A Programming API Implementation for Secure Data Analytics Applications with Homomorphic Encryption on GPUs

Shuangsheng Lou
Computer Science and Engineering
The Ohio State University
lou.125@osu.edu

Gagan Agrawal
Computer and Cyber Sciences
Augusta University
gagrawal@augusta.edu

***Abstract*—As sensitive data is frequently stored and processed in environments that are either shared, untrusted or otherwise can be compromised, privacy is frequently a concern. To address this, a method that has been gaining popularity is to use Homomorphic Encryption (HE), which allows computation over encrypted data, i.e., without decrypting the data first. However, the overhead of such analytics (up to 4 orders of magnitude) is a detriment and while there have been a few previous efforts on reducing these overheads through the use of accelerators like GPUs, programmability is a concern. This paper addresses both performance and programmability concerns with the use of HE. We port the major pieces of Simple Encrypted Arithmetic Library (SEAL) from Microsoft to GPU using CUDA. Through these GPU-based functions, a new HE application can be developed easily. We demonstrate this by developing encrypted versions of three applications – CNN, k-means, and KNN. The speedups of execution time for CNN, k-means and KNN on a single GPU over CPU implementation achieve up to 81, 133 and 7 respectively.**

***Index Terms*—Homomorphic Encryption, Secure Data Analytics, Cloud applications, Security, Programmability, GPU**

I. Introduction

Over the last two decades, data analytics applications have been studied and have achieved remarkable results in many domains. At the same time, the rapidly increasing volume has created the challenge of finding resources to match the scale of data. Naturally, the community is attracted to the advantages of cloud systems such as elasticity, flexibility, and sharing of resources by users from several organizations [5], [9], [20].

However, using the cloud to enable data-driven collaboration creates new challenges. A prominent challenge is the *security* and *privacy* of the data [19], [22], [11], [24]. Cloud (data) security has been recognized as one of the major challenges for cloud systems since their inception [23], [8], [14], [13]. Specifically, an attacker or adversary having an unauthorized access to the storage on the cloud can mine the data and retrieve large amounts of confidential data, which can be a severe risk in many areas, such as the financial and medical fields.

Traditional encryption solutions mainly address the security and privacy issues for the data storage or during the transmission of data, and not during the data analysis itself. To securely perform analytics in the public clouds that may not be fully trustful, a popular approach that has emerged in this area is to utilize Homomorphic Encryption (HE). HE supports many arithmetical operations over ciphertexts without requiring decryption [16], [1]. By careful design, this can enable one to fully evaluate functions on encrypted data without decryption [12]. Overall, there are three notable classes of schemes [1] – partially homomorphic encryption schemes (PHE), somewhat homomorphic encryption schemes (SWHE) and fully homomorphic encryption schemes (FHE). PHE [16] allows only one type of operation, either addition or multiplication, for an unlimited number of computation times. SWHE allows certain types of operations for a limited number of computation times, whereas FHE [6], [2] allows an unlimited number of operations for an unlimited number of computation times.

With the goal of providing secure, privacy-preserving data analytics when the data stored in a public cloud is analyzed, it is natural to build on FHE [1] instead of PHE or SWHE. However, the FHE-based privacy-preserving solution has a big drawback, which is its high computational overhead. Another challenge for developing data analytics functions using HE is that with only a few operations supported using HE (multiplication and addition), data analytics algorithms cannot be translated to encrypted versions without modification. For example, in Convolutional Neural Network (CNN), activation functions such as Rectified Linear Unit (ReLU) and Sigmoid are used and need to be replaced with others functions that only use addition and multiplication.

The goal of this work is to address the computational, programmability, and algorithm modification challenges associated with developing HE-based data analytics. We do this by porting the major functions of the SEAL library from Microsoft[3] on GPUs. SEAL aims to provide high-performance and easy-to-use encryption functions that allow computations to be performed directly on encrypted data.

Overall, the contributions of this work are as follows: 1) We ease the programmability for data analytics applications with homomorphic encryption by designing secure advanced functions on the top of the SEAL library, 2) The development and introduction of integrated functions to perform comparison operation into CKKS scheme [6], [2], which is one of the schemes within the SEAL library, and 3) Demonstration of this functionality on GPUs by showing the development of

three significant applications – CNN, K-means clustering, and k-Nearest Neighbors (KNN) search, and 4) Comprehensive experimental evaluations of three data analytics applications that compare the efficiency of the CPU-based implementation to GPU-based implementation. The speedups of execution time for CNN, K-means and KNN on a single GPU over CPU implementation achieve up to 81x, 133x and 7x respectively and the operation optimizations of homomorphic encryption help us achieve up to 28% and 11% execution time reduction of CNN and KNN respectively.

II. Preliminaries

A. Fully Homomorphic Encryption

The scheme used in this work is the Cheon-Kim-Kim-Song (CKKS) scheme [4]. CKKS allows us to perform computations on vectors of complex values (thus real values as well). The CKKS scheme allows additions and multiplications on encrypted real or complex numbers, but yields only approximate results. In applications such as summing up encrypted real numbers, evaluating machine learning models on encrypted data, or computing distances of encrypted locations, CKKS is a good choice. Figure 1 provides a high-level view of CKKS. We can see that a message m, which is a vector of values on which we want to perform certain computation, is first encoded into a plaintext polynomial $p(X)$ and then encrypted using a public key.

Once the message m is encrypted into c, which is a couple of polynomials, CKKS provides several operations that can be performed on it, such as addition, multiplication, and rotation. If we denote a function by f, which is a composition of homomorphic operations, then decrypting $c' = f(c)$ with the secret key will yield $p' = f(p)$. Therefore, once we decode it, we will get $m = f(m)$. The central idea to implement a homomorphic encryption scheme is to have homomorphic properties on the encoder, decoder, encryptor, and the decryptor.

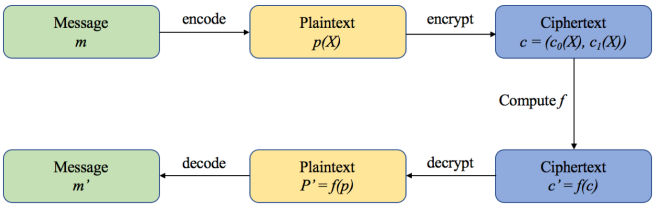

Fig. 1: High-level Overview of CKKS

B. Microsoft Seal Library

We built our privacy-preserving data clustering framework on the Seal library which is a free and open-source cross platform software library developed by Microsoft Research [18] and CKKS scheme described above is one of the supported schemes. The library includes the following key functions.

Encode and Decode: Encode/Decode process transforms the plaintext, a vector in R^N, to/from a polynomial in the N_{th} order cyclotomic polynomial ring (equivalently, the coefficients-vector of the polynomial) by the Canonical Embedding technique. Here we denote the degree of our polynomial degree modules by N, which will be a power of 2. We also denote the m-th cyclotomic polynomial by $\Phi_M(X) = X^N + 1$.

Encryption and Decryption: Encryption process transforms the plaintext to ciphertext and decryption process transforms the ciphertext to plaintext. The ciphertext can be expressed into a cyclotomic polynomial. If n is a positive integer, then the n^{th} cyclotomic polynomial is defined as the unique monic polynomial having exactly the primitive n^{th} roots of unity as its zeros. The public key and private key are needed in this process and are also cyclotomic polynomials. The security of this process is guaranteed by the hardness of Learning-With-Errors (LWE) problem, which is the foundation of CKKS [15].

Homomorphic operations: Homomorphic operations supported by the framework are addition, multiplication, and scaling operation. Scaling performs homomorphic rounding operation so as to keep the modulus of the ciphertext linear with respect to the depth of the computational circuit.

Addition: Suppose we have two messages, μ and μ' and that we encrypt them into $c = (c_0, c_1)$ and $c' = (c'_0, c'_1)$. Then $c\_add = c + c' = (c_0 + c'_0, c_1 + c'_1)$ is a correct encryption of $\mu + \mu'$, i.e. when we decrypt it using secret s we get (approximatively) $\mu + \mu'$. Indeed, the decryption mechanism c_add yields.

Plaintext-ciphertext multiplication: Suppose we have a plaintext μ, encrypted into the ciphertext $c = (c_0, c_1)$ and a plaintext μ'. Then to obtain the ciphertext of the multiplication, we simply need to output $c_{mult} = \mu' \times c_0 + \mu' \times c_1$. Indeed, when decrypting c_{mult} we get

$$\mu' \times c_0 + \mu' \times c_1 \times s = \mu' \times (c_0 + c_1 \times s) = \mu' \times (\mu + e) = \mu' \times \mu + \mu' \times e \quad (1)$$

Ciphertext-ciphertext multiplication: We can use the following operations to do ciphertext-ciphertext multiplication:

- $Cmult(c, c') = (d_0, d_1, d_2)$
- $Decryptmult(c_{mult}, s) = d_0 + d_1 \times s + d_2 \times s^2$

While ciphertexts were usually only a couple of polynomials, here we have 3 polynomials for our ciphertext. By using the same reasoning as before, to correctly decrypt the next product we will need 5 polynomials, then 9, and so on. Therefore the size of the ciphertext will grow exponentially and it will not be usable in practice if we were to define ciphertext-ciphertext multiplication like that. So relinearization is required for ciphertext-ciphertext multiplication so that the ciphertext size is not increased at each step.

Relinearization: The essence of relinearization is finding a couple of polynomials $(d'_0, d'_1) = Relin(c_{mult})$ such that: Decrypt$((d'_0, d'_1), s) = d'_0 + d'_1 \times s = d_0 + d_1 \times s + d_2 \times s^2$. So relinearization allows to have a polynomial couple, and not triple, such that once it is decrypted using the regular decryption circuit, which only needs the secret key, and not its square, we get the multiplication of the two underlying plaintexts. Therefore, if we perform relinearization after each ciphertext-ciphertext multiplication, we will always have ciphertexts of the same size, with the same decryption circuit.

Now, to define $Relin$, we will define $(d'_0, d'_1) = d_0 + d_1 + P$ where P represents a couple of polynomials such that Decrypt $(P, s) = d_2 \times s^2$.

Rescaling: We need a final operation called rescaling to manage the noise and avoid overflow. Rescaling is a kind of modulus switch operation, which removes the last of the primes from $coeff\_modulus$. As a side-effect, it also scales down the ciphertext by the removed prime. Usually we want to have perfect control over how the scales are changed, which is why for the CKKS scheme it is more common to

use carefully selected primes for the $coeff\_modulus$. More precisely, suppose that the scale in a CKKS ciphertext is S, and the last prime in the current $coeff\_modulus$ (for the ciphertext) is P. Rescaling to the next level changes the scale to S/P, and removes the prime P from the $coeff\_modulus$, as is common in modulus switching. The number of primes limits how many rescaling can be done, and thus limits the multiplicative depth of the computation.

It is possible to choose the initial scale freely. One good strategy can be to set the initial scale S and primes P_i in the $coeff\_modulus$ to be very close to each other. If ciphertexts have scale S before the multiplication, they have scale S^2 after multiplication, and S^2/P_i after rescaling. If all P_i are close to S, then S^2/P_i is close to S again. This way we stabilize the scales to be close to S throughout the computation. Generally, for a circuit of depth D, we need to rescale D times, i.e., we need to be able to remove D primes from the coefficient modulus. Once we have only one prime left in this term, the remaining prime must be larger than S by a few bits to preserve the pre-decimal-point value of the plaintext.

III. Proposed Approach to Developing GPU Library

Our overall goal is to both obtain high performance using GPUs while simplifying the use of Seal library-type functionality for application development. In this section, we will introduce the integrated functions designed to perform encode, decode, encryption, decryption, and homomorphic operations.

A. Integrated functions

```
(a) function Ciphertext Multiply_Plain
(Ciphertext x, Plaintext p){
Ciphertext x2;
evaluator.multiply_plain(x,p,x2);
evaluator.rescale_to_next_inplace(x2);
return x2;}
(b) function Ciphertext Multiply
(Ciphertext x, Ciphertext x2){
evaluator.multiply_inplace(x,x2);
evaluator.relinearize_inplace(x, relin_keys);
evaluator.rescale_to_next_inplace(x);
return x;}
(c) vector<double> FloatToVector
(double*Fvalue, size_t slot_count){
vector<double> Vvalue;
Vvalue.reserve(slot_count);
for (size_t i = 0; i < slot_count; i++)
input.push_back(Fvalue[i]);
return Vvalue;}
```

Listing 1: New Integrated Functions for GPU

It is complicated for developers to manually program the applications even though the basic operations have been supported by the CKKS scheme. Developers need to think about the encryption of data and the side effect brought by homomorphic encryption, i.e. the size of encryption data. We introduce a number of functions to address programmer burden.

(1)*Ciphertext Multiply_Plain(Ciphertext x, Plaintext p)*
This function computes the multiplication between the ciphertext x and plaintext p and then eliminate the scale effect of data. It combines plaintext-ciphertext multiplication and rescaling operation together, as shown in Listing 1 (a).

(2)*Ciphertext Multiply(Ciphertext x, Ciphertext x2)*
When executing the multiplication between two ciphertexts, relinearization and rescaling operations are needed to eliminate the scale effect of data. So in this integrated function, we combined these three functions to encapsulate the scaling effect and implementation details for developers. The step-by-step operation is shown in Listing 1 (b).

(3)*vector<double> FloatToVector(double* Fvalue, size_t slot_count)*
The purpose of FloatToVector function is to convert the data of *double* type into *vector* type so that they can be operated with different homomorphic operations. The function is shown as Listing 1 (c). The input parameter $slot\_count$ indicates the size of input data $Fvalue$. The reserve function is to assign $slot\_count$ size memory for the output vector named $Vvalue$. The $push\_back$ function allocates the value of the input to the $Vvalue$.

B. Aggregate functions

```
(a) dim3 block_dim_conv(CONV_COLS, CONV_ROWS,
CONV1_OUTPUT_CHANNELS)

(b) dim3 block_dim_fc1(d_dims[3], d_dims[2],
d_dims[1])

(c) dim3 block_dim_mm(HALF_TILE_SIZE,
HALF_TILE_SIZE, layer)

(d) dim3 block_dim_avg1(b_dims[2], b_dims[3],
layers)
```

Listing 2: Aggregate GPU Mapping Functions

GPU programming involves the mapping between operating data and threads. On top of basic functions, certain aggregate functions are designed for different applications so that developers can easily implement GPU-based applications with homomorphic encryption.

To illustrate the need for such functions, consider the Convolutional Neural Network (CNN). Before the first convolutional layer, we need to transform the provided dataset or intermediate data to proper dimensions so that it be can properly be mapped to different threads for GPU programming. The function $block\_dim\_conv$ is designed to transform the dimensions of input dataset into the dimensions of the first convolutional layer, as shown in Listing 2 (a). $CONV\_COLS$, $CONV\_ROWS$, $CONV1\_OUTPUT\_CHANNELS$ indicates the dimensions of column, row and channel of the first convolutional layer. In order to transform the dimensions of intermediate data for the first fully convolutional layer, another function $block\_dim\_fc1(d\_dims[3], d\_dims[2], d\_dims[1])$ is designed listed in Listing 2 (b). The parameter $d\_dims[3]$, $d\_dims[2]$, $d\_dims[1]$ indicates the the dimensions of column, row and channel of first fully convolutional layer, respectively. When considering the multiplication operation, the designed mapping is shown in List 2 (c). $HALF\_TILE\_SIZE$ indicates the tile size for multiplication and layer is the number of the layer in that CNN layer. The designed mapping function for average pooling is shown in List 2 (d). $b\_dims[2]$, $b\_dims[3]$ and $layers$ indicate the dimension size of kernel size.

IV. Secure Data Analytics Applications

We now explain how we have used the functions introduced in the last section to develop three key applications.

A. CNN Application

Convolution Neural Network (CNN) has been applied in many scenarios related to object recognition, such as ImageNet classification and face recognition [10], as it exhibits good classification performance. To implement this application, the following functionality was developed for a GPU.

Preprocessing Layer: Before the first convolutional layer, the dimensions of the provided dataset have to be transformed to map to different threads for GPU programming. In this function there are no data operations such as addition or multiplication involved. Threads work concurrently on $BLOCK\_SIZE$ to unroll the matrix and there is no communication needed between blocks for unrolling the data.

Convolutional Layer: A convolutional layer is a set of filters that operates on the input points. For the first layer, the input is the raw image. This step only includes addition and multiplication and we can use the same computation over the encrypted data. The function $matrix\_multiplication$ is designed as a kernel function for tile-based matrix multiplication. The parameter $matrix\_b$ is a pointer to the b_unroll vector, $matrix\_c$ is a pointer to the c vector and $start$ is the first number's index of the current batch. Each block deals with a part of a number. Block index in the x direction is for one of these part blocks and block index in y direction is for column indices. Registers are used to store common indices.

Activation Layer: Every activation function takes a single number and performs a certain fixed mathematical operation on it. There are several activation functions we may encounter in practice including $ReLU(ReLU(x) = max(0, x))$, $Sigmoid(\sigma = \frac{1}{1+e^{-x}})$, and $Tanh(2\sigma(2x) - 1)$ functions. We cannot calculate these functions over encrypted values and we should find replacements for these functions that only include addition and multiplication operations. To make it simpler and easier, the square function is used for the activation function in our implementation.

Pooling Layer: After an activation layer, a pooling layer is designed. Two of the most popular pooling layers are max pooling and average pooling. We cannot use max pooling because of the lack of max operation over encrypted data. We use a scaled-up version of average pooling [7] to calculate the summation of values without dividing them by the number of values. We implement average pooling with addition only. The main computation is averaging the results of matrix multiplication. In the algorithm, we first locate the index of the number this thread is averaging, then loop through all channels and calculate the average.

Fully Connected Layer: The fully connected layer has the same structure as hidden layers in classic neural networks. The output of each neuron is the dot product of two vectors: the output of neurons in the previous layers and the related weight for each neuron. This is implemented through a kernel function for tile-based matrix multiplication specialized for fully forward.

In our work, a simple CNN model is built and trained in advance to classify the gender of the person in the image – this training is done using the public ***IMDB-WIKI*** dataset [17] in our experiments. The CNN architecture that is used for our experiments contains two convolution layers followed by pooling layers and two fully connected layers. In the first convolutional layer, 32×32 input is processed by 16 filters of size 5×5 pixels, followed by a square function. Next is the first pooling layer where $16 \times 28 \times 28$ output of the previous filter task is processed to take the average value of 2×2 regions with a two-pixel stride. In the second convolutional layer, $16 \times 14 \times 14$ input processed by 32 filters of size 5×5 pixels, again followed by a square activation function; whereas in the second pooling layer, $32 \times 10 \times 10$ output of the previous filter task is processed to take the average value again with the same hyper-parameters as above. Next, the first fully connected layer receives the $32 \times 5 \times 5$ output of the previous filter task and contains 256 neurons, followed by a square activation function. Finally, the second fully connected layer receives the 256-dimensional output of the previous filter task and contains only 2 neurons that map to the final classes for gender.

B. K-means Application

The HE k-means clustering task uses the typical squared Euclidean distance as the similarity metric, as shown in Algorithm 1. Apart from the initialization phase, the algorithm (as well as its non-encrypted counterpart) involves four phases: encryption, assignment, update, and termination. At first, k data points are selected at random and assigned as initial data centers $c_1, ..., c_k$ during the initialization phase. In the encryption phase, m data points are encoded and encrypted for the following homomorphic encryption. Then the algorithm computes the squared Euclidean distance between each data point and each data center.

The algorithm of ***SquaredEuclideanDistance*** is listed in Algorithm 2, showing how to get the squared Euclidean distance through a series of homomorphic encryption operations. At first, we applied ***Multiply_Plain*** function designed in List 1 (a) to compute the multiplication between ciphertext t_2 and plaintext -1 and assign the result to t_2. Then the addition between t_1 and t_2 is computed through basic add function. At last square function is used to get the square result between t_1 and t_2. Then each data point is assigned to the closest data center c'_h. Then the new data centers $c'_1, ..., c'_k$ will be updated based on their assigned data points. Finally, in the termination phase, the algorithm verifies whether the number of iteration satisfies the pre-defined termination condition. If not, the algorithm continues to the next iteration with the new clusters as input.

Since the CKKS scheme of Seal library supports the addition and multiplications on encrypted real numbers, it is easy for us to compute the Euclidean distance between encrypted data points and data centers. It is important to note that the algorithm also involves the comparison operation in the assignment phase to find the closest data center for each data point, which is not supported by the CKKS scheme. CAM protocol [21] is adopted to solve this problem. After computing the Euclidean distance, the CAM protocol first decrypts the received encrypted Euclidean distance to obtain the plaintexts Pst_{ij}. A new assignment matrix of $m \times k$ is created with all entries set to be zeros. It runs the ***Min(.)*** to compare Pst_{ij} for $j \in [k]$ and find the minimum distance Pst_{ih}. The corresponding value in the assignment matrix (i^{th} row and h^{th} column) is marked to one to indicate the belonging of data point t_i. With the help of assignment matrix, the assigned data points for each data center can be easily known and the new k encrypted data centers are computed on those encrypted data points. The unencrypted iteration is also used to help to verify whether the algorithm should be terminated or continued.

Algorithm 1 Encrypted Version of K-means Clustering

Input: m data points $t_1, ..., t_m$ and iteration n
Initialization: Select k data points randomly and assign them as initial data centers $c_1, ..., c_k$
{Encryption Phase}
Encode and encrypt m data points
{Assignment Phase}
for i = 1 to m do
  for j = 1 to k do
  SquaredEuclideanDistance(t_i, c_j)
  end for
end for
  CAM protocol
{Update Phase}
for j = 1 to k do
  Compute cluster center for c'_j
end for
{Termination Phase}
for j = 1 to k do
  $c_j \leftarrow c'_j$
end for
if iteration < n
  iteration ++
  Go to Step 1
end if

Algorithm 2 Squared Euclidean Distance Computation with Encryption

Input: two data points t_1, t_2
Output: Squared distance of t_1, t_2
$t_2 \rightarrow$ Multiply_Plain($t_2, -1$)
evaluator.add(t_1, t_2,result)
square(result, result2)
relinearize_inplace(result2, relin_keys)
rescale_to_next_inplace(result2)

Algorithm 3 Encrypted Version of KNN

Input: m data points $t_1, ..., t_m$ and test data point p
Output: class
Encode and encrypt m data points, test data point p
Set K
for i = 1 to m do
  SquaredEuclideanDistance(t_i, p_j)
end for
Find K nearest neighbors using CAM protocol $P = \{p_1, p_2, ..., p_k\}$ for $1 \leq k \leq m$
Identify the the p using k nearest neighbor's method.
Compute cluster center for c'_j

C. KNN Application

The goal of KNN is to identify k objects that are nearest to the given test point. The encryption-based algorithm of KNN is shown in Algorithm 3. For selecting a proper k value, the Euclidean distance is computed between the test data point and each training data point using Algorithm 2.

D. Optimization

We will talk about the specific optimization techniques for the secure data analytics applications discussed.

Reduction of Scaling operation: The CNN application involves lots of addition and multiplication operations. After each addition or multiplication, we must scale down the ciphertext. However, if we frequently recall the scaling operation, it would take more time than the time consumed to scale down the ciphertext only once in the end.

If we had an initial vector of values z, it is multiplied by a scale Δ during encoding to keep some level of precision. So the underlying value contained in the plaintext μ and ciphertext c is $\Delta \times z$. So the problem when we multiply two ciphertexts c and c' is that the result holds the result $z \times z' \times \Delta^2$. So it contains the square of the scale which might lead to overflow after a few multiplications as the scale might grow exponentially. Suppose we must do L multiplications, with a scale Δ, then we will define q as:

$$q = \Delta^L \times q_0 \tag{2}$$

Once we have chosen the number L of multiplications we want to perform and set q accordingly, it is easy to define the rescaling operation – we simply divide and round our ciphertext.

V. Experiment Results

All experiments are conducted on the Pitzer cluster of Ohio Supercomputer Center. All single GPU experiments (Dual Intel Xeon 8268s) were involved in dual NVIDIA Volta V100 with 32GB GPU memory. The configuration for the CPU environment is Dual Intel Xeon 8268s Cascade Lakes, which has 48 cores per node.

A. CNN

In this application, we treated 100 images as one data chunk. GPU can provide significant speedups over CPU with or without encryption. HE has high costs, as reported in various earlier studies, but GPUs can help reduce their costs. As shown in Figure 2, without any encryption technique, the speedup of GPU over CPU is 190x. After using the CKKS scheme to encrypt the data, the speedup of GPU over CPU still achieves 81x. GPU implementation involves many relinearization and rescaling operations. During matrix multiplication, rather than executing relinearization and rescaling operations each time after ciphertext-ciphertext multiplication, we delay part of relinearization and rescaling operations to reduce their cost – we scaled the data every 5 iterations. The performance of CNN application on GPU increases 28% after utilizing this mechanism.

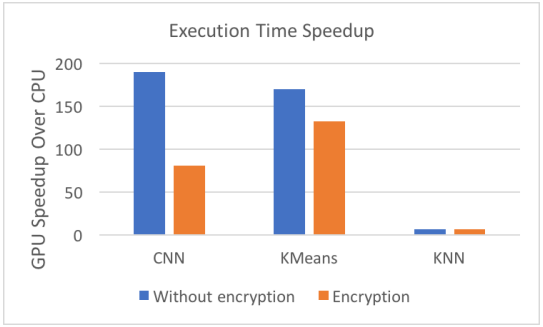

Fig. 2: GPU versus CPU Speedup

B. KMeans

In the KMeans application, the number of data points is 100, whose number of attributes is 288. The number of iterations in the algorithm is set to 100. GPU performs better with or without encryption. As shown in Figure 2, without any encryption technique, the speedup of GPU to CPU is 170x. After using the CKKS scheme to encrypt the data, the speedup of GPU to CPU still achieves 133x. The main reason that its homomorphic encryption speedup performance is much better than CNN's is the use of CAM protocol. By using CAM protocol, a part of the computation is performed directly on unencrypted data, which reduces the size of encrypted data greatly.

C. KNN

In the KNN application, each data chunk has 10000 points, which have three different labels. The value of k chosen in our experiment is three. As shown in Figure 2, the speedup of GPU to CPU with or without any encryption technique is 7x and 6.7x, respectively. We also applied our optimization technique to this application and it achieves 11% reduction on its execution time.

VI. Conclusion

Security and privacy are the major issues concerning the clients as well as the providers of cloud services as a lot of confidential and sensitive data are stored in the cloud which can provide valuable information to an attacker. In this paper, our goal has been to help address performance issues associated with the use of Homomorphic Encryption, while also alleviating the programming difficulties. For this purpose, secure functions for GPU-based processing of data analytics applications are developed, starting on top of the SEAL library from Microsoft. Three secure data analytics applications are implemented, including CNN application, KMeans application and KNN application using CKKS homomorphic encryption. We evaluated the performance of our implementations with both CPU and GPU and achieved up to 81x, 133x and 7x speedup by using GPUs for CNN, KMenas and KNN respectively. Our operation optimizations also help us achieve up to 28% and 11% execution reduction of CNN and KNN, respectively.

Acknowledgements: This work was partially supported by the following NSF grants:1629392, 2007793, 2034850, 2131509, and 2018627.

References

[1] Abbas Acar, Hidayet Aksu, A. Selcuk Uluagac, and Mauro Conti. A survey on homomorphic encryption schemes: Theory and implementation. *ACM Comput. Surv.*, 51(4), July 2018.

[2] Zvika Brakerski and Vinod Vaikuntanathan. Fully homomorphic encryption from ring-lwe and security for key dependent messages. In Phillip Rogaway, editor, *Advances in Cryptology – CRYPTO 2011*, pages 505–524, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[3] Hao Chen, Kim Laine, and Rachel Player. Simple encrypted arithmetic library - seal v2.1. In Michael Brenner, Kurt Rohloff, Joseph Bonneau, Andrew Miller, Peter Y.A. Ryan, Vanessa Teague, Andrea Bracciali, Massimiliano Sala, Federico Pintore, and Markus Jakobsson, editors, *Financial Cryptography and Data Security*, pages 3–18, Cham, 2017. Springer International Publishing.

[4] Jung Hee Cheon, Andrey Kim, Miran Kim, and Yongsoo Song. Homomorphic encryption for arithmetic of approximate numbers. In Tsuyoshi Takagi and Thomas Peyrin, editors, *Advances in Cryptology – ASIACRYPT 2017*, pages 409–437, Cham, 2017. Springer International Publishing.

[5] Ewa Deelman, Karan Vahi, Mats Rynge, Gideon Juve, Rajiv Mayani, and Rafael Ferreira Da Silva. Pegasus in the cloud: Science automation through workflow technologies. *IEEE Internet Computing*, 20(1):70–76, 2016.

[6] Craig Gentry. *A Fully Homomorphic Encryption Scheme*. PhD thesis, Stanford, CA, USA, 2009. AAI3382729.

[7] Ran Gilad-Bachrach, Nathan Dowlin, Kim Laine, Kristin Lauter, Michael Naehrig, and John Wernsing. Cryptonets: Applying neural networks to encrypted data with high throughput and accuracy. In Maria Florina Balcan and Kilian Q. Weinberger, editors, *Proceedings of The 33rd International Conference on Machine Learning*, volume 48 of *Proceedings of Machine Learning Research*, pages 201–210, New York, New York, USA, 20–22 Jun 2016. PMLR.

[8] Lori M Kaufman. Data security in the world of cloud computing. *IEEE Security & Privacy*, 7(4):61–64, 2009.

[9] Kate Keahey, Renato Figueiredo, Jos Fortes, Tim Freeman, and Maurcio Tsugawa. Science clouds: Early experiences in cloud computing for scientific applications. *Cloud computing and applications*, 2008:825–830, 2008.

[10] S. Lawrence, C. L. Giles, Ah Chung Tsoi, and A. D. Back. Face recognition: A convolutional neural-network approach. *Trans. Neur. Netw.*, 8(1):98–113, January 1997.

[11] Qin Liu, Guojun Wang, and Jie Wu. Time-based proxy re-encryption scheme for secure data sharing in a cloud environment. *Information sciences*, 258:355–370, 2014.

[12] A. QaisarAhmadAlBadawi, J. Chao, J. Lin, C. F. Mun, S. J. Jie, B. H. M. Tan, X. Nan, A. M. M. Khin, and V. Chandrasekhar. Towards the alexnet moment for homomorphic encryption: Hcnn, the first homomorphic cnn on encrypted data with gpus. *IEEE Transactions on Emerging Topics in Computing*, pages 1–1, 2020.

[13] Muthu Ramachandran and Victor Chang. Towards performance evaluation of cloud service providers for cloud data security. *International Journal of Information Management*, 36(4):618–625, 2016.

[14] R Velumadhava Rao and K Selvamani. Data security challenges and its solutions in cloud computing. *Procedia Computer Science*, 48:204–209, 2015.

[15] Oded Regev. On lattices, learning with errors, random linear codes, and cryptography. *J. ACM*, 56(6), September 2009.

[16] R L Rivest, L Adleman, and M L Dertouzos. On data banks and privacy homomorphisms. *Foundations of Secure Computation, Academia Press*, pages 169–179, 1978.

[17] Rasmus Rothe, Radu Timofte, and Luc Van Gool. Deep expectation of real and apparent age from a single image without facial landmarks. *International Journal of Computer Vision (IJCV)*, July 2016.

[18] Microsoft SEAL (release 3.6). https://github.com/Microsoft/SEAL, November 2020. Microsoft Research, Redmond, WA.

[19] Danan Thilakanathan, Shiping Chen, Surya Nepal, and Rafael A Calvo. Secure data sharing in the cloud. In *Security, privacy and trust in cloud systems*, pages 45–72. Springer, 2014.

[20] Paul Watson, Phillip Lord, Frank Gibson, Panayiotis Periorellis, and Georgios Pitsilis. Cloud computing for e-science with carmen. In *2nd Iberian Grid Infrastructure Conference Proceedings*, pages 3–14. Citeseer, 2008.

[21] W. Wu, J. Liu, H. Wang, J. Hao, and M. Xian. Secure and efficient outsourced k-means clustering using fully homomorphic encryption with ciphertext packing technique. *IEEE Transactions on Knowledge and Data Engineering*, pages 1–1, 2020.

[22] QI Xia, Emmanuel Boateng Sifah, Kwame Omono Asamoah, Jianbin Gao, Xiaojiang Du, and Mohsen Guizani. Medshare: Trust-less medical data sharing among cloud service providers via blockchain. *IEEE Access*, 5:14757–14767, 2017.

[23] Xiaojun Yu and Qiaoyan Wen. A view about cloud data security from data life cycle. In *2010 international conference on computational intelligence and software engineering*, pages 1–4. IEEE, 2010.

[24] Gansen Zhao, Chunming Rong, Jin Li, Feng Zhang, and Yong Tang. Trusted data sharing over untrusted cloud storage providers. In *2nd IEEE International Conference on Cloud Computing Technology and Science*, pages 97–103. IEEE, 2010.