Soft Computing (2021) 25:14413–14428
https://doi.org/10.1007/s00500-021-06225-y

OPTIMIZATION

GPU-Accelerated implementation of a genetically optimized image encryption algorithm

Brijgopal Bharadwaj[1] · J. Saira Banu[1] · M. Madiajagan[1] · Muhammad Rukunuddin Ghalib[1] · Oscar Castillo[2] · Achyut Shankar[3]

Accepted: 30 August 2021 / Published online: 30 September 2021

Abstract

This paper presents a GPU-accelerated implementation of an image encryption algorithm. The algorithm uses the concepts of a modified XOR cipher to encrypt and decrypt the images, with an encryption pad, generated using the shared secret key and some initialization vectors. It uses a genetically optimized pseudo-random generator that outputs a stream of random bytes of the specified length. The proposed algorithm is subjected to a number of theoretical, experimental, and mathematical analyses, to examine its performance and security against a number of possible attacks, using the following metrics - histogram analysis, correlation analysis, information entropy analysis, NPCR and UACI. The performance analysis carried out shows an average speedup-ratio of 3.489 for encryption, and 4.055 for decryption operation, between the serial and parallel implementations using GPU. The algorithm aims to provide better performance benchmarks, which can significantly improve the experience in the relevant use-cases, like real-time media applications.

Keywords Pseudo-random generator · GPU · CUDA programming · Symmetric key · Image encryption · Genetic optimization

1 Introduction

Cryptography has been an integral part of the human communication process, since old ages. It serves the vital need of exchanging ideas and information with only a certain group of people, via the usage of ingenious ways to filter out the common people from this subset of peers. But the field has been forced to evolve constantly, as the cryptanalysis methods become more sophisticated and efficient at breaking the ciphers used for encryption. This has especially been true, since the internet revolution that came around in the 1960s, and the world was made aware about the wonders a computing device could do. (D'agapeyeff 2016; Davies 1997) The evolution of computers and E-services has been the pivotal point in the rapid development of both, cryptography and cryptanalysis, as it allows the computations on the scales that were deemed infeasible in the pre-computer era. (Altigani et al. 2019; Armin et al. 2016; Riek and Böhme 2018).

In the modern world, there has been a significant growth in the applications of the field of cryptography, as the world collectively produces astronomical amounts of data

✉ Achyut Shankar
ashankar1@amity.edu

Brijgopal Bharadwaj
brijgopalbharadwaj1999@gmail.com

J. Saira Banu
jsairabanu@vit.ac.in

M. Madiajagan
madiajagan.m@vit.ac.in

Muhammad Rukunuddin Ghalib
ruk.ghalib@ieee.org

Oscar Castillo
ocastillo@tectijuana.mx

1 School of Computer Science Engineering, Vellore Institute of Technology, Vellore, India

2 Division of Graduate Studies and Research, Tijuana Institute of Technology, Tijuana, Mexico

3 Department of Computer Science & Engineering, ASET, Amity University, Noida, India

on a regular basis. The use-cases for such algorithms span across various domains, like military, social-media, financial sector, IoT, and cloud-based services. (Boritz and No 2005; Stergiou et al. 2018; Kalra and Sood 2015) Particularly in the instances where the data being dealt with is of image/video domain, cryptography and security are one of the first concerns raised by the people, governments, and law authorities alike. The shear amount of data involved in these cases also makes it prudent to have an efficient and secure algorithm for real-time applications. Thus, it suffices to say that there is always scope of improvement. (D'agapeyeff 2016; Davies 1997; Boritz and No 2005) Many applications and algorithms have been developed to address these needs, and this paper also follows suite.

In this paper, a GPU-accelerated image encryption algorithm has been proposed, that draws from the concepts of the traditional XOR cipher. It uses a genetically-optimized pseudo-random number generator (PRNG) to generate identical encryption and decryption pad on both the ends of the communication, via the usage of a shared secret key and a unique encryption/decryption vector. To achieve this, the algorithm proposes the architecture of a specially designed helical lattice structure, to incorporate cryptographic convolutions in the mathematics involved, in order to guarantee unidirectional attributes. The involvement of GPU in the algorithm allows it to make use of the parallel-computing architecture. This improves the performance of the paradigm's parts that follow the SIMD structure, with respect to the traditional approaches that have been employed to solve the same problem. (Antwerpen 2011; Zafar et al. 2010) The paper also presents a genetic optimization exercise in detail, that was performed to enhance the security metrics of the PRNG, by the process of hyper-parameter tuning. The proposed algorithm aims to provide better performance benchmarks than the compared counterparts, to present a smoother and a much-more seamless experience for the users of the application.

The contributions of this paper are listed in Sect. 2. Section 3 presents a brief summary of the currently available literature pertinent to the field of research of this paper, and also describes the notable contributions that were referred to during the process of research. Section 4 provides a detailed description of the algorithm being proposed here, and Sect. 5 provides the algorithm followed to optimize the latter, using the concepts of genetic evolution. Section 6 provides the experimental analysis carried out to quantify the effectiveness and efficiency of the proposed algorithm, followed by a detailed security analysis in Sect. 7. Section 8 concludes the presented work, followed by the relevant references.

2 Contributions

This paper extends upon the ideas introduced by the encryption paradigm proposed in Bharadwaj and Sairabanu (2020) by incorporating the SIMD computing architecture into it, for achieving superior efficiency benchmarks. It also presents a genetic optimization of the hyper-parameters involved in the PRNG unit, to maximize the security measures of the proposed algorithm. A comparative analysis is also included with some of the previously proposed algorithms, and using quantifiable mathematical metrics, demonstrates the performance enhancements. The improvements introduced in this paper make the proposed algorithm a lot more suitable for modern-day high-speed applications, without compromising on security parameters. The possible applications include high-speed transmission in video-conferencing and streaming applications, IoT infrastructure, media-major social-media platforms, speed-sensitive financial applications, etc.

3 Literature survey

The encryption ciphers have been a subject of active research, since the internet revolution came around. Many such ciphers rely on pseudo-random number generators to operate, in one way or another. Thus, it is essential that the PRNG being used by the system, is cryptographically secure and efficient. Mishra et al. (Mishra and Mankar 2015) propose algorithms for the encryption and decryption of textual data, using pseudo-random generator, and linear congruential generator. They also analyse its performance using various security metrics, and also show the algorithm in action, by considering instances of popular attacks on ciphers. Almalkawi et al. (Almalkawi et al. 2019) present an image encryption algorithm, that is designed to be useful for the wireless network applications, that require the PRNG to be lightweight and efficient. They make use of the hybrid chaotic systems in the design of the PRNG to induce the required levels of randomness in the system. Similarly, the algorithm proposed by Ramesh et al. (Ramesh and Jain 2015) uses a two-stage methodology for transforming the images in their encrypted counterparts, by the usage of Altered Sophie Germain Prime (ASGP) based pseudo-random number generator, and the Lehmer Random Number Generator (LRNG), whose outputs are used to perform an XOR and a mapped-swapping transformation for encryption. Bharadwaj et al. (Bharadwaj and Sairabanu 2020; Bharadwaj et al. 2018) propose a symmetric key encryption scheme that uses a pseudo-random generator (PRNG) to encrypt images, using a derivative of the traditional XOR cipher, that tends to its shortcomings via the

application of a one-time-pad architecture, along with a custom one-way function. These algorithms are used as the foundation for the presented algorithm. Zhang et al. (Zhang et al. 2017) provide a fast implementation of the traditional AES algorithm, specifically configured to encrypt and decrypt images, by permuting the first block of plain image with an initial vector, followed by an AES' block-chaining mode implementation to encrypt each image block of 128 bits. The vector and cipher image are transmitted to the other end, where the secret key and the vector are used in conjunction to decrypt the cipher. (Ramesh and Jain 2015; Bharadwaj and Sairabanu 2020) and (Zhang et al. 2017), are used for performing comparative analysis with the proposed algorithm, to obtain better insights regarding performance and security parameters.

The graphics processing unit (GPU) is a massively multi-threaded architecture, containing a number of processing elements. An NVIDIA GPU is composed of a number of threads, which are grouped into blocks, which themselves are grouped into grids, as shown in Fig. 1. Each block can have a maximum number of threads, whose exact number depends on the GPU version. For the modern GPUs, each block is composed of 1024 threads. These threads can be accessed via a 1D, 2D or 3D indexing scheme, where the programmer is free to choose the scheme that shows most natural relationship with the problem at hand. Similarly, all the blocks in a grid can be accessed via a 1D, 2D, or 3D indexing scheme, the choice of which is up to the programmer. (Buck 2007; NVIDIA 2020; Zeller 2011).

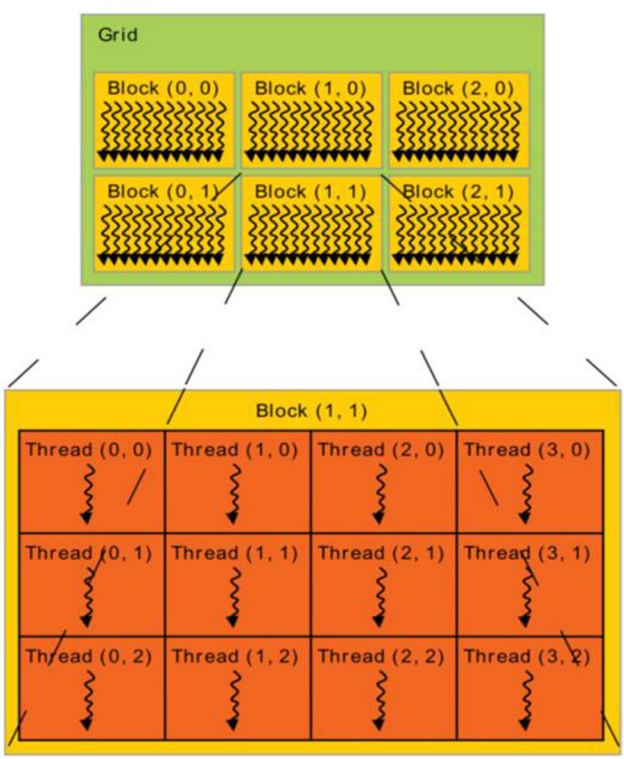

Fig. 1 Indexing in GPU, taken from (NVIDIA 2020)

Until recently, the use-cases of GPU were related to just graphics and rendering related operations. But the introduction of the general-purpose GPUs (GPGPUs) opened up new frontiers for the compute-intensive problems, that have inherent parallelism in their architecture. They can leverage the power of GPUs to perform non-graphics related operations. One of the most popular frameworks to allow the usage of GPUs as a set of compute engines is the CUDA environment, provided by NVIDIA's CUDA-enabled GPUs. CUDA C + + is an augmented version of the traditional C + + environment, which allows the definition of special functions, often referred to as the kernels, which are executed in parallel by a user-defined number of threads.

The NVCC compiler is used to compile the CUDA programs, which separates the source code into device and host components, where the host components are executed by the standard C + + compilers like GCC, CL, etc. A number of algorithms, utilizing the computational efficiency of GPUs, have been previously proposed. W. K. Lee et al. (Lee et al. 2016) examine the implementation of the block ciphers like AES, CAST, Blowfish, etc., and present techniques to accelerate this process. Manssen et al. (Manssen et al. 2012) provide a review of the existing CUDA-based random-number generators, in the context of the massively parallel simulations, like the Monte Carlo and molecular dynamics simulations. Riesinger et al. (Riesinger et al. 2018) review the non-standard PRNGs for normally distributed random numbers in the context of GPU implementation. These were used as the sources of inspiration, while implementing the SIMD architecture in the proposed algorithm, and were used in analysis and comparison, in order to arrive at the best approach to perform the considered task, and achieve satisfactory results effectively.

Inampudi et al. (Inampudi et al. 2018) present an implementation of the AES algorithm on the GPGPU, using the OpenCL API, via the usage of data decomposition technique, where they divide the workload between the 256 concurrently executing threads of the GPU, and compare their implementation to the serial version of the same algorithm. Sheshadrinathan et al. (Seshadrinathan and Dempski 2008) presented another implementation for AES using the NVIDIA architecture, and their results are used for comparison with the proposed algorithm.

Genetic algorithms are also a field of study, which has often been integrated with other disciplines, such as the encryption systems and pseudo-random generators to obtain better performance and security metrics. Kösemen

et al. (Kösemen et al. 2018) present a pseudo-random generator based on genetic programming to be used in wireless identification platforms. Abdullah et al. (Abdullah et al. 2012) make use of the genetic algorithms in the process of image encryption process, via a chaotic function logistic map. Dutta et al. (Dutta et al. 2014) propose a genetic algorithm based secret key encryption method, that uses genetic mutation along with the *Blum Blum Shub* PRNG function to induce randomness in the output. Sen et al. (Sen et al. 2017) propose an algorithm that uses the genetic algorithm to generate an intermediate key, which is unique for each encryption, which is used to impart a stochastic nature to the encryption process to increase its level of security. Affenzeller et al. (Affenzeller et al. 2009) describe the general architecture of genetic algorithms, which can be used for the purpose of optimization of a given function.

4 Proposed algorithm

Figure 2 presents a schematic diagram of all the involved modules and components, to facilitate the understanding of all the interactions and connections between them.

4.1 Randomization transform

The randomization transform (RT) is a function, represented by f: $\mathbb{R}^2 \rightarrow \mathbb{R}^2$, that accepts two 64-bit variables, R_{prev} and S_{prev}. They are subjected to the operations described in (1.1–1.4), to obtain the output values, R_{new} and S_{new}. It should be noted that the constants A, B, C, and D are hyper-parameters that need to be tuned, in order to obtain optimal results from the considered algorithm, and L is defined to be the last digit obtained from the decimal representation of R_{prev}. The implementation of the RT is presented in Algorithm 1.

$$RT\left(R_{\text{prev}}, S_{\text{prev}}\right) = \left(R_{\text{new}}, S_{\text{new}}\right) \tag{1.1}$$

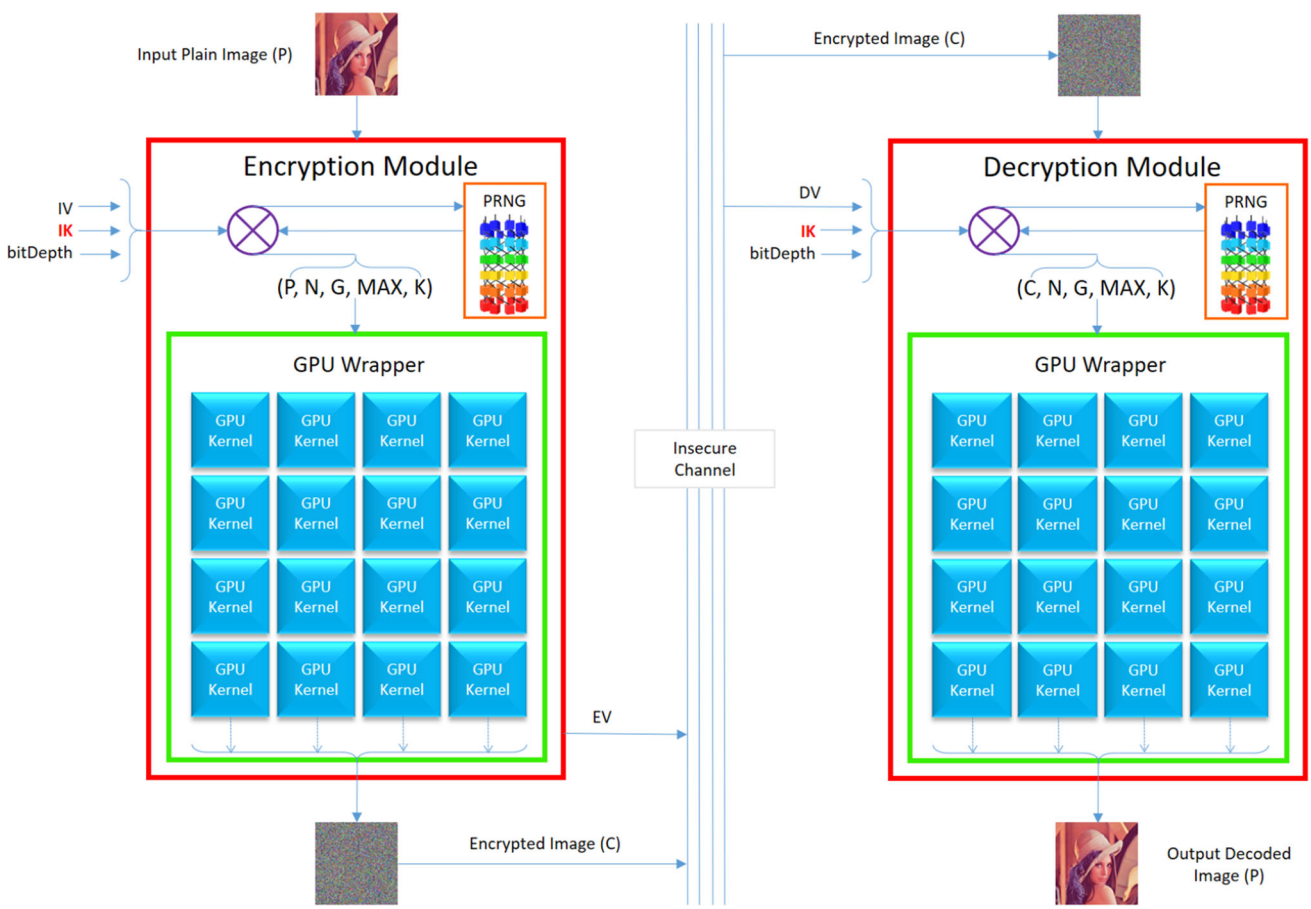

Fig. 2 Schematic diagram of the proposed image encryption algorithm

$$T = (S_{\text{prev}} \% A) \oplus B \tag{1.2}$$

$$S_{\text{new}} = (T \times 2^{\text{L}}) \% C \tag{1.3}$$

$$R_{new} = (R_{prev} \oplus S_{new}) \% D \tag{1.4}$$

Algorithm 1: Randomization Transform (RT)

Parameters: $R_{prev}, S_{prev} \in [0, 2^{64}]$
Initialization: $R_{new} = 0, S_{new} = 0, L = 0$

$L = R_{prev} \% 10$
$T = (S_{prev} \% A) \oplus B$
$S_{new} = (T \times 2^{L}) \% C$
$R_{new} = (R_{prev} \oplus S_{new}) \% D$

Return: R_{new}, S_{new}

4.2 Pseudo-random number generator

The crux of the entire encryption-decryption routine proposed in this paper is the PRNG module, which is responsible for providing a cryptographically-secure set of random-bytes in an efficient way. These bytes are generated such that the same random-bytes can be replicated on the other end of the communication, only when the corresponding correct inputs are provided to it. In order to motivate the internal workings of the PRNG module, a discussion about the underlying geometric structure used for the computation of these random bytes is mandatory.

The structure in question can be visualized as a cylindrical unit comprised of a number of rings stacked on top of each other, where each ring itself is made of cubical computation units. A 3D visualization of the same can be found in Fig. 3. As can be seen, there are two separate family of connections between all the cubical units present in the structure. One family of the connections has an anti-clockwise rotation, associated with each ring it encounters. The other family travels vertically down the structure, in a straight line.

The shape of the considered structure is defined by a set of two parameters: *nSeeds* and *nRings*, which control the number of cubical units per ring, and the total number of rings, respectively. Each cubical unit represents an instance of RT, and the two family of connections described above represent the stream of two inputs and outputs, consumed and produced by each instance of RT, respectively. These connections, visually signify that the S_{new} values produced by the previous ring, are passed to the S_{prev} input of the next ring of the structure. Also, the R_{new} values are shifted by one position in the anti-clockwise direction, before passing on to the R_{prev} inputs of the next ring.

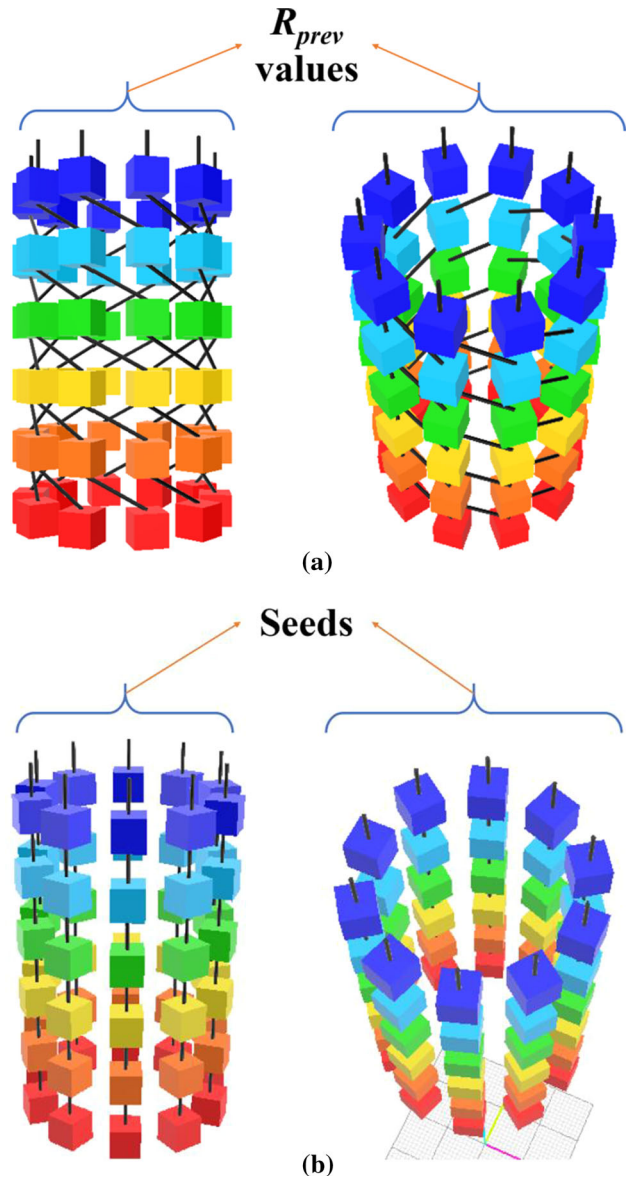

Fig. 3 Value propagation in the PRNG for **a** R_{prev} values and **b** Seeds

This convolution operation is performed, in order to ensure that, even if the attacker somehow discovers the output stream of random bytes for any given process of encryption/decryption, he/she is still unable to compute the values of the initial seed values and the common secret key used for the computation, without resolving to the brute-force approach of trial and error. By extension, the future seed values are also secure, as they can't be obtained from the decoded seeds without the secret key. However, this applies an upper-bound on the degree of parallelism that can be achieved in the involved computations, which can be addressed in future.

The PRNG module expects an input of three parameters: *initKey*, *initVal*, and *N*. *N* signifies the required number of random bytes, and the values of *initKey* and *initVal* are

used to initialize the units of the first ring of the specified structure, using the following approach:

1. The value of *nSeeds*, which controls the number of RT units per ring, is computed using (2).

$$nSeeds = ceil\left(\frac{initKey.length()}{64}\right) \tag{2}$$

2. The value of *nRings* is initialized using (3), to set the number of rings in the given structure.

$$nRings = ceil\left(\frac{N}{nSeeds \times 7}\right) \tag{3}$$

3. The given *initKey* was padded with leading zeroes to make its length a multiple of 64. Then it was divided into '*nSeeds*' number of 64-bit binary strings, to be used as the seeds for initializing the S_{prev} values of the first ring's input.
4. The 64-bit value received as *initVal* is used to initialize the R_{prev} inputs of the first ring.

   Once the structure is fully defined and instantiated, the process of computation is initiated for each ring sequentially. When a given ring receives the inputs from its parent ring, the values are consumed by the RTs to compute the new value-pair. Also, for each RT unit, the obtained value of R_{new} is sampled-out, before passing it on to the next ring. These values are stored in a 2D-array called *Helix*. Once all the *RT* units present in the structure have been sampled in a similar fashion, the algorithm sequentially iterates over all the values recorded in *Helix*. It extracts random bytes from each value, by repeatedly taking a modulus with 255, adding 1, and storing the results obtained in *randBytes*. It should be noted that this operation results in no random byte being 0, thus further ensuring that no pattern-recognition algorithms, such as SVMs and neural-networks can be used to acquire anchor-points for a known-plaintext attack. Using this operation, seven random bytes can be extracted from each 64-bit value present in the *Helix* data-structure. After iterating through all the values, the entire *randBytes* array is returned as the output stream of random bytes. The implementation of the PRNG is presented in Algorithm 2.

Algorithm 2: Pseudo-Random Number Generator

```
Parameters:    initKey, initVal, N
Initialization:  Seed[ ], Helix[ ][ ], R[ ]

bin_to_dec(str):
   d = 0, n = str.length()
   for i = 0, 1, …, n-1 do:
      d += a = (str[i] - '0') * pow(2, n-i-1)
   return d
nSeeds = ceil((initKey.length()/64))
nRings = ceil(N/(7 × nSeeds))

for j = 0, 1, …, nSeeds-1 do:
   Seed[j] = bin_to_dec(initKey.substr(64j, 64))
   Helix[0][j] = RT(Seed[j], initVal)

for i = 1, 2, …, nRings-1 do:
  for j = 0, 1, …, nSeeds-1 do:
    Helix[i][j] = RT(Seed[j], Helix[i-1][(j-1)%nSeeds])
  end
end

for i = 0, 1, …, nRings-1 do:
   for j = 0, 1, …, nSeeds-1 do:
      for j = 0, 1, …, nSeeds-1 do:
         idx = ((i × nSeeds) + j) × 7 + k
         if idx < N:
            R[idx] = (Helix[i][j] % 255) + 1
            Helix[i][j] = Helix[i][j] ≫ 9
         else:
            goto RET
         end
      end
   end
end

RET:    return R
end
```

4.3 Encryption module

To encrypt a group of images, run the following sequence of commands on each one to get the encrypted equivalents as the output. The following operations assume that the input image is in a single-channel grayscale format, although the same techniques can be used to encrypt multi-channel coloured images as well, by treating each channel separately. The implementation is presented in Algorithm 3.

1. Receive the input image *P*, its bit-depth *bitDeph*, along with an initial key *IK* and initialization vector *IV* to be used in the process of encryption.

2. The maximum possible pixel-intensity value *MAX* with the given bit-depth is computed using (4), and stored for future use.

$$MAX = 2^{\text{bitDepth}} - 1 \tag{4}$$

3. The number of random-bytes required to perform encryption of each pixel, referred to as the group-size *G* of the input image is calculated using (5), and is also stored for future use.

$$G = ceil\left(\frac{bitDepth}{8}\right) \tag{5}$$

4. The total number of random-bytes needed to encrypt the entire image *N* are computed in accordance with (6) and recorded.

$$N = rows(P) \times cols(P) \times G \tag{6}$$

5. The encryption-vector *EV* is calculated using (7), by performing a bitwise XOR operation between the value of *IV* and the one's complement of the sum of the pixel intensity-values of *P*.

$$EV = IV \oplus sum(P)' \tag{7}$$

6. Another pixel-matrix *C* is initialized to record the encrypted output for the image *P*, and to be presented as the final output of the image encryption module.
7. A call is made to the pseudo-random number generator *PRNG* to get back the required number of random bytes, by passing on the values of *IK* as *initKey*, the value of *EV* as *initVal*, and the number of pixels required *N*. The received random bytes are stored as the encryption key *K*.
8. The values computed for *K*, *N*, *G*, and *MAX*, along with *P* are used to place a call to the *GPU_wrapper* module, which returns the encrypted image *C*.
9. *C* is finally transmitted on the insecure channel as the output of the entire process, along with EV, to be treated as the encrypted counterpart of input image *P*.

For a secure and successful operation of the encryption module, the algorithm assumes that the two parties involved in this operation, have mutually agreed upon a common value of *IK*. *IK* is assumed to be secret throughout the working span of the presented algorithm. In order to achieve this, a number of existing algorithms, like EC-DHA, RSA, etc. can be utilized. It is also essential that the value of *IV* is unique for each and every run, which ensures that even if the same data is encrypted repeatedly, along with the same key, the obtained encryption outputs C_1, C_2, C_3, etc. are still different. For the purpose of implementation, the value of *IV* was taken to be the number of nanoseconds elapsed, since the Unix epoch. Also note that the value of *IV* and *EV* is not secret, and can be transmitted as plain-text over an insecure channel, along with the encrypted image *C*.

Algorithm 3: Encryption Methodology

```
Parameters: P, IK, IV, bitDepth

MAX = 2^bitDepth − 1
G = ceil(bitDepth / 8)
N = rows(P) × cols(P) × G
EV = ~sum(P) ⊕ IV
K = PRNG(IK, EV, N)
C = GPU_wrapper(P, N, K, G, MAX)

Return: C, EV
```

4.4 Decryption module

The encrypted image *C*, along with the value of *EV* can be broadcasted by the sender over an insecure channel, for the receivers who have a copy of *IK*. These users can process *C* using the following steps, and reproduce the plain-image *P*.

1. Receive the encrypted image *C*, its bit-depth *bitDeph*, along with the initial key *IK* and value of EV as the decryption vector *DV*.
2. The maximum possible pixel-intensity value *MAX* with the given bit-depth is computed using (4), and stored for future use.
3. The number of random-bytes required to perform decryption of each pixel, referred to as the group-size *G* of the input image is calculated using (5), and is also stored for future use.
4. The total number of random-bytes needed to decrypt the entire image, *N,* are computed in accordance with (6) by replacing *P* with *C*, and recorded.
5. Another pixel-matrix *P* is initialized to record the decrypted output for the image *C*, and to be presented as the final output of the image decryption module.
6. A call is made to the pseudo-random number generator *PRNG* to get back the required number of random bytes, by passing on the values of *IK* as *initKey*, the value of *DV*, and the number of pixels required *N*. The received random bytes are stored as the encryption key *K*.
7. The values computed for *K*, *N*, *G*, and *MAX*, along with *C* are used to place a call to the *GPU_wrapper* module, which returns the encrypted image *C*.
8. *P* is finally returned as the output of the entire process, to be treated as the decrypted counterpart of the received image *C*.

Algorithm 4: Decryption Methodology

```
Parameters: C, IK, DV, bitDepth

MAX = 2^bitDepth − 1
G = ceil(bitDepth / 8)
N = rows(C) × cols(C) × G
K = PRNG(IK, DV, N)
P = GPU_wrapper(C, N, K, G, MAX)

Return: P
```

The application of the above-mentioned algorithm, effectively nullifies the modifications introduced by the encryption process, thus providing the receiver with the exact copy of the initially encrypted image. Algorithm 4 presents the implementation of the decryption module.

4.5 GPU_wrapper

This module is an assistant module to the next one presented below, and is intended to perform the low-level operations like memory management and computations to define the structure of the calls to the GPU hardware. It accepts the input of the image *img*, secret-key *K*, *N*, *grpSz*, and *MAX*. As discussed previously, the GPU has two parameters associated with each kernel call, the number of threads per block, and the number of blocks per grid. For the GPU used in the testing of the presented algorithm, each block can have a maximum of 1024 active threads, and that is the value we use for each block. As for the number of blocks to be used, we calculate that using (8), after which the process of preparing the GPU to perform the required computation is initiated.

$$\begin{matrix} Blocks \\ Per\ Grid \end{matrix} = ceil\left(\frac{N}{grpSz \times \begin{matrix} Threads \\ Per\ Block \end{matrix}}\right) \tag{8}$$

First, we reserve the memory for the last four variables accepted from the parent function, by making corresponding calls to the *cudaMalloc* method of the CUDA interface. After that, we copy the actual contents of these variables to the newly allocated space within the GPU environment, via the *cudaMemcpy* method. We upload the input image to an object of the OpenCV's *GpuMat* class, the *mat* equivalent in the GPU environment, by the *upload* method. All the variables instantiated in GPU are passed on to the *kernel,* where the actual computation takes place, after which we extract the transformed image using the *download* method of the *GpuMat* class. Finally, all the allocated memory in the GPU is systematically freed up, and the transformed image is returned implicitly, by the *img* reference. Algorithm 5 presents the discussed operations in detail.

4.6 GPU Kernel

The *kernel* is the piece of the algorithm, which is executed within the GPU environment. It is responsible for performing a pixel-wise XOR operation between the pixels of the input image, and the *pad* generated from the '*grpSz*' number of random bytes, referred to as the key. As inputs, it accepts the key *K*, the pixel-data of the image as *src*, the number of rows and columns as *rows* and *cols* respectively, the distance between successive rows in bytes as *step*, *grpSz* and *MAX*. It is executed by all the threads, running parallel to each other in all the blocks activated by the *GPU_wrapper*. Using *kernel*'s instructions, each thread starts off by initializing the pad to be 0. After that, each thread determines the coordinates of the image-pixel it is supposed to be operating on, using (9.1–9.3). It should be noted that *blockIdx*, *blockDim*, and *threadIdx* are built-in vectors offered by the GPU environment, to allow easy-access to the structure of the computing environment and the identity of the thread in question.

$$idx = blockIdx.x * blockDim.x + threadIdx.x \tag{9.1}$$

$$x = floor\left(\frac{idx}{cols}\right) \tag{9.2}$$

$$y = idx - (x * cols) \tag{9.3}$$

Algorithm 5: GPU_Wrapper

```
Parameters: img, K, N, grpSz, MAX
Initialization: d_K=d_MAX=d_img=d_grpSz=NULL

tpb = 1024
bpg = ceil(N / (tpb×grpSz))

cudaMalloc((void**) &d_K, sizeof(K))
cudaMalloc((void**) &d_MAX, sizeof(MAX))
cudaMalloc((void**) &d_grpSz, sizeof(grpSz))

cudaMemcpy(d_K, &K[0], sizeof(K))
cudaMemcpy(d_MAX, &MAX, sizeof(MAX))
cudaMemcpy(d_grpSz, &grpSz, sizeof(grpSz))
d_img.upload(img)

GPU_kernel <<<bpg, tpb>>> (d_K, d_img.data,
        … d_img.rows, d_img.cols, d_img.step,
        … d_grpSz, d_MAX)

d_img.download(img)
cudaFree(d_K)
cudaFree(d_MAX)
cudaFree(d_grpSz)

return img
```

GPU_Kernel's implementation is discussed in Algorithm 6. It computes the pad, by appending the consecutive '*grpSz*' number of bytes together. This pad is then subjected to a bitwise *AND* operation with *MAX*, to make its bit-length comparable to the pixel value. It is then *XOR*ed to the output of the previous operation to obtain the transformed pixel-value. *GPU_Kernel* does not explicitly return any parameter, as all the necessary parameters were passed referentially, and hence retain their desired values after execution of any considered *GPU_Kernel* module. Once all the threads in GPU complete their execution, the control is transferred back to the *GPU_wrapper*.

Algorithm 6: GPU_Kernel

Parameters: *K, src, rows, cols, step, grpSz, MAX*
Initialization: *pad* = 0

idx = *blockIdx.x* × *blockDim.x* + *threadIdx.x*
$x = floor\left(\frac{idx}{cols}\right)$
$y = idx - (x * cols)$
$rowptr = src + (x \times step)$
for *k* in 0, 1, …, *grpSz*-1 **do**:
  $pad = pad \mid K[idx \times rpSz + k] << k$
end
$rowptr[y] = rowptr[y] \oplus (pad \,\&\, MAX)$

5 Genetic optimization

As described in the previous section, the operation of the Randomization Transform (RT) depends upon the value of four hyper-parameters, namely A, B, C, and D. In order to optimize the performance of RT, so that it produces a more uniform frequency distribution of the random-bytes being generated by the PRNG, we adopted the process of genetic mutation to obtain the corresponding values for a better, more uniform distribution of the random values. In order to allow the required experimentation, the structure of RT was modified in such a way that it accepts another input parameter, *params*, that is a vector with four 64-bit elements. Each member of *params* stores the value supplied for the corresponding hyper-parameter. Similar modifications were made to the PRNG module, so that it can also receive the *params* vector, and pass it on to RT. In addition to this, some methods for defining the structure of the genetic algorithm were also created. The overall structure and ordering of the used genetic algorithm is described in Algorithm 7.1–7.5.

The *cal_population_fitness_* method was defined to accept a given population, make a call to the PRNG with each offspring's genes as the value for the *params* vector, and return a corresponding fitness vector, describing how desirable is each chromosome in a given population. In order to calculate the fitness of a given chromosome, the frequency distribution of the returned sequence of *N* random numbers is considered, and its coefficient of variation (*COV*) is used as the fitness metric.

The *select_mating_pool* method selects '*num_parents*' number of chromosomes from the population, with the lowest *COV* values to be the parents for the next generation, as these present the best chance of resulting in a uniform frequency distribution. The *crossover* method produces a set of *offsprings* by randomly choosing a crossover point, and interchanging all the genes lying after it with those of the cyclically next parent.

Algorithm 7.1: Cal_Population_Fitness

Parameters: *population*
Initialization: *fitness[ ]*

for *i* = 0, 1, …, *population.length*-1 **do**:
  R = PRNG(*IK, IV, N, population*[*i*])
  values, counts = histogram(*R*)
  fitness[*i*] = COV(*counts*)
end
Return *fitness*

Algorithm 7.2: Select_Mating_Pool

Parameters: *population, fitness, num_parents*
Initialization: *parents[ ][ ]*

for *i* = 0, 1, …, *num_parents*-1 **do**:
  min_fitness_idx = where*(fitness* == min*(fitness))*
  parents[i] = *population*[*min_fitness_idx*]
  fitness[*min_fitness_idx*] = *INF*
end

Return *parents*

The *mutation* method receives the *offsprings* produced by the *crossover* method, along with the maximum number of mutations that can be induced in the *offsprings*. It iterates over all the chromosomes, and for each of them, generates a vector of random numbers between 0 and 1. If the obtained number for a given gene is greater than 0.75, we modify the gene by assigning a random value of similar upper bound to it. Considering the structure of RT, the lower and upper bounds of the members of the *params* vector, were set accordingly. Once all the chromosomes are subjected to the same process, the obtained *offsprings* are returned as output.

6 Experimental results

To visually observe the results obtained on performing the operation of encryption and decryption using the proposed algorithm, we encrypted and decrypted the standard test images as shown in Fig. 4. The values of *IV* and *IK* used for this process are listed in Tables 1 and 2 respectively. As can be seen, there has been no loss of visual information in the entire encryption-decryption process.

The genetic algorithm described in Algorithm 7 was run multiple times, and the obtained best values for the *params* vector, along with their improvement graphs can be seen in Table 3 and Fig. 5 respectively. The *COV* value obtained for the best output of the genetic algorithm was observed to be 0.018369, which is much better than the *COV* value obtained for the algorithm proposed in Bharadwaj and Sairabanu (2020), which was observed to be 0.069726. This difference results in a smoother histogram for the proposed algorithm, and hence a higher security standard.

Algorithm 7.3: Crossover

```
Parameters: parents, offspring_size, nGenes
Initialization: offspring[ ][ ]

for i = 0, 1, …, offspring_size-1 do:
   p1 = rand_int(parents.length)
   p2 = rand_int(parents.length)
   if p1== p2 do:
      p2 = (p1+1)% parents.length
   End
   crs_pt = rand_int(nGenes)
   offspring[i][0:crs_pt] = parents[p1][0:crs_pt]
   offspring[i][crs_pt : ] = parents[p2][crs_pt : ]
end

Return offspring
```

Algorithm 7.4: Mutation

```
Parameters: crossover_offspring, nMutations,
            LB, UB, offspring_size, nGenes

for i = 0, 1, …, offspring_size-1 do:
  for j = 0, 1, …, nGenes-1 do:
     p = rand()
     ctr = nMutations
     if p>0.75 and ctr>0 do:
        crossover_offspring[i][j] = rand_int(LB[j],UB[j])
        ctr = ctr – 1
     end
  end
end

Return crossover_ffspring
```

Algorithm 7.5: Genetic Algorithm

```
Initialization: N_GENES = 4
                N_PARENTS = 5
                N_OFFSPRINGS = 5
                N_MUTATIONS = 2
                N_CHROMOSOMES = 10
                N_GENERATIONS = 100
                population[ ][ ]
                best_output[ ]
                best_fitness[ ]
                LB = {10^18, 0, 10^18, 10^18}
                UB = {10^19, 10^6, 10^19, 10^19}
                p1 = N_PARENTS
                p2 = N_PARENTS+N_OFFSPRINGS

for i = 0, 1, …, N_CHROMOSOMES-1 do:
  for j = 0, 1, …, N_GENES-1 do:
     population[i][j] = rand_int(LB[j], UB[j])
  end
end

for g = 0, 1, …, N_GENERATIONS-1 do:
  fitness = cal_population_fitness(population)
  best_idx = where(fitness==min(fitness))
  best_output = population[best_idx]
  best_fitness[g] = fitness[best_idx]

  parents = select_mating_pool(population,
                        … fitness, N_PARENTS)

  crossover_offspring = crossover(parents,
                        … N_OFFSPRINGS, N_GENES)

  mutation_offspring = mutation(crossover_offspring,
                        … N_MUTATIONS, LB, UB,
                        … N_PARENTS, N_GENES)

  population[0:p1] = parents
  population[p1:p2] = mutation_offspring
  for i = p2, p2+1, …, N_CHROMOSOMES-1 do:
     for j = 0, 1, …, N_GENES-1 do:
        population[i][j] = rand_int(LB[j], UB[j])
     end
  end
end

Return best_output, best_fitness
```

The configurations of the Google cloud VM instance used for the simulation of the given algorithm are: *n1-standard-2* type (2 vCPUs, 7.5 GB RAM), located in the *us-west1-b* zone, with an NVIDIA Tesla-K80 GPU, and Ubuntu 18.04LTS operating system. The program was designed using the C + + language, and was compiled with the NVCC compiler, along with OpenCV version 4.1.1, and with CUDA version 10.1. The algorithm was

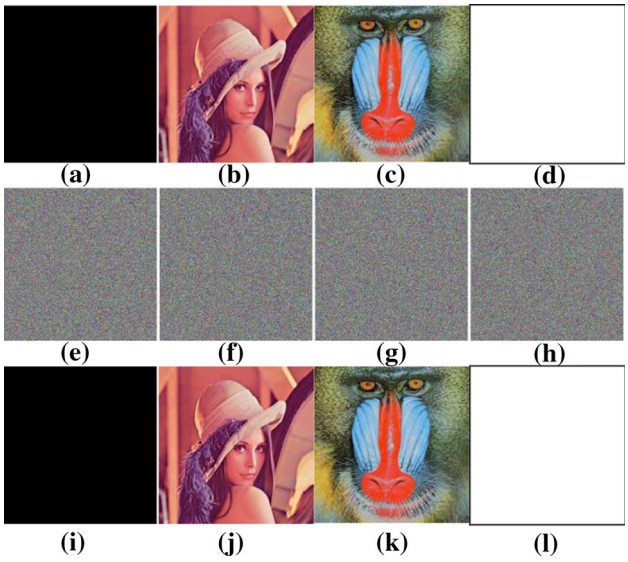

Fig. 4 Application of the proposed algorithm to the **a–d** All-black, Lena, Mandrill, and All-white test images, to obtain the **e–h** encrypted images, and their **i–l** decrypted counterparts

Table 1 Initialization vector values

All-black	15	CB	22	F8	09	C4	57	D4
Lena	15	CB	22	F8	0C	FA	FD	7C
Mandrill	15	CB	22	F8	10	41	7C	F4
All-white	15	CB	22	F8	13	73	48	6C

Table 2 Initial secret key

38	33	3B	61	F1	2C	F9	7B
59	FC	B2	17	09	29	C3	15
B9	28	8A	FD	3E	ED	1E	4C
22	51	0C	02	A6	3E	0D	D0

Table 3 Values of hyper-parameters corresponding to the best output of the genetic algorithm

Hyper-parameter	Value
A	10,000,000,000,000,000,000
B	126,501
C	6,507,463,502,728,333,312
D	13,787,056,135,999,930,368

designed to take the test images as input, and perform encryption and decryption operation 500 times on them, using the parameters specified in the previous sections. The obtained time-durations for the corresponding operations were averaged, and the obtained values were used to

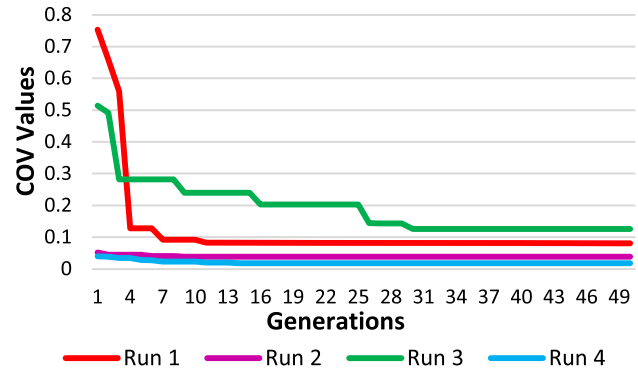

Fig. 5 Multiple runs of the genetic algorithm

compute the average speeds of encryption and decryption respectively, as presented in Table 4. Table 5 compares the average encryption and decryption speeds of the proposed algorithm, and the ones proposed in Failed (2015); Bharadwaj and Sairabanu 2020; Zhang et al. 2017) and (Seshadrinathan and Dempski 2008) respectively. It can be seen that the performance achieved using the proposed algorithm, is far better than the other schemes.

To calculate the speedup ratio between the serial (from Bharadwaj and Sairabanu (2020)) and parallel (with GPU) implementations of the proposed algorithm, the execution speeds were considered from Table 5. These values were used in (10), where $(T_{\text{serial}}, T_{\text{parallel}})$ represent the execution times, and $(V_{\text{serial}}, V_{\text{parallel}})$ represents execution speeds of the respective algorithms. The obtained values of the speedup ratio were 3.489 for encryption, and 4.055 for decryption, respectively.

$$Speedup, S = \frac{T_{\text{serial}}}{T_{\text{parallel}}} = \frac{V_{\text{parallel}}}{V_{\text{serial}}} \tag{10}$$

7 Security analysis

7.1 Key space

The presented algorithm is capable of functioning with a key of arbitrary length. It's only constrained by the system-specific parameters, such as the available memory, processing power, required level of security, etc. In spite of this, it is recommended to use at least a 256-bit long key, to obtain a respectable level of security. In general, the key-space will be 2^n for an n-bit key.

7.2 Analysis against various attacks

When the presented algorithm is subjected to a known-plaintext or chosen-plaintext attack, the attacker cannot decipher the seed values, without knowing the initial secret

Table 4 Speeds of execution for proposed algorithm

Image	Image size (pixels)	Image size on disk	Encryption Speed (Mbps)	Decryption speed (Mbps)
Black	512 × 512	4 KB	575.135	845.843
Lena	512 × 512	312 KB	329.422	396.858
Mandrill	512 × 512	314 KB	329.531	393.998
White	512 × 512	4 KB	656.123	999.816

key (assumed secure for smooth operation). Moreover, re-iterating the same plaintext would also not help, due to the uniqueness of the *IV* parameter.

For a Man-In-The-Middle (MITM) attack, as long as the key-exchange mechanism ensures that the authenticity of the parties involved in the communication is verified before transmission, and the key-space is of recommended size (or more), the algorithm is deemed secure. It is so, because of the uni-directional nature of the involved computations, which make it mathematically impossible to replicate the seed values, by back-tracking the values in the algorithm.

The algorithm does not explicitly provide a means to combat a Denial-of-Service (DoS) attack, for which suitable arrangements must be performed by the user, during the development of the system in consideration.

7.3 Histogram analysis

The pixel values of the input images, and their respective encrypted counterparts were subjected to a frequency analysis, and the obtained frequency distribution was plotted on a 2D plot to obtain the histogram of each image. The intensity values were plotted on the X-axis, and frequency on Y-axis. Upon examining these plots through Fig. 6, it was observed that even though some contextual information was visible from the histograms of the input images, there was no such feature visible in the same for the encrypted images.

7.4 Correlation analysis

The correlation analysis was performed on the input and encrypted images, by taking n random samples of pairs of pixels in the horizontal, vertical, and diagonal directions, respectively for each image. Each ordered pair of pixels is represented by (u_i, v_i). The correlation coefficient for the drawn samples can be calculated using (11). (Ye et al. 2018)(Vihari and Mishra 2012).

$$r_{uv} = \frac{\sum_{i=1}^{n}(u_i - \overline{u})(v_i - \overline{v})}{\sqrt{\sum_{i=1}^{n}(u_i - \overline{u})^2}\sqrt{\sum_{i=1}^{n}(v_i - \overline{v})^2}} \tag{11}$$

Following this approach, the correlation coefficients obtained for a sample size of $n = 5000$, corresponding to each considered image, can be found in Table 6. The Fig. 7 plots the (u_i, v_i) pairs in horizontal direction on the two axes. It can be clearly seen that considerable levels of correlations are detected for the input images, whereas the same is in a close proximity to zero, for the encrypted images. This signifies a negligible degree of correlation. Figure 8 presents the comparison of the results of proposed and the previous algorithms. As can be seen, the proposed algorithm has comparable results to Bharadwaj and Sairabanu (2020), and outperforms (Failed 2015) and (Zhang et al. 2017).

7.5 Information entropy

In order to quantify the amount of information that is contained within a given binary data, the measure called information entropy is used. In the context of 8-bit grayscale images, its value is limited in the range [0,8]. It is defined in such a manner that for an 8-bit truly random image I_r, the information entropy $H(I_r)$ has the maximum possible value, i.e. 8. The value of information entropy is computed using (12) for the test images and their encrypted counterparts, and the results are presented in Table 7. (Ye et al. 2018).

$$H = -\sum_{i=0}^{255} p(i)\log_2[p(i)] \tag{12}$$

It is quite clear from Table 7 that the entropy values obtained for the test images, are having a significant deviation from the desired value, i.e. 8. But, for the encrypted images, the entropy values are at par with the expected values. Also, when the obtained values were compared with the algorithms proposed in Failed (2015); Bharadwaj and Sairabanu 2020), and (Zhang et al. 2017) in Fig. 9, we can see that the values are in a close agreement to those of Bharadwaj and Sairabanu (2020), but are quite closer to the desired value of 8, in comparison to the same values obtained using the algorithms from Failed (2015) and (Zhang et al. 2017), indicating better entropy levels for our algorithm.

Table 5 Comparative analysis of the encryption and decryption speeds

Algorithm	Encryption speed (Mbps)	Decryption speed (Mbps)
Proposed Alg	472.553	659.129
Ref. (Ramesh and Jain 2015)	11.6099*	11.6121*
Ref. (Zhang et al. 2017)	48.7710	44.6203
Ref. (Bharadwaj and Sairabanu 2020)	135.413	162.524
Ref. (Seshadrinathan and Dempski 2008)	402 (approx.)	402 (approx.)

*Computed from the provided execution time

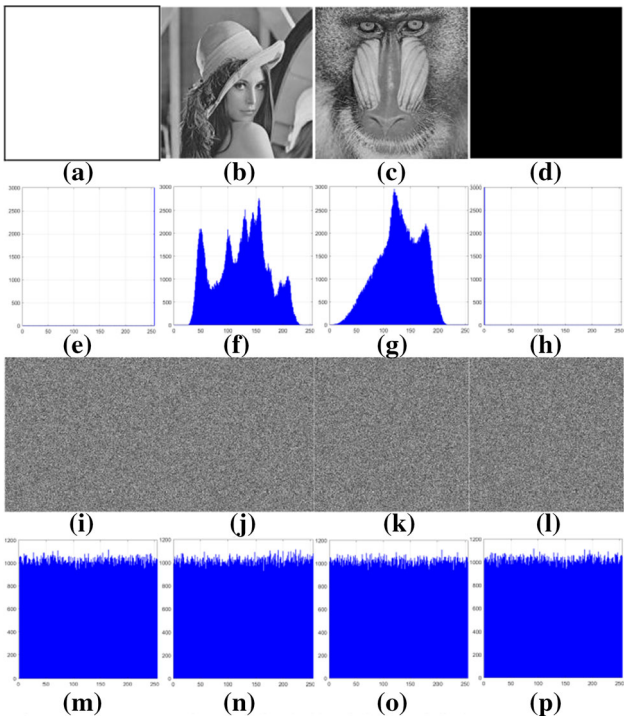

Fig. 6 Histogram analyses to the **a–h** original and **i–p** encrypted images in horizontal direction

7.6 Differential analysis/sensitivity analysis

The differential attack is a process of obtaining two or more pairs of plaintexts and their corresponding ciphertexts, and using these to deduce information, pertaining to the encryption key used to generate them, and/or to reduce the time needed to deduce the encryption key. In order to secure any given cipher from any such analytical approaches, it is essential that two key properties called confusion and diffusion, first identified by Shannon (1949), be incorporated into it. This can be done by using certain mathematical transformations, or changing the structure of operation of the cipher.

In order to mathematically quantify the confusion and diffusion capacity of a given image encryption algorithm, two parameters are used extensively in the literature, namely number of pixel change rate (NPCR) and unified average changing intensity (UACI). NPCR indicates how many pixels changed their value, when the cipher operated upon the given input image. The UACI measures the average difference between the values of pixels that

Table 6 Correlation coefficients

Image		Horizontal	Vertical	Diagonal
All-back	Figure 7a	Undefined	Undefined	Undefined
	Figure 7i	− 0.001081	− 0.001576	− 0.001240
Lena	Figure 7b	0.966180	0.984140	0.951990
	Figure 7j	− 0.000652	0.004573	− 0.001306
Mand-rill	Figure 7c	0.932240	0.909501	0.860810
	Figure 7k	0.000167	− 0.007593	− 0.001755
All-white	Figure 7d	Undefined	Undefined	Undefined
	Figure 7l	− 0.001057	− 0.001824	0.000371

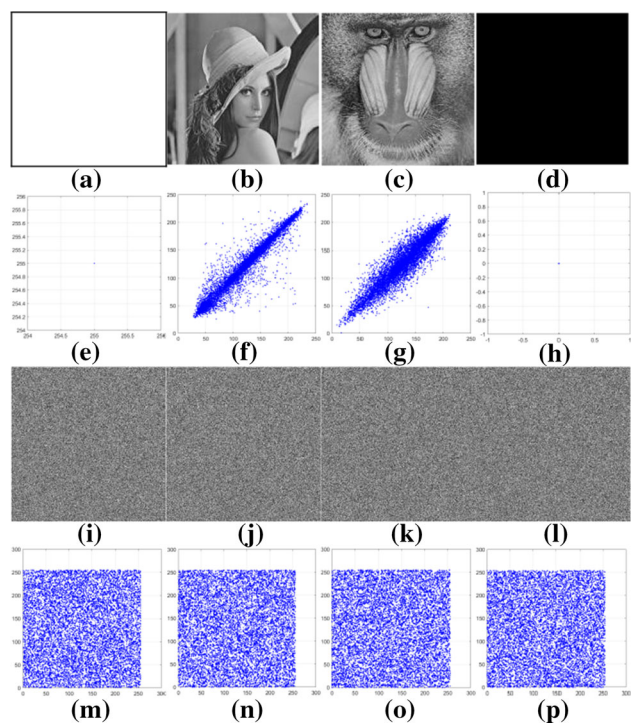

Fig. 7 Correlation analyses to the **a–h** original and **i–p** encrypted images in horizontal direction

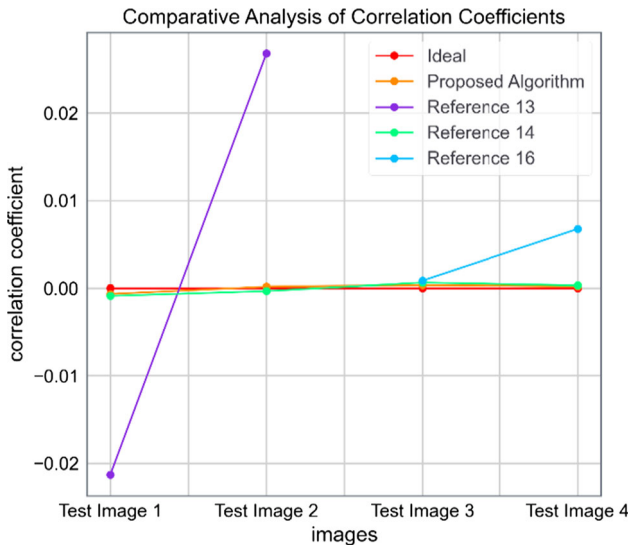

Fig. 8 Comparative analysis of correlation Coefficients

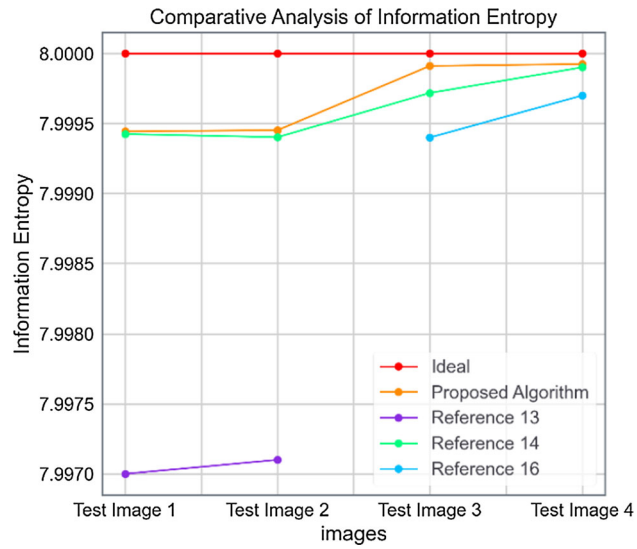

Fig. 9 Comparative analysis of Information Entropy

changed their values, with respect to their counterparts in the plaintext images.

For two images I_1 and I_2 of same dimensions M × N, their NPCR and UACI values are calculated using (13.1–13.2) and (14), respectively. Here, L represents the number of levels present in the given images' gray values.

$$NPCR = \frac{1}{M \times N} \sum_{i=0}^{M} \sum_{j=0}^{N} D(i,j) \times 100\% \tag{13.1}$$

$$D(i,j) = \begin{cases} 0, if I_1(i,j) = I_2(i,j) \\ 1, if I_1(i,j) \neq I_2(i,j) \end{cases} \tag{13.2}$$

$$UACI = \frac{1}{M \times N} \sum_{i=0}^{M} \sum_{j=0}^{N} \frac{|I_1(i,j) - I_2(i,j)|}{L-1} \times 100\% \tag{14}$$

In order to further investigate the sensitivity of the presented algorithm to changes in the input image and the encryption key, and to see the confusion and diffusion processes in action, two sets of NPCR and UACI values were computed for each image. For the first set of values, the encryption operation was performed two times: first using the inputs in their presented form, and the second time by flipping the first bit of the encryption key used for encryption of the image, to measure the key-sensitivity. Also, for the next set, the encryption operation was performed by only changing the value of the top-right pixel of the test image, to observe the plain-block sensitivity.

It should be noted that for a truly random image, the expected NPCR and UACI values are 99.6094% and 33.4653%, respectively. These values, along with the values obtained after performing the described analysis on the test images, are presented in Table 8. The NPCR and UACI values computed for the considered test images using this approach, and the ones reported in Ramesh and Jain (2015); Bharadwaj and Sairabanu 2020), and (Zhang et al. 2017) are presented in Fig. 10. It is quite clear that the proposed algorithm is performing at-par with (Bharadwaj and Sairabanu 2020), and better than (Ramesh and Jain 2015) and (Zhang et al. 2017), for NPCR values. With respect to UACI values, it can be observed that the proposed algorithm works at-par with (Ramesh and Jain 2015)

Table 7 Results of information entropy analysis

Image	All-black	Lena	Mandrill	All-white
Plain	0	7.445061	7.292549	0
Encrypted	7.993687	7.999443	7.999451	7,993,643

Table 8 Key and plain block sensitivities

Encrypted image	Key sensitivity (%)		Plain block sensitivity (%)	
	NPCR	UACI	NPCR	UACI
Theoretical	99.6094	33.4635	99.6094	33.4635
All-black	99.6033	33.4651	99.6189	33.4383
Lena	99.5983	33.3972	99.5955	33.4725
Mandrill	99.5972	33.4856	99.5976	33.4762
All-white	99.5895	33.4176	99.5640	33.4799

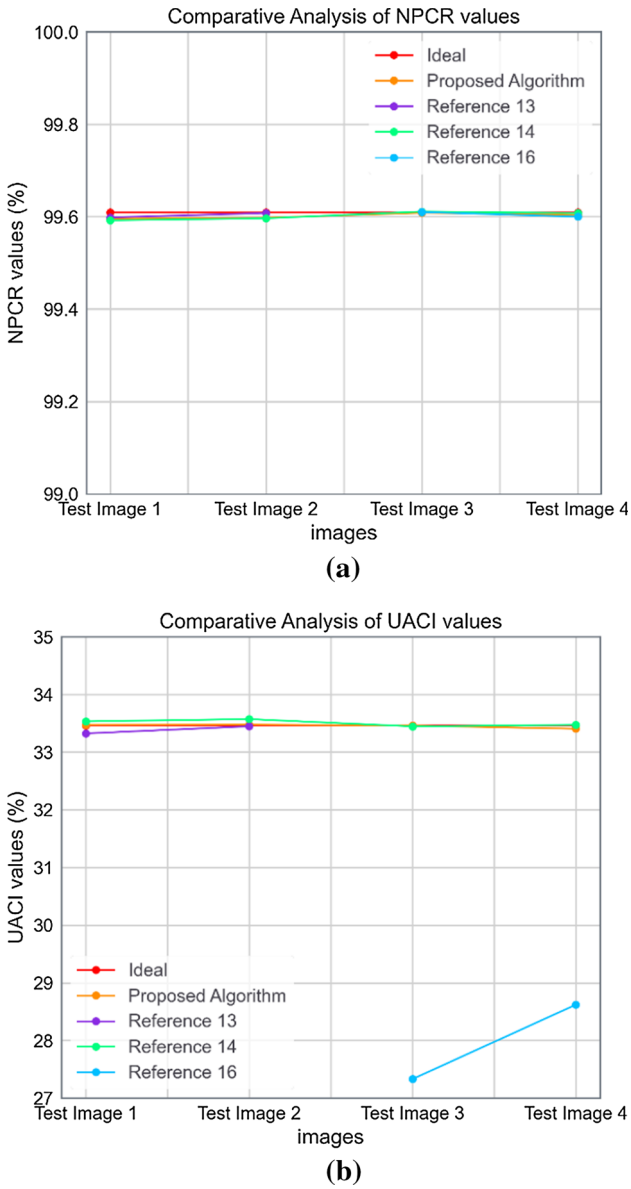

Fig. 10 Comparative analysis of **a** NPCR and **b** UACI values

and (Bharadwaj and Sairabanu 2020), and has better results than (Zhang et al. 2017).

8 Conclusion

This paper presents a symmetric-key image encryption algorithm, that accepts certain input parameters along with a shared secret key, to perform the process of encryption and decryption. The encryption and decryption processes make use of a GPU-accelerated encryption paradigm, that utilizes the concepts of an XOR cipher, along with a genetically-optimized pseudo random generator, to generate a stream of random bytes that are used to obtain the encryption pad, needed to encrypt the pixel-values. The presented algorithm was subjected to a series of mathematical analyses, to categorically prove its security against a number of possible attacks. The performance of the algorithm was compared with that of other algorithms of the same problem-domain, and similar implementations, to quantify the performance enhancements and derive the speed-up ratio.

In future, the presented algorithm can be experimentally compared against the previously used algorithms, to gather insights related to the relative benchmarking in a production environment. The structure of the helical structure involved in the corresponding computations can also be reviewed in future, to further increase the scope of parallelism, in comparison to the same of the current version. Also, if a video stream is considered as a collection of individual frames, the same algorithm can be used to encrypt a video stream with suitable modifications. Such an application will prove to have an extensive use-case relevance to video-sharing features, implemented in the virtual meeting and video-conferencing platforms, that have recently gained traction, due to the COVID-19 pandemic.

Declarations

Conflict of interest All authors certify that they have no affiliations with or involvement in any organization or entity with any financial interest or non-financial interest in the subject matter or materials discussed in this manuscript.

References

Abdullah AH, Enayatifar R, Lee M (2012) A hybrid genetic algorithm and chaotic function model for image encryption. AEU-Int J Electron Commun 66(10):806–816

Affenzeller M, Wagner S, Winkler S, Beham A (2009) Genetic algorithms and genetic programming: modern concepts and practical applications. CRC Press, Florida

Almalkawi IT, Halloush R, Alsarhan A, Al-Dubai A, Al-karaki JN (2019) A lightweight and efficient digital image encryption using hybrid chaotic systems for wireless network applications. J Inf Secur Appl 49:102384

Altigani A, Hasan S, Shamsuddin SM, Barry B (2019) A multi-shape hybrid symmetric encryption algorithm to thwart attacks based on the knowledge of the used cryptographic suite. J Inf Secur Appl 46:210–221

Antwerpen DV (2011) Improving SIMD efficiency for parallel Monte Carlo light transport on the GPU. In Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics (HPG '11). Association for Computing Machinery (ACM), New York, NY, USA, pp 41–50.

Armin J, Thompson B, Kijewski P (2016) Cybercrime economic costs: No measure no solution. Combatting cybercrime and cyberterrorism. Springer, Cham, pp 135–155

Bharadwaj B, and Sairabanu J (2020) Image encryption using a Modified Pseudo-Random Generator. In 2020 International

Conference on Emerging Trends in Information Technology and Engineering (ic-ETITE) pp 1–6. IEEE.
Bharadwaj B, Shukla S, Shalini L (2018) Symmetric key encryption using a simple pseudo-random generator to provide more secure communication. Int J Tech Res Appl 6(2):76–81
Boritz JE, No WG (2005) Security in XML-based financial reporting services on the Internet. J Account Public Policy 24(1):11–35
Buck I (2007) GPU computing with NVIDIA CUDA. In ACM SIGGRAPH 2007 courses (SIGGRAPH '07). Association for Computing Machinery, New York, NY, USA, 6–es.
D'agapeyeff A (2016) Codes and ciphers-A history of cryptography. Read Books Ltd., Redditch
Davies D (1997) A brief history of cryptography. Inf Secur Tech Rep 2(2):14–17
Dutta S, Das T, Jash S, Patra D, and Paul P (2014) A cryptography algorithm using the operations of genetic algorithm & pseudo random sequence generating functions. International Journal, 3(5).
Inampudi GR, Shyamala K, and Ramachandram S (2018) Parallel implementation of cryptographic algorithm: AES using OpenCL on GPUs. 2nd International Conference on Inventive Systems and Control (ICISC), Coimbatore, pp 984–988.
Kalra S, Sood SK (2015) Secure authentication scheme for IoT and cloud servers. Pervasive Mob Comput 24:210–223
Kösemen C, Dalkiliç G, Aydin Ö (2018) Genetic programming-based pseudorandom number generator for wireless identification and sensing platform. Turk J Electr Eng Comput Sci 26(5):2500–2511
Lee WK, Cheong HS, Phan RCW, Goi BM (2016) Fast implementation of block ciphers and PRNGs in Maxwell GPU architecture. Clust Comput 19(1):335–347
Manssen M, Weigel M, Hartmann AK (2012) Random number generators for massively parallel simulations on GPU. Eur Phys J Special Topics 210(1):53–71
Mishra M, and Mankar VH (2015) Text encryption algorithms based on pseudo random number generator. International Journal of Computer Applications, 111(2).
NVIDIA Corporation (2020) CUDA C++ Programming Guide. NVIDIA Docs. https://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf
Ramesh A, and Jain A (2015) Hybrid image encryption using Pseudo random number generators, and transposition and substitution techniques. 2015 International Conference on Trends in Automation, Communications and Computing Technology (I-TACT-15), Bangalore, pp. 1–6
Riek M, Böhme R (2018) The costs of consumer-facing cybercrime: An empirical exploration of measurement issues and estimates. J Cybersecur 4(1):tyy004
Riesinger C, Neckel T, Rupp F (2018) Non-standard pseudo random number generators revisited for GPUs. Futur Gener Comput Syst 82:482–492
Sen A, Ghosh A, and Nath A (2017) Bit level symmetric key cryptography using genetic algorithm. In 2017 7th International Conference on Communication Systems and Network Technologies (CSNT) (pp. 193–199). IEEE.
Seshadrinathan M, and Dempski KL (2008) Implementation of advanced encryption standard for encryption and decryption of images and text on a GPU. IEEE Computer Society Conference on Computer Vision and Pattern Recognition Workshops, Anchorage, AK, pp 1–6.
Shannon CE (1949) Communication theory of secrecy systems. Bell Syst Tech J 28(4):656–715
Stergiou C, Psannis KE, Kim BG, Gupta B (2018) Secure integration of IoT and cloud computing. Futur Gener Comput Syst 78:964–975
Vihari PLV, and Mishra M (2012) Chaotic image encryption on GPU. In Proceedings of the CUBE International Information Technology Conference (CUBE '12). Association for Computing Machinery, New York, NY, USA, 753–758.
Ye R, Li Y, and Li Y (2018) An image encryption scheme based on fractal interpolation. In Proceedings of the 3rd International Conference on Multimedia and Image Processing (ICMIP 2018). Association for Computing Machinery, New York, NY, USA, 52–56.
Zafar F, Olano M, and Curtis A (2010) GPU random numbers via the tiny encryption algorithm. In Proceedings of the Conference on High Performance Graphics (HPG '10). Eurographics Association, Goslar, DEU, pp 133–141.
Zeller C (2011) Cuda c/c++ basics. NVIDIA Coporation, California
Zhang Y, Li X, and Hou W (2017) A fast image encryption scheme based on AES, 2nd International Conference on Image, Vision and Computing (ICIVC), Chengdu, pp 624–628.