Journal of Computer Science and Technology Studies (JCSTS))
ISSN: 2709-104X
DOI: 10.32996/jcsts
Journal Homepage: www.al-kindipublisher.com/index.php/jcsts

Safer and More Efficient Parallel Cryptographic Algorithm and its Implementation in the GPU

Artan Berisha
Department of Mathematics, Faculty of Mathematics and Natural Sciences, University of Prishtina, Prishtina, Kosova
Corresponding Author: Artan Berisha, **E-mail**: artan.berisha@uni-pr.edu

ARTICLE INFORMATION

Received: April 08, 2021
Accepted: June 01, 2021
Volume: 3
Issue: 1
DOI: 10.32996/jcsts.2021.3.1.2

KEYWORDS

GPU, Parallel algorithms, Hill cipher technique, Non-invertible matrix, Hash algorithm, One-way hash function

ABSTRACT

In the digital world, the demand for data security during communication has increased. Hash functions are one of the cryptographic algorithms that provide data security in terms of data authenticity and integrity. Nowadays, most online applications require user authentication. These authentications are done on the server-side, which he must manage. As the number of applications increases, building a one-way function will be faster for calculating a hash value for small data such as passwords. In this paper, we will present a sequential cryptographic algorithm and its parallel implementation. We performed security analyses, executed comparisons for different amounts of data, and provided steps for further developing this algorithm. With the construction of this one-way function, we have provided the calculation of hash value in a shorter time for data in small quantities, which speeds up the authentication process on the server and thus speeds up the online services provided by the respective applications. A comparison was made between sequential implementation, parallel implementation on the CPU, and parallel implementation on the GPU using CUDA (Computer Unified Device Architecture) platform.

1. Introduction

With the fast development of the Internet, many client/server-based services architecture such as online shopping, online payments etc., has become the main services. So the authentication identity of remote users has become very important. In order to eliminate the integrity and authentication issues, some authors proposed protocols using one-way hash functions (Fan et al., 2005) (Hwang et al., 2010) (Song, 2010) (Berisha et al., 2012). Smart cards are used as multi-server authentication protocol with hash algorithms; thus, the large number of authentications can slow internet traffic and slow down services.

In recent years, GPU (Graphical Processing Unit) has become more frequent in the acceleration of calculations for different issues such as Computer Science, Mathematics, Biology, Chemistry, etc. Lately, It is being used to accelerate cryptographic algorithms for increased effectiveness and efficiency. For this, the company NVIDIA has released the platform CUDA (Computer Unified Device Architecture) programming On GPU. This has made easier parallel programming without caring much about mapping variables. This paper will be implemented in GPU parallel algorithm) (Berisha et al., 2012). The second part of the paper will contain a description of the algorithm) (Berisha et al., 2012) in the form sequentially. The third part will contain a study of GPU programming and architecture. The fourth will be the parallel implementation of the algorithm (Berisha et al., 2012) and proof of some theorems related to this algorithm as a hash function. In the end, results will be shown and conclusions drawn and the work in the future.

2. Description of one way hash algorithm

2.1 Hash functions

It is hard to design a function that accepts a variable input and give fixed output with non reversible property. These functions are called hash functions and in real world are built on the idea of a compression Function) (Berisha et al., 2012). In general, the hash function is H: {0,1}^* → {0,1}^n for some n. In order for H to be a hash Function it is needed some basic properties. It can be applied to any block size of data, produce a fixed-length output and (Stallings, 2005).

- **Preimage resistant, for given $\mathbf{y} \in \{0,1\}^n$ it is "hard" to find $\mathbf{x} \in \{0,1\}^n$ such that $\mathbf{H(x) = y}$.**
- **Second preimage resistant, for given $\mathbf{x} \in \{0,1\}^*$ it is "hard" to find $\mathbf{x'} \in \{0,1\}^*$, $\mathbf{x \neq x'}$ such that $\mathbf{H(x) = H(x')}$.**
- **Collision resistant, it is "hard" to find $\mathbf{x}$, $\mathbf{x'}$ $\{0,1\}^*$, such $\mathbf{x \neq x'}$ and $\mathbf{H(x) = H(x')}$.**

A one-way hash function is a very important cryptographic primitive. It is used for data integrity and Authentication. The output length of the value of a hash function is fixed and the input is variable length. Most usually cryptographic hash functions used today are SHA-2, SHA-3 where MD5 is broken (Schneir, 1996) and it was used to break SSL. Most of the hash functions have to give an output greater than 160 bit. This is because of the birthday attack, which says that to find a collision for a cryptographic hash function with n bit output with probability 50% we expect *n*=2 input values (Paar, 2011). Nowadays, with the computing resources capability, all algorithms that have more than 2^{80} input cases for brute force attack are considered secure. Because of this, the output of a hash the function must be greater than 160 bit to be secure (it means $2^{\frac{160}{2}} = 2^{80}$). Also, NIST, in its Secure Hash Standard uses a 160-bit hash value. This makes it even harder for the birthday attack. It requires 2^{80} random text to Find two hash codes with the same value (Schneir, 1996).

2.2 Sequential algorithm

In (Taha et al., 2011), authors proposed a model based on matrix multiplication. It is called Hill cipher, and this model is based on the non-invertible matrix for a practical one-way hash function. This non-invertible matrix multiplied plaintext to generate the hash value. Proposed a solution are given in (Berisha et al., 2012). which automates the model for a one-way hash function given by (Taha et al., 2011). The non-invertible matrix for multiplying plaintext will be generated by given size m of the square matrix. The algorithm will generate a non-invertible matrix as a sum of two permutation matrices. The elements of the generated matrix will be from GF(2), which means that their value is 0,1. In (Berisha et al., 2012). there are proposed two designs of algorithms for generating hash value. Both models are based on Cipher Block Chaining (CBC), the second model differs from the first in some additional operations. These additional operation steps are to create diffusion using non-linear function F. Before calculating the hash value the plaintext must be converted to binary data, and divided to column vectors $\{\boldsymbol{B_1}, \boldsymbol{B_2}, \dots, \boldsymbol{B_N}\}$ of size *(mx1)*, $B_i = (b_{i0}, b_{i1}, \dots, b_{i,m-1})$, $b_{ij} \in \{0,1\}$.

The padding for both proposed models is done different but in one stage, they are common. First, we see if the size of the last column vector is less than *m*. If yes, it will be padded with values 1 and summed with initial vector $\boldsymbol{H_0}$ modulo 2. The plaintext now contains N column vectors with size (*mx*1). For the first model this padding will do the work. After that, the process of calculating hash value will begin with the early generated non-invertible matrix. For the second model the number of column vector must be a multiply of number 2, so if $N \bmod 2 \neq 0$ then the last column will be summed with the initial vector $\boldsymbol{H_0}$ modulo 2, and *N* = *N* - 1 is the number of vector columns.

Sequential hash algorithm [1]

```
INPUT: Non-invertible matrix P of size $m$ and value $M$.
OUTPUT:   $m$ bit hash value.
STEP 1    Convert value M to binary form $b_0, b_1, \dots, b_k$
STEP 2    Padding value algorithm
STEP 3    Initialize $S = 0, H = H^0$,
          $H^0 = h_0^0 h_1^0 \dots h_{m-1}^0$ , $h_i^0 = 0$ for i=0,1,...,m-1.
STEP 4    for j from 0 to n-1
              STEP 5   for i from 0 to m-1
              $M'_{ij} = b_{i+mj}$
STEP 6    for i from 0 to n/m-1
              STEP 6'   if (i mod 2 == 0 && i<>0)
                              $H = F(H, H^i)$
                    else for r from 0 to m-1
                              $h_r^i = h_r$
              STEP 7    for j from 0 to m-1
                 $B_j = M'_{ij} + h_j$
                         STEP 8 for k from 0 to m-1
                                STEP 9  for t from 0 to m-1
                                    $S = (P_{kt} * B_t + S)(\bmod 2)$
                                    $h_k = S$
                                    $S = 0$
```

OUTPUT $h_0, h_1, \dots, h_{m-1}$.

3. Compute unified device architecture

3.1 GPU programming

It is a platform for programming and parallel computation developed by NVIDIA and implemented in graphic processing units (GPU). Driven by the market demand for a platform for general use in programming and computing in real time, with higher graphics GPU has managed to present some hardware and software techniques as the architecture of unified shader (UNIFIED shader architecture) processors range (streaming processors) (Schneir, 1996) (Paar, 2011).

The traditional GPU includes processors range, which can be used for operations in pixels and vertices (Thomson et al., 2002). First of all these processors are organized in groups called Streaming Multiprocessor (SM). Eg NVIDIA G80 has 128 series processors, while one SM has 8 stream processors. This means that the G80 GPU contains 16 SM. Role of SM in GPU is the creation, management and execution of threads in hardware. Implements a synchronization barrier for data to be read and those that will be calculated. To manage the number of multiple threads architecture then SM applies Single Instruction Multiple Threads (SIMT). With this SIMT an SM manages, and executes ranks threads in a group of 32 threads called warp. This should take into account when the number of threads is not a multiple of 32, so these exceptions should be avoided (Arul et al., 2009).

As a device for computation, the GPU is built for parallel programming and computation compared with CPU that is sequential computation. From this discrepancy in the method of calculation of CPU and GPU it is because the GPU is built for numerous calculations, with pronounced parallelism (reflection in the computer graphics). For that, the GPU are devices with the possibility of processing a large set of data (Comba et al., 2003) (Luebke, 2008). For parallel computation, the user can specify the number of threads to be executed on the GPU. This could make by stating the number of threads to be executed in a SM specifying the size of the block. Also, the user will determine the number of blocks within the network (Garland et al., 2008). All threads within a network create kernel which upon completion of calculations can send the result to the CPU (Figure 1).

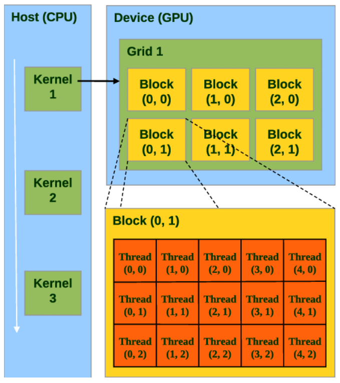

Figure 1. Network, blocks and threads

GPU is designed for operation in a large batch of data (Figure 2), so the problem should be reflected in the the architecture of the GPU. In order to improve the calculation and reduce difficulties for programming the GPU, NVIDIA company has introduced a platform called CUDA parallel programming (NVIDIA, 2010). This platform is based on the language C/C ++, recently more heading toward C ++, this has enabled all those who have knowledge on the language, C/C ++ have facilities in different implementations in parallel programming.

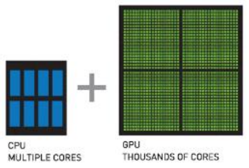

Figure 2. Inside structure of CPU and GPU

Below is the code for calculating the sum of two vectors as well as the multiplication of two matrices in CUDA.
Code for the sum of two vectors:

```
// Core to calculate the sum of two vectors
__global__ void vecAdd(float * in1, float * in2, float * out, int len) {
            int i = blockDim.x * blockIdx.x + threadIdx.x;
            if (i < len)
            out[i]=in1[i]+in2[i];
}

int threads = 256;
int number_of_threads = (length + threads -1)/threads;
//Execution of kernel
vecAdd<<<number_of_threads, threads>>>(Input1, Input2, Output, length);
```

Code for matrix multiplication:

```
//Kernel for matrix multiplication
__global__ void matrixMultiply(float * A, float * B, float * C,
          int nrAR, int nrAC, int nrBR, int nrBC, int nrCR, int nrCC) {
            int Row = blockIdx.y*blockDim.y+threadIdx.y;
            int Col = blockIdx.x*blockDim.x+threadIdx.x;
                        if ((Row < nrAR) && (Col < nrBC)) {
                        float Cvalue = 0.0;
                        for (int i = 0; i < numAColumns; ++i)
                        Cvalue += A[Row*nrAC+i] * B[Col+i*nrBC];
                        C[Row*nrCC+Col] = Cvalue;
                        }
            }
dim3 dimGrid((nrCC-1)/BLOCK_SIZE+1, (nrCR-1)/BLOCK_SIZE+1, 1);
dim3 dimBlock(BLOCK_SIZE, BLOCK_SIZE, 1);
//Execution of kernel
matrixMultiply<<<dimGrid,dimBlock>>>(d_A,d_B, d_C,nrAR, nrAC, nrBR, nrBC, nrCR, nrCC);
```

4. Parallel implementation of algorithm

The sequential algorithm implemented in (Berisha et al., 2012) is inadequate to build a parallel version of it, this is because of the CBC (Cipher Block Chaining) mode used to calculate the hash value. This mode uses the output from the previous step as input parameter and then delivers the resulting value which is used for the next step. So each step depends on the previous calculation which disables the parallel implementation of the algorithm. To realize a parallel version of the algorithm, we should use CTR (Counter) mode instead of CBC mode, method which can be easily implemented (Berisha et al., 2012). This method returns the encryption block in stream cipher, here is generated a string of bits, which is guaranteed not to be repeated for a long time (although the generation of these numbers is done in ascending order, increasing by one). Below is the implementation of an encryption block, which then will implemented in CUDA kernel with different block size and threads.

4.1 Parallel algorithm

```
INPUT:    Non invertible matrix P of size $m$, CTRi and value $M$.
OUTPUT:   $m$ bit hash value.
STEP 1    Convert value M to binary form $b_0, b_1, \dots, b_{m-1}$
STEP 2    Initialize $S = 0$
STEP 3    For fixed value of i (i- block)
              for j from 0 to m-1
                  $M'_{ij} = b_{i+mj}$
    STEP 4  for j from 0 to m-1
              $B_j = M'_{ij} + CTRj$
    STEP 5 for k from 0 to m-1
              STEP 6  for t from 0 to m-1
                  $S = (P_{kt} * B_t + S)(\mathrm{mod}\ 2)$
                  $h_k = S$
```

$$S = 0$$

OUTPUT $h_0, h_1, \dots, h_{m-1}$.

5. Results

Modified hash algorithm (Berisha et al., 2012) is implemented in CUDA platform and executed in GPU. We used a test bed with graphic card GeForce GT 610 with 48 cores and 1 GB RAM. The effect of changing file size for calculating hash value was chosen. Below is a table (Table. 1) with our gained results from (Berisha, 2015) with added column for execution time for parallel version implemented on GPU. For parallel version implemented in Java we used 4 core processor with 3.4 GHz except for the four last cases we used 16 core processor with speed 2.8 up to 4.2 GHZ.

Table 1. Time (ms) for calculating hash value with proposed parallel model and sequential model for different file size

File size (bit)	Calculating hash value (ms) sequential model (A)	Calculating hash value (ms) parallel model (B) (Java)	Calculating hash value (ms) parallel model (C) (GPU)	Speedup =A/B	Speedup = A/C
256	11	0	0	N/A	N/A
512	11	0	0	N/A	N/A
1024	11	0	0	N/A	N/A
2048	11	0	0	N/A	N/A
4096	11	0	0	N/A	N/A
8192	11	0	0	N/A	N/A
16384	17	31	25	0.54	0.68
32768	30	31	25	0.96	0.83
65536	62	31	20	2.00	3.1
131072	124	47	27	2.63	4.59
262144	237	78	41	3.03	5.78
524288	501	125	73	4.00	6.86
1048576	2855	191	125	14.94	22.84
2097152	5266	406	174	12.97	30.26
4194304	10811	764	327	14.15	33.06

We can see the proposed parallel algorithm is faster for the file size greater than 64 KB, and for a large amount of data it is 30 times faster than sequential model and two times faster than implemented parallel algorithm (Luebke, 2008) in Java.

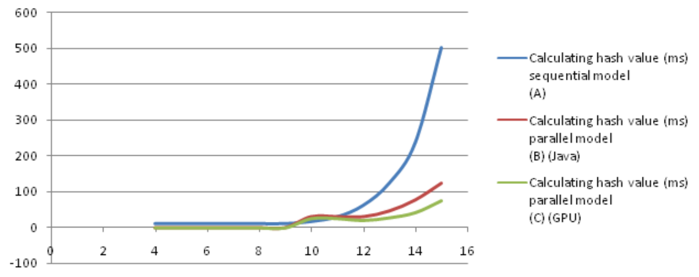

Figure 3. Graphical view of performance of proposed parallel algorithms in Java and GPU implementation against sequential algorithm.

6. Conclusion and Future Work

In this paper we proposed a parallel model of the earlier proposed sequential model. For a small amount of data (1 KB – 64 KB) this proposed parallel model is not faster than a sequential model, but for a larger amount of data (greater than 64 KB), it is faster (Figure 3). For a larger amount of data, the speedup increases exponentially. In future, we need to see for non-invertible matrices in GF(*pm*) and to speed up the calculating process by using Parallel implementation of matrix multiplication.

References

[1] Fan C.I., Chan Y. C., Zhang Z.K. (2005). Robust remote authentication scheme with smart cards. Computers & Security; 24 (8), 619–28. https://doi.org/10.1016/j.cose.2005.03.006

[2] Hwang M.S., Chong S.K., Chen T.,Y. (2010). Dos-resistant ID-based password authentication scheme using smart cards. *Journal of Systems and Software ;83*(1):163–72. https://doi.org/10.1016/j.jss.2009.07.050

[3] Song R.G. (2010). Advanced smart card based password authentication protocol. *Computer Standards & Interfaces ;32*(5-6):321–5. https://doi.org/10.1016/j.csi.2010.03.008

[4] Berisha, A., Baxhaku, B., Alidema, A. (2012). A Class of Non Invertible Matrices in *GF*(2) forPractical One Way Hash Algorithm. New York : International Journal of Computer Applications,, Vol. 54, 10.5120/8667-2574

[5] Stalings, W. (2005) *Cryptography and Network Security, PrIncipIes and Practice*. s.l. : Prentice Hall,

[6] Schneir, B. (1996). *Appiled Cryptography*. s.l. : Wiley Computer Publishing.

[7] Paar, C., Pelzl, P., (2011). *Understanding Cryptography*. s.l. : Springer.

[8] Taha, A.M., Farajallah, M., Tahboub, R. (2011) A Practical One Way Hash Algorithm based on Matrix Multiplication. . s.l. : *International Journal of Computer Applications*, *23*(2),0975-8887. 10.5120/2859-3677

[9] Huang, Q., Huang, Z., Werstein, P. and Purvis, M. (2008). GPU as a General Purpose Computing Resource. In Proceedings of the 2008 Ninth international Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT 2008). Dunedin, New Zealand, December,151-158. 10.1109/PDCAT.2008.38

[10] R. Suda, R., Aoki, T., Hirasawa, S., Nukada, A.,Honda, H. and Matsuoka, S. (2009) Aspects of GPU for General Purpose High Performance Computing. In Proceedings of the 2009 Asia and South Pacific Design Automation Conference (ASP-DAC 2009). Yokohama, Japan, January 2009, pp.216-223. 10.1109/ASPDAC.2009.4796483

[11] Thompson, C. J., Hahn, S., and Oskin, M. (2002). Using Modern Graphics Architectures for General-Purpose Computin: a Framework and Analysis. In Proceedings of the 35th Annual ACM/IEEE international Symposium on Microarchitecture (MICRO-35). Istanbul, Turkey. November 306-317. 10.1145/774861.774894

[12] Arul, S., Dash, M., Tue, M., and Wilson, N. (2009).Hierarchical Agglomerative Clustering Using Graphics Processor with Compute Unified Device Architecture, In Proceedings of International Conference on Computer Design and Applications (ICCDA 2009), Singapore, May 2009, pp. 556-561. 10.1109/ICSPS.2009.167

[13] Comba, J. L. D., Dietrich, C. A., Pagot, C. A. and Scheidegger, C. E.. (2003) Computation on GPUs: From a Programmable Pipeline to an Efficient Stream Processor. *Revista de Informatica Te ´ orica e Aplicada*, 1, 41-70.

[14] Luebke, D. (2008). CUDA: Scalable Parallel Programming for High-Performance Scientific Computing. In Proceedings of 5th IEEE International Symposium on Biomedical Imaging: From Nano to Macro (ISBI 2008), Paris France, 836-838. 10.1109/ISBI.2008.4541126

[15] Garland, M., Grand, L. S. and Nickolls, J. et al.l. (2008). Parallel Computing Experiences with CUDA. *IEEE Micro, 28*(4), 13-27. 10.1109/MM.2008.57

[16] NVIDIA. CUDA [EB/OL]. (2010-01-09). http://www.nvidia.cn/object/cuda home cn.html

[17] Berisha, A. (2015). Parallel Implementation of Proposed One Way Hash Function, , Scardus Conference, Tetove. udc: 004.421.032.24:003.26