,17,25].

**Modular encryption** techniques employ a fixed modulus, represented as $q$, that defines the size of the ciphertexts. This approach has witnessed vast improvements [42,43], particularly due to its reasonable execution time (Table 8). The quick addition and multiplication times from FV [26] and SEAL [40] demonstrate their superior speed compared with our GPU-based framework.

However, these schemes present a compromise between bootstrapping and efficiency. Often categorized as somewhat homomorphic encryption, they predefine the number of computations or the magnitude of multiplications, lacking a process for noise reduction. The resultant larger ciphertexts are consequent of large $q$ values.

For example, we selected the ciphertext moduli of 250 and 881 bits for FV-NFLlib [26] and SEAL [40], respectively. The polynomial degrees ($d$) were chosen as 13 and 15 for the two frameworks as it was required to comply with the targeted bit security to populate Table 8. It is noteworthy that smaller $q$ and $d$ will result in a faster runtime and smaller ciphertexts, but they will limit the number of computations as well. Therefore, this modular representation requires fixing the number of homomorphic operations limiting the use cases.

**Approximate number** representations were recently proposed by Cheon et al. (CKKS [44]) in 2017. These schemes also provide efficient single instruction multiple data (SIMD) [45] operations similar to the modular representations as mentioned above. However, they have an inexact but efficient bootstrapping mechanism, which can be applied in less precision-demanding applications. The cryptosystem also incurs larger ciphertexts (7MB) similar to the modular approach as we tested it for $q = 1050$ and $d = 15$. Here, we did not discuss HELib [46], the first cornerstone of all HE implementations, since its cryptosystem BGV [43] is enhanced and utilized by the other modular HE schemes (such as SEAL [40]).

Our goal is to parallelize a fully homomorphic encryption (FHE) scheme. Most homomorphic encryption (HE) schemes following modular encryption are either somewhat or adopt approximate bootstrapping while requiring more memory post encryption. As a result, we opt for a bitwise bootstrappable encryption scheme: TFHE.

In terms of **hardware solutions**, few have been studied and utilized to enhance the efficiency of FHE computations. Following the formulation of FHE with ideal lattices, most efficiency enhancements have been approached from the standpoint of asymptotic runtimes. A select few approaches have dealt with the inclusion of existing multiprocessors like GPU or FPGAs [47] to achieve quicker homomorphic operations. 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.

Lei et al. ported FHEW-V2 [25] to GPU [51] and extended the Boolean implementation to 30-bit addition and 6-bit multiplication with a speed up to ≈2.5. Since TFHE extends FHEW and performs better than its predecessor, we consider TFHE as our baseline framework.

In 2015, a GPU-based HE scheme, CuHE [49], was proposed. However, it was not fully homomorphic as it did not have bootstrapping; hence, we did not include it in our analyses. Later in 2018, two GPU FHE libraries, cuFHE [13] and NuFHE [14], were released. Both libraries focused on optimizing the Boolean gate operations. Recently, Yang et al. [52] benchmarked cuFHE and its predecessor, TFHE, and analyzed the speedup, which we also discuss in this article (Table 8).

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

Recently, Zhou et al. improved TFHE by reducing and performing the serial operations of bootstrapping in parallel [53]. However, they did use any hardware acceleration to the existing FHE operations. We consider this work as an essential future direction that can be integrated to our framework for better executing times.

Cingulata or Armadillo [15] is also a related work that proposed a compiler toolkit designed to work on homomorphically encrypted data, written in C++. Cingulata can handle a large number of parallel operations to mitigate the homomorphic encryption's performance overhead. However, in this work, we propose to perform similar optimization on GPUs, using CUDA-enabled computations.

More recently, Concrete [54], an open-source compiler using TFHE, was proposed, which simplifies the complexities of general computation under FHE. It provides several translations that allow arbitrary computations to be performed under encryption using a vanilla Python script with additional decorators. In the future, we look forward to extending our GPU-parallel features according to this framework.

### *6.2. Secure String Distances in Genomic Data*

In one of the earlier attempts with a secure multiparty setting, Jha et al. [55] proposed a privacy-preserving genomic sequence similarity in 2008. Their paper showed three different methods to mirror the Levenshtein distance algorithm using a garbled circuit. However, for a sequence of 25 nucleotides, it took around 40 s to compute the distance metric between two strings. In 2015, Wang et al. [9] proposed an approximation of the original edit distance in a more realistic setting, where the authors utilized a reference genomic sequence to compute the edit distance. However, we analyzed its accuracy in one of our earlier works [10] and showed that the accuracy drops for longer input sequences.

In a recent attempt, Shimzu et al. [33] proposed a Burrows–Wheeler transformation for finding target queries on a genomic dataset. The authors attempted the set-maximal matches using oblivious transfer on a two-party privacy setting. However, we employ a completely different cryptographic technique as we do not require the researcher to stay active upon providing their encrypted queries. Therefore, the whole computation can be

offloaded to a cloud server and harness its full computational capacity. One of the first attempts with FHE to compute edit distance was conducted by Cheon et al. [44]. Given the advances in 2017, their cryptographic scheme was impressive, though taking 16.4 s to compute a 8 × 8 block of string inputs. However, the underlying techniques have improved, allowing a larger string comparison using FHE techniques as we have shown in this work.

## 7. Conclusions

In this study, we developed algebraic circuits for fully homomorphic encryption (FHE), making them available for any complex operations. In addition, we investigated the use of CPU-level parallelism to enhance the execution speed of underlying FHE computations. A significant innovation in our work is the introduction of a GPU-level parallel framework that leverages novel optimizations such as bit coalescing, compound gate, and tree-based vector accumulation. Furthermore, we implemented this framework in genomic string operations and evaluated its effectiveness. The experimental results demonstrate that the methodology we propose is 20 times and 14.5 times quicker than the existing method for executing Boolean gates and multiplications, respectively.

**Author Contributions:** Conceptualization, M.M.A.A.; Methodology, M.M.A.A.; Software, M.T.M.T.; Validation, M.M.A.A. and N.M.; Writing—original draft, M.M.A.A. and M.T.M.T.; Writing—review & editing, N.M.; Supervision, N.M. All authors have read and agreed to the published version of the manuscript.

**Funding:** The research is supported in part by the CS UManitoba Computing Clusters and Amazon Research Grant. N.M. was supported in part by the NSERC Discovery Grants (RGPIN-04127-2022) and Falconer Emerging Researcher Rh Award.

**Institutional Review Board Statement:** Not applicable.

**Informed Consent Statement:** Not applicable.

**Data Availability Statement:** There are no proprietary or closed-source senstiive date used for the analysis.

**Conflicts of Interest:** The authors declare no conflicts of interest.

## Appendix A. GPU 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 GTX 1080, has 20 SMs, where SM contains 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.

### *Appendix A.1. Memory Hierarchy in GPU*

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. Figure A1 portrays the memory categories and their organization. We present a brief discussion on the memory categories:

(a) Register;
(b) Cache;
(c) Shared;
(d) Constant; and

(e) Global.

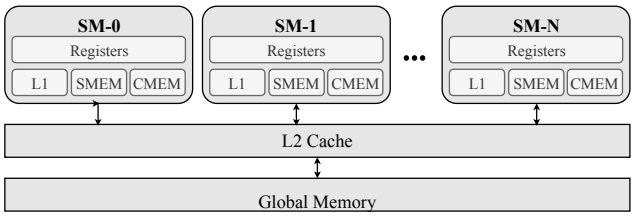



**Figure A1.** GPU memory hierarchy.

**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 a private L1 cache. On the contrary, the L2 cache has a latency that is higher than that of 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.Shared memory is private to SMs as well, but public to the threads inside the SMs.

**Constant memory.** 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 A1) among all memory categories, yet the slowest and nonpersistent. One major limitation of global memory is that it is fixed, while the main memory can be changed for the CPUs.

**Table A1.** 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 |

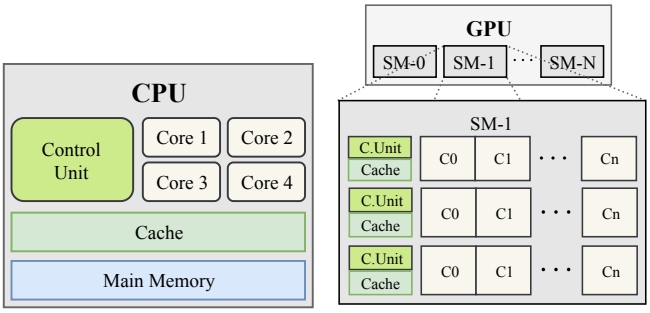
*Appendix A.2. Computational and Memory Hierarchy Coordination*

The coordination between computation and memory hierarchy is a crucial aspect to take 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 are inaccessible by others inside the same SM. The number of grids can be at most the number of global memories, and the global memory is shareable from all SMs.

Bit coalescing (Section 6.1) discusses the unification of LWE samples. Hence, 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.

*Appendix A.3. Architectural Differences with CPU*

**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 SMs, where each SM possesses a large number of cores. For example, in Figure A2, the CPU comprises 4 independent cores, while the GPU consists of $N$ SMs with $n$ CUDA cores in each SM. Thus, GPUs offer more parallel computing power for any computation.




(**a**) CPU architecture (**b**) GPU architecture

**Figure A2.** A schematic illustration of CPU (**a**) and GPU (**b**) architecture. Unit and $C_n$ represent a control unit and a core in GPU, respectively. (**b**) illustrates an SM construction.

**Computation complexity.** Although GPUs provide more scopes of parallelism, GPU cores lack the computational power. CPU cores have higher clock cycle (3.40 GHz) than GPU (1734 MHz), as shown in Table A1. Moreover, CPU cores are capable of executing complex instruction of small data. On the contrary, a GPU core is simple and typically consists of an execution unit of integers and float numbers [56].

**Memory space.** Table A1 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 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 a 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.

## References

1. Gentry, C. Fully homomorphic encryption using ideal lattices. In Proceedings of the STOC, Bethesda, MD, USA, 31 May 31–2 June 2009; Volume 9, pp. 169–178.
2. Pham, A.; Dacosta, I.; Endignoux, G.; Pastoriza, J.R.T.; Huguenin, K.; Hubaux, J.P. ORide: A Privacy-Preserving yet Accountable Ride-Hailing Service. In Proceedings of the 26th USENIX Security Symposium (USENIX Security 17), Vancouver, BC, Canada, 16–18 August 2017; pp. 1235–1252.
3. Kim, M.; Song, Y.; Cheon, J.H. Secure searching of biomarkers through hybrid homomorphic encryption scheme. *BMC Med. Genom.* **2017**, *10*, 42. [CrossRef] [PubMed]
4. Chen, H.; Gilad-Bachrach, R.; Han, K.; Huang, Z.; Jalali, A.; Laine, K.; Lauter, K. Logistic regression over encrypted data from fully homomorphic encryption. *BMC Med. Genom.* **2018**, *11*, 81. [CrossRef] [PubMed]
5. Morshed, T.; Alhadidi, D.; Mohammed, N. Parallel Linear Regression on Encrypted Data. In Proceedings of the In Proceedings of 16th Annual Conference on Privacy, Security and Trust (PST), Belfast, Ireland, 28–30 August 2018; pp. 1–5.
6. Naveed, M.; Ayday, E.; Clayton, E.W.; Fellay, J.; Gunter, C.A.; Hubaux, J.P.; Malin, B.A.; Wang, X. Privacy in the genomic era. *ACM Comput. Surv. (CSUR)* **2015**, *48*, 6. [CrossRef] [PubMed]
7. Aziz, M.M.A.; Sadat, M.N.; Alhadidi, D.; Wang, S.; Jiang, X.; Brown, C.L.; Mohammed, N. Privacy-preserving techniques of genomic data: A survey. *Briefings Bioinform.* **2017**, *20*, 887–895. [CrossRef]
8. 23AndMe.com. Our Health + Ancestry DNA Service—23AndMe Canada. Available online: https://www.23andme.com/en-ca/dna-health-ancestry (accessed on 20 November 2020).

9. Wang, X.S.; Huang, Y.; Zhao, Y.; Tang, H.; Wang, X.; Bu, D. Efficient genome-wide, privacy-preserving similar patient query based on private edit distance. In Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, 12–16 October 2015; ACM: New York, NY, USA, 2015; pp. 492–503.
10. Al Aziz, M.M.; Alhadidi, D.; Mohammed, N. Secure approximation of edit distance on genomic data. *BMC Med. Genom.* **2017**, *10*, 41. [CrossRef]
11. Guerrini, C.J.; Robinson, J.O.; Petersen, D.; McGuire, A.L. Should police have access to genetic genealogy databases? Capturing the Golden State Killer and other criminals using a controversial new forensic technique. *PLoS Biol.* **2018**, *16*, e2006906. [CrossRef]
12. Chillotti, I.; Gama, N.; Georgieva, M.; Izabachène, M. Faster fully homomorphic encryption: Bootstrapping in less than 0.1 seconds. In Proceedings of the Advances in Cryptology—ASIACRYPT 2016: 22nd International Conference on the Theory and Application of Cryptology and Information Security, Hanoi, Vietnam, 4–8 December 2016; Springer: Berlin/Heidelberg, Germany, 2016; pp. 3–33.
13. CUDA-Accelerated Fully Homomorphic Encryption Library. 2019. Available online: https://github.com/vernamlab/cuFHE (accessed on 15 December 2023 ).
14. NuFHE, a GPU-Powered Torus FHE Implementation. 2019. Available online: https://github.com/nucypher/nufhe (accessed on 15 December 2023).
15. Cingulata. 2019. Available online: https://github.com/CEA-LIST/Cingulata (accessed on 15 December 2023).
16. Cheon, J.H.; Kim, M.; Lauter, K. Homomorphic computation of edit distance. In Proceedings of the International Conference on Financial Cryptography and Data Security, San Juan, Puerto Rico, 26–30 January 2015; Springer: Berlin/Heidelberg, Germany, 2015; pp. 194–212.
17. Chillotti, I.; Gama, N.; Georgieva, M.; Izabachène, M. Faster packed homomorphic operations and efficient circuit bootstrapping for TFHE. In Proceedings of the International Conference on the Theory and Application of Cryptology and Information Security, Hong Kong, China, 3–7 December 2017; Springer: Berlin/Heidelberg, Germany, 2017; pp. 377–408.
18. Morshed, T.; Aziz, M.; Mohammed, N. CPU and GPU Accelerated Fully Homomorphic Encryption. In Proceedings of the 2020 IEEE International Symposium on Hardware Oriented Security and Trust (HOST), Los Alamitos, CA, USA, 7–11 December 2020; pp. 142–153. [CrossRef]
19. Regev, O. On lattices, learning with errors, random linear codes, and cryptography. *J. ACM* **2009**, *56*, 34. [CrossRef]
20. Cheon, J.H.; Han, K.; Kim, A.; Kim, M.; Song, Y. Bootstrapping for approximate homomorphic encryption. In Proceedings of the Annual International Conference on the Theory and Applications of Cryptographic Techniques, Tel Aviv, Israel, 29 April–3 May 2018; Springer: Berlin/Heidelberg, Germany, 2018; pp. 360–384.
21. Boura, C.; Gama, N.; Georgieva, M. Chimera: A Unified Framework for B/FV, TFHE and HEAAN Fully Homomorphic Encryption and Predictions for Deep Learning. *IACR Cryptol. ePrint Arch.* **2018**, *2018*, 758.
22. Lomont, C. Introduction to Intel Advanced Vector Extensions. Intel White Paper 2011, pp. 1–21. Available online: https://hpc.llnl.gov/sites/default/files/intelAVXintro.pdf (accessed on 16 December 2023).
23. Frigo, M.; Johnson, S.G. 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), Seattle, WA, USA, 12–15 May 1998; Volume 3, pp. 1381–1384.
24. Brakerski, Z.; Gentry, C.; Halevi, S. Packed ciphertexts in LWE-based homomorphic encryption. In Proceedings of the International Workshop on Public Key Cryptography, Nara, Japan, 26 February–1 March 2013; Springer: Berlin/Heidelberg, Germany, 2013; pp. 1–13.
25. Ducas, L.; Micciancio, D. FHEW: Bootstrapping Homomorphic Encryption in Less Than a Second. Cryptology ePrint Archive, Report 2014/816,2014 . Available online: https://eprint.iacr.org/2014/816 (accessed on 20 December 2023).
26. Fan, J.; Vercauteren, F. Somewhat Practical Fully Homomorphic Encryption. *IACR Cryptol. Eprint Arch.* **2012**, *2012*, 144.
27. McGeoch, C.C. Parallel Addition. *Am. Math. Mon.* **1993**, *100*, 867–871. [CrossRef]
28. Karatsuba, A.A.; Ofman, Y.P. Multiplication of many-digital numbers by automatic computers. In *Proceedings of the Doklady Akademii Nauk*; Russian Academy of Sciences: Moskva, Russia, 1962; Volume 145, pp. 293–294.
29. Chandra, R.; Dagum, L.; Kohr, D.; Menon, R.; Maydan, D.; McDonald, J. *Parallel Programming in OpenMP*; Morgan Kaufmann: Cambridge, MA, USA, 2001.
30. NVIDIA. GeForce GTX 1080 Graphics Cards from NVIDIA GeForce. Available online: https://www.nvidia.com/en-us/geforce/products/10series/geforce-gtx-1080/ (accessed on 20 December 2023).
31. Fickett, J.W. Fast optimal alignment. *Nucleic Acids Res.* **1984**, *12*, 175–179. [CrossRef]
32. Sotiraki, K.; Ghosh, E.; Chen, H. Privately computing set-maximal matches in genomic data. *BMC Med. Genom.* **2020**, *13*, 72. [CrossRef]
33. Shimizu, K.; Nuida, K.; Rätsch, G. Efficient Privacy-Preserving String Search and an Application in Genomics. *Bioinformatics* **2016**, *32*, 1652–1661. [CrossRef]
34. Durbin, R. Efficient haplotype matching and storage using the positional Burrows–Wheeler transform (PBWT). *Bioinformatics* **2014**, *30*, 1266–1272. [CrossRef]
35. Xie, P.; Bilenko, M.; Finley, T.; Gilad-Bachrach, R.; Lauter, K.; Naehrig, M. Crypto-nets: Neural networks over encrypted data. *arXiv* **2014**, arXiv:1412.6181.

36. Takabi, H.; Hesamifard, E.; Ghasemi, M. Privacy preserving multi-party machine learning with homomorphic encryption. In Proceedings of the 29th Annual Conference on Neural Information Processing Systems (NIPS), Barcelona, Spain, 5–10 December 2016.
37. Chillotti, I.; Gama, N.; Georgieva, M.; Izabachène, M. TFHE: Fast Fully Homomorphic Encryption Library. August 2016. Available online: https://tfhe.github.io/tfhe/ (accessed on 20 December 2023).
38. Rivest, R.L.; Shamir, A.; Adleman, L. A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM* **1978**, *21*, 120–126. [CrossRef]
39. Paillier, P. Public-key cryptosystems based on composite degree residuosity classes. In Proceedings of the Advances in cryptology, EUROCRYPT, Prague, Czech Republic, 2–6 May 1999; Springer: Berlin/Heidelberg, Germany, 1999; pp. 223–238.
40. Microsoft SEAL (Release 3.2). 2019. Microsoft Research, Redmond, WA. Available online: https://github.com/Microsoft/SEAL (accessed on 20 December 2023).
41. Gentry, C.; Sahai, A.; Waters, B. Homomorphic encryption from learning with errors: Conceptually-simpler, asymptotically-faster, attribute-based. In *Advances in Cryptology–CRYPTO 2013*; Springer: Berlin/Heidelberg, Germany, 2013; pp. 75–92.
42. Brakerski, Z.; Vaikuntanathan, V. Efficient fully homomorphic encryption from (standard) LWE. *SIAM J. Comput.* **2014**, *43*, 831–871. [CrossRef]
43. Brakerski, Z.; Gentry, C.; Vaikuntanathan, V. (Leveled) fully homomorphic encryption without bootstrapping. *ACM Trans. Comput. Theory* **2014**, *6*, 13. [CrossRef]
44. Cheon, J.H.; Kim, A.; Kim, M.; Song, Y. Homomorphic encryption for arithmetic of approximate numbers. In Proceedings of the International Conference on the Theory and Application of Cryptology and Information Security, Hong Kong, China, 3–7 December 2017; Springer: Berlin/Heidelberg, Germany, 2017; pp. 409–437.
45. Flynn, M.J. Some computer organizations and their effectiveness. *IEEE Trans. Comput.* **1972**, *100*, 948–960. [CrossRef]
46. Halevi, S.; Shoup, V. Algorithms in helib. In Proceedings of the Annual Cryptology Conference, Santa Barbara, CA, USA, 17–21 August 2014; Springer: Berlin/Heidelberg, Germany, 2014; pp. 554–571.
47. Doröz, Y.; Öztürk, E.; Savaş, E.; Sunar, B. Accelerating LTV based homomorphic encryption in reconfigurable hardware. In Proceedings of the International Workshop on Cryptographic Hardware and Embedded Systems, Saint-Malo, France, 13–16 September 2015; Springer: Berlin/Heidelberg, Germany, 2015; pp. 185–204.
48. López-Alt, A.; Tromer, E.; Vaikuntanathan, V. 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, New York, NY, USA, 20–22 May 2012; ACM: New York, NY, USA, 2012; pp. 1219–1234.
49. Dai, W.; Sunar, B. cuHE: A homomorphic encryption accelerator library. In Proceedings of the International Conference on Cryptography and Information Security in the Balkans, Koper, Slovenia, 3–4 September 2015; Springer: Berlin/Heidelberg, Germany, 2015; pp. 169–186.
50. Dai, W.; Doröz, Y.; Sunar, B. Accelerating swhe based pirs using gpus. In Proceedings of the International Conference on Financial Cryptography and Data Security, San Juan, Puerto Rico, 26–30 January 2015; Springer: Berlin/Heidelberg, Germany, 2015; pp. 160–171.
51. Lei, X.; Guo, R.; Zhang, F.; Wang, L.; Xu, R.; Qu, G. Accelerating Homomorphic Full Adder based on FHEW Using Multicore CPU and GPUs. In Proceedings of the 2019 IEEE 21st International Conference on High Performance Computing and Communications; IEEE 17th International Conference on Smart City; IEEE 5th International Conference on Data Science and Systems (HPCC/SmartCity/DSS), Zhangjiajie, China , 10–12 August 2019.
52. Yang, H.; Yao, W.; Liu, W.; Wei, B. Efficiency Analysis of TFHE Fully Homomorphic Encryption Software Library Based on GPU. In Proceedings of the Workshops of the International Conference on Advanced Information Networking and Applications, Matsue, Japan, 27–29 March 2019; Springer: Berlin/Heidelberg, Germany, 2019; pp. 93–102.
53. Zhou, T.; Yang, X.; Liu, L.; Zhang, W.; Li, N. Faster bootstrapping with multiple addends. *IEEE Access* **2018**, *6*, 49868–49876. [CrossRef]
54. Zama. Concrete: TFHE Compiler That Converts Python Programs into FHE Equivalent, 2022. Available online: https://github.com/zama-ai/concrete (accessed on 20 December 2023).
55. Jha, S.; Kruger, L.; Shmatikov, V. Towards practical privacy for genomic computation. In Proceedings of the 2008 IEEE Symposium on Security and Privacy (sp 2008), Oakland, CA, USA, 18–21 May 2008; pp. 216–230.
56. NVidia, F. *Nvidia's Next Generation Cuda Compute Architecture*; NVidia: Santa Clara, CA, USA, 2009.