Accelerating Encryption Algorithms Using Parallelism

Cristina-Loredana Duță, Gicu Michiu, Silviu Stoica
Department of Computer Science and Engineering
University Politehnica of Bucharest
Bucharest, Romania
email :{cristina.duta, gicu.michiu, silviu.stoica}@cti.pub.ro

***Abstract*—The importance of protecting the information has increased rapidly during the last decades and as a consequence so did the need for cryptographic algorithms. So we want to make these methods that protect our data as fast as we can and also as secure as we can. In this project, we use parallelism for encryption algorithms to bring out the full potential of it, by implementing two cryptographic modes such as CBC and ICBC for AES. The aim of this project is to show the remarkable reduction in encryption and decryption time of cryptographic systems when using parallel paradigms (OpenCL, Cuda, OpenMP and MPI) and also to evaluate and to compare the performances of serial versus parallel implementation.**

***Index terms:* symmetric cryptography, parallelization, brute force, OpenCL, Cuda, OpenMP**

I. INTRODUCTION

When they first appeared, the main purpose of Graphic Processing Units (GPUs) was to handle the generation and modification of graphical data. As soon as GPUs became increasingly powerful (in terms of floating point operations per second) compared to available CPUs in the last few years, the interest in using these devices for tasks other than the interactive generation of graphical output appeared.

The use of general purpose graphics hardware to accelerate cryptographic solutions has a long history. The first paper about cryptography on graphic hardware was published in August 1999 by Gershon Kedem and Yuriko Ishihara. They succeed to crack a UNIX password cipher using a graphic engine, PixelFlow that ran at 100 MHz

Programming GPUs are supported by new programming models based on the C language, the most known and widely used of them being vendor specific CUDA and the industry-wide OpenCL standard.

Modern GPUs can be attractive for parallel processing because these architectures by design have hundreds of processing cores and have high on-chip bandwidth close to one order in magnitude larger than modern CPUs. The advantages brought by GPUs are: good support for hiding latency in memory transactions (through massive multithreading with low context switch overhead) and the fact that the processing of instructions in the thread contexts is based on the Single Instruction Multiple Data (SIMD) processing paradigm which therefore makes them suitable for algorithms that can expose a high degree of data parallelism.

In this paper we perform an evaluation of Advanced Encryption Standard (AES).

AES is a symmetric cryptographic algorithm published by NIST. In this paper we have written in OpenSSL the code for an improved mode of operation of AES (Interleaved Cipher Block Chaining). The next step was implementing it using OpenCL, OpenMP and CUDA and to compare the results obtained in terms of performance.

Several recent research papers describe the acceleration of symmetric block ciphers as well as other cryptographic algorithms using General-Purpose Graphics Processing Unit (GPGPU) frameworks. However, referring to symmetric block ciphers, prior work focused on AES (CBC mode and ICBC mode) and the CUDA , OpenCL and OpenMP implementations, and only a few of these implementations were released as an Open Source software.

In this paper, we develop a new parallelization technique to speed up the AES-ICBC algorithm. The rest of the paper is organized as follows: Section 2 presents the related work. Section 3 gives an overview of AES algorithm and its modes of operations and also shows the architecture of our proposed solution on general-purpose multi-core processor and multi-core (OpenMP, OpenCL, and CUDA). Section 4 presents some details regarding the implementations realized. Section 5 explains our parallelization techniques compared to serial implementations of the algorithms and other implementation details. Section 6 shows the experimental results of the new approaches compared to the serial ones. Section 7 describes our conclusions and future work.

II. RELATED WORK

In the past few years, they have been some parallelization of block algorithms. One of them described the parallelization of the AES algorithm [5]. The AES algorithm was divided into parallelizable and unparallelized parts. They have shown that the iterative loops included in the most time consuming functions (responsible for the data blocks encryption and decryption) are fully parallelizable. In order to parallelize these loops they made some transformations of the body loops and used the variable privatization technique.

After some research they realized that the total running time of the AES algorithm consisted of the following time-consuming operations: 1) data reading from an input file; 2) data encryption; 3) data decryption; 4) data writing to an output file (both encrypted and decrypted text).

Their implementation of parallelized AES algorithm speed-up depends considerably in two major factors: the

capability of parallelizing the most time-consuming loops and the methods used for reading and writing data in and from the file.

The parallelized codes applied to most time-consuming loops compared to the original codes showed that they are some visible benefits in speedup of the algorithm.

The platforms on which the experiments was done was one SGI computer with one, two, four, eight and sixteen threads and this showed that the parallelized AES algorithm gains considerably time at execution of encryption or decryption. The gain earned in the most time-consuming loops was good enough in their opinion but every few years better performance is desired. Because of the sequential nature of reading data at those dates, the unparallelized code although time consuming is that which deals with file reading and writing. Because of this impediment the total gain in speed is not equal to the total gain succeeded with the parallel code. The parallel AES algorithm presented by them in this paper can be also helpful for hardware implementations. The hardware synthesis of the AES algorithm would depend on the appropriate adjustment of the data transmission capacity and the computational power of hardware.

Another approach in AES parallelization was presented in [6], where the authors described in their paper that multithreaded ciphers using ICBC are a good match for SMPs because there is no data sharing and no communication among the threads. The threads are naturally load balanced and computationally intensive, spending hundreds of cycles per cache line of input data brought from memory.

Their multithreaded implementation achieved encryption rates of 92 Mbytes/s on a 16-processor SMP at 1 GHz, reaching a factor of almost 10 improvements over a uniprocessor, which achieves 9 Mbytes/s.

They managed to hide the line cache fetch in advance completely by having the number of computation cycles per cache line very large. Also this large number of computing cycles allowed the bus to service requests of cache fetch in advance from up to 16 processors. Even so serial initialization code, software barrier costs, and bus occupancy prevent multithreaded ciphers from achieving perfect speedups on SMPs.

The cryptography community has proposed Interleaved Cipher Block Chaining (ICBC) mode for maintaining the balance between safety of data and speed of encryption. Thus, interleaved chaining loosens the recurrence imposed by CBC, enabling the multiple encryption streams to be overlapped. The number of interleaved chains can be chosen to balance performance and adequate chaining to get good data diffusion.

The parallelized ICBC algorithm was tried also with hardware capabilities [7] .This paper was used just as a starting idea because its purpose is more related to some hardware capabilities of some processors and coprocessors.

Related to CUDA parallelization [8], microprocessors with multiple cores and Graphical Processing Units (GPUs) are widely available at affordable prices. Considering the computational demands of the cryptographic algorithms, these parallel platforms are relevant to parallelize the existing algorithms to enhance the performance. CUDA programming language is used to parallelize the algorithms in GPU. Traditional cryptographic algorithms are sequential. But it is proved that with advances in hardware computational technologies such as RISC Processors, ASIC and FPGA chips, GPU and Multi-core processors and software technologies such as Decomposition and Loop Parallelization, it is possible to enhance speedup and achieve better security through parallelization.

In this approach, parallelization was done with popular cryptographic algorithms such as AES, 3DES, DES, IDEA, etc. employing the hardware or software technologies specified above. In some cases the performance is analyzed based on throughput, while the others relied on speedup. After analyzing all the techniques considered for parallelization, it has been identified that hardware techniques are efficient over that of software techniques. However, software techniques claim wide acceptance because they can be implemented on wide variety of computing systems without the need for specialized hardware units.

They have been also some OpenCL approaches like [9], presented at International Conference on Computational Science and Its Applications named The AES Implantation Based on OpenCL for Multi/many Core Architecture and its final goal was to develop a full OpenSSL library implementation on heterogeneous computing devices such as multi-core CPUs and GPUs. In this article, they presented a study on an implementation, named clAES, of the symmetric key cryptography algorithm AES using the OpenCL emerging standard. They showed a comparison of the results obtained benchmarking clAES on various multi/many core architectures. They also introduced some basic concepts of AES and OpenCL in order to describe the details of clAES implementation.

III. ARHITECTURE

The basic algorithm from which we started this project is an implementation of AES encryption algorithm using CBC-mode (represented in Figure 1).

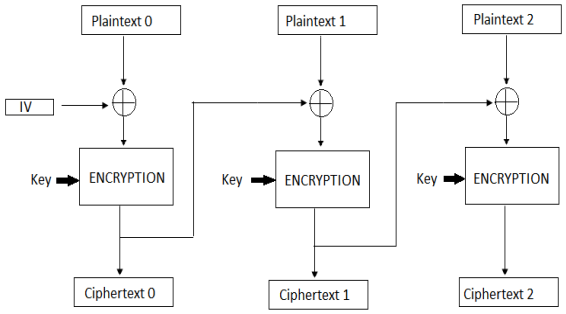

Figure 1. Cipher Block Chaining mode encryption

From this serial code we will develop an implementation of Interleaved CBC mode (represented in Figure 2) in

different environments to obtain a maximum rate of encryption for large amount of data. We will take into account parallelizing input/output data.

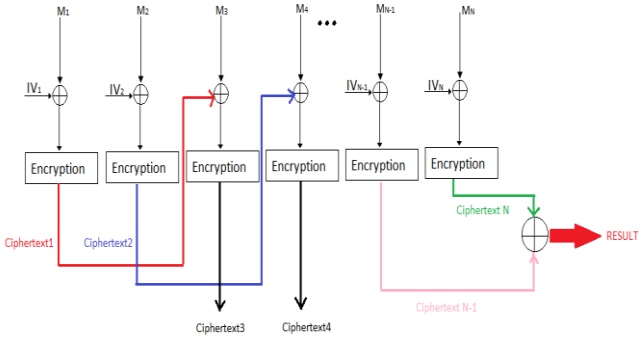

Figure 2. Interleaved Cipher Block Chaining

The programming models that we are using are CUDA, OpenCL, OpenMP and MPI. Each of them offers several advantages, but we want to establish which is the best for accelerating encryption algorithms.

The structure of AES implementation, in our project, is represented in Figure 3.

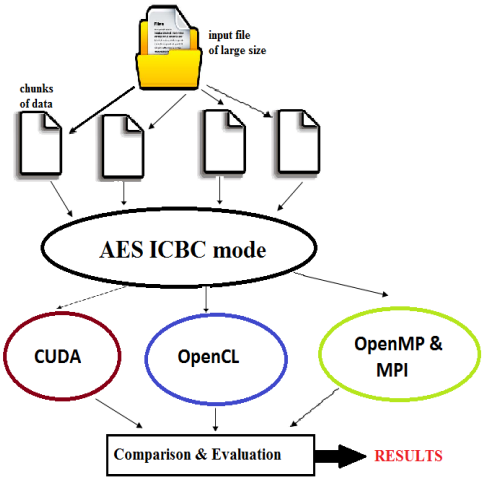

Figure 3. General architecture for AES implementation

There are two important elements to consider in our implementation: the data and the encryption process.

A. *Data*

The concept of data varies from simple things like personal information to the large amount of transactions used online. The quantity is not the same but the concept of privacy remains the same. The problems appear with high flow of data, real time processing becoming very hard to achieve. A solution proposed by us to obtain a certain degree of parallelism is to read buffers of fixed large size, thus limiting the system calls. The buffer is then used for obtaining data for encryption.

B. *Encryption process*

The algorithm chosen for analysis is AES. From initial configurations of the algorithms, we can create processes/threads/parallel regions, which can bring a certain degree of parallelism. This is done by separating the algorithms in the following steps: 1) generate the round key [12], 2) actual encryption (represented in Figure 4).

The generation of the keys can begin right after the reading *of* the data buffers and consists of three parts: key setup, key expansion unit, memory of internal (round) key.

Concerning the encryption process in ICBC-mode (represented in Figure 2) we use N streams of plaintext blocks, each of them encrypted independently with different initialization vectors (IV). The next set of block is encrypted right after the previous N have finished.

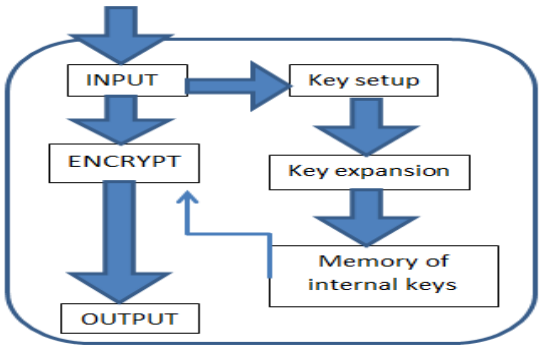

Figure 4. Basic Architecture

The equation that defines the process of encryption [12], for IBC mode is represented in Formula 1.

$$C_i=\text{Encrypt}(M_i, IV_i) \ : \ C_i=\text{Encrypt}(M_i, C_{i-1}) \quad (1)$$

In this way, the interleaved chaining recurrence is overlapped, permitting multiple encryption streams. These models of parallelism are relevant in the case of a large amount of input. We want to obtain results for different sizes of the input data and also to compare which of the used programming models is the best in terms of performance.

IV. IMPLEMENTATION

In this section we will describe the most important details regarding our implementation in OpenCL, OpenMP and CUDA and mention the problems and difficulties encountered during this process.

A. *OpenCL implementation*

One of the advantages brought by OpenCL is that it provides data parallelism. In our implementation, the input text is splitted into multiple parts depending on the number of work items found in a work group. In this situation, the workload is being processed in parallel by each Compute Unit of the OpenCL device.

There are some components that can be executed on the CPU device and, in our case, these are:

1) The input text (plaintext or chipertext) is loaded by reading the file directly using code lines for this action

2) The other elements necessary for the encryption/ decryption algorithm such as key, initialization vector are read also from files

3) The type of action desired is specified as a function in the main program.

4) All the S-boxes, P-boxes and other buffers required are transferred to global memory

The function that we have chosen to be executed by the OpenCL device is the function that performs the encryption in the program because we determined through several profiling tests that it is computational intensive and it consumes the most time of the total execution time.

A problem encountered during the implementation of this approach was the fact that that OpenCL's preprocessor doesn't play well with *#include*, but *include* directive worked nonetheless if the file was explicitly preprocessed in a Makefile via the classic c preprocessor (*cpp*) or included directly in the *.cpp* file.

Another difficult problem we had was the managing of all the functions that AES uses in the process of encryption. We decided that each of the operations of AES algorithm (substitute bytes, shift rows, mix columns and add round key) are serial functions called by the kernel.

B. OpenMP implementation

The first approach in parallelization was with one thread that would read chunks of data from the file, which is desired to be encrypted and 1, 2, 3 or 4 threads that would encrypt the data read by the main thread. This approach uses shared memory and the entire file was logically splitted at the number of encrypting threads at the execution level. In this way, when a part of the file was finished reading the thread, which would be mapped to that part of the file could start his task of encrypting the specific chunk of data. For notifying the thread of the availability of data, we needed to have one flag variable for every encrypting thread. The flag variables represent shared variables, which would be shared by all the threads.

With these shared variables it was encountered the need of the most common form of synchronization in threaded parallel programs, the ***mutual exclusion***. This synchronization method resolves the access to critical zone like the shared flag variables. OpenMP has support for safe access to a critical zone but the synchronization degrades the performance gained through parallelism.

In addition to mutual exclusion, this approach was not the best because in order for the encryption threads to get notified regarding data availability, they should do busy pooling on those variables. In this way, the time that could be gained will be lost because of busy waiting.

Because the busy pooling was not a good idea, the next step was to determine a way to read chunks of data and encrypt them in parallel without losing the time. The chosen approach includes using a lock variable for every encrypting thread and also splitting the file in *<number of threads>* logical parts. The time that was lost through the busy pooling is gained here by letting the main thread to continue reading, or letting other threads, which have available data to encrypt their chunks.

This is achieved because at the start of execution, the main thread sets all locks and when the encrypting threads start executing, they block by trying to set the already locked variable. The main thread unsets the locks when it succeeds to read some portions of data. From the OpenMP lock variables implementation is guaranteed that when a thread unsets a lock variable, one of the threads that are sleeping on the specific variable will be awaked and it will start to execute his task until it will be preempted by the scheduler.

C. CUDA implementation

For CUDA implementation because of some compatibility issues and because of different framework architecture, the AES has been implemented without using the OpenSSL library. The AES algorithm proposed has been done using as a base the two approaches presented in [12] and [14].

After the first step of the implementation, the AES algorithm, in order to parallelize and gain some time for encryption, has been modified to function in interleaved cipher block chaining mode. The gain in processing time was in terms of cost, similar to what we obtained in the OpenMP and in the OpenCL implementations. Another element that affects the cost of this solution is a longer IV that must be supplied to the algorithms as input in order to guarantee a stronger security. The cost is directly proportional with the gain desired because at N blocks being encrypted in parallel, the algorithm needs an IV of length N times longer than the standard IV (where the IV standard length is 16 Bytes).

Because of some implementation issues, this is the only approach that was implemented in CUDA, but for future research a data parallelization like the OpenMP approach can also be implemented.

V. PERFORMANCE EVALUATION

In this section we describe the performance evaluation of the AES algorithm taking into consideration the serial and the parallel implementations.

For evaluating the performance of the three parallel paradigms we ran tests on several sets of input files with various dimensions, the largest one having 1.4GB. The experimental results for the serial implementation are specified in Table 1 and a graphical representation of these results is presented in Figure 5.

TABLE I. SERIAL TIME RESULTS

FILE SIZE	TOTAL TIME	ENCRYPT TIME	READ TIME	WRITE TIME
1 MB	0,020	0,020	0,0001	0,0001
10 MB	0,160	0,060(37%)	0,050(31%)	0,040(25%)
100 MB	2,030	0,890(44%)	0,590(29%)	0,450(22%)

500 MB	10,962	5,061(46%)	2,981(27%)	2,521(23%)
1.4 GB	30,546	15,373(50%)	7,942(26%)	6,811(22%)

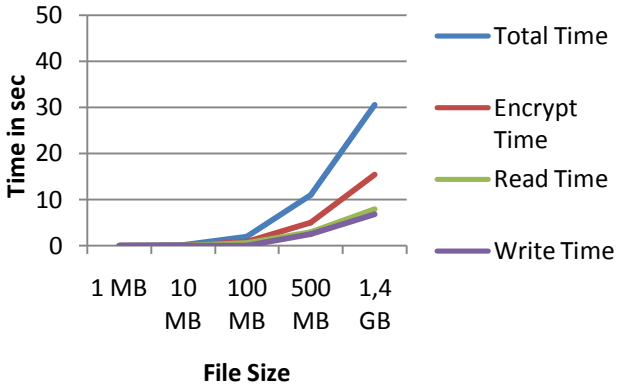

Figure 5. The plot of the serial implementation results

The serial results were used for the comparison with the parallel results obtained with OpenMP, CUDA, OpenCL and to determine the gain in speed and the advantages of using a parallel approach for encryption algorithms.

As it can be observed, as the size of the file grows, the time of encryption compared to I/O operations time tend to become equal but we have to be aware that the I/O operation consists of both reading the initial clear text and writing the cipher text after it was encrypted. Also being aware that in general I/O operations are very time consuming we can determine that the encryption is intense computational and consumes very much time from the total execution.

After the initial data and context was established, it was started the OpenMP documentation on how the parallel approach for AES algorithm could be done.

The results specified in Table 2 correspond to OpenMP's implementation when we used five threads in the encryption function.

TABLE II. OPENMP TIME RESULTS

FILE SIZE	TOTAL TIME	ENCRYPT TIME	READ TIME	WRITE TIME
1 MB	0,030	0,030	0,00001	0,00001
10 MB	0,160	0,066	0,040	0,039
100 MB	1,8	0,890	0,490	0,450
500 MB	9,4	5,55	1,8	2,0
1.4 GB	25,51	15,373	5,832	4,31

Comparing the serial results with the OpenMP results we obtain the graphic shown in Figure 6. Observing the graph we can determine that as the size of file grows the time gained by the parallel implementation grows also. These parallel results were tested with 5 threads. There were some test with 2 threads but the gain was little so the 5 thread results are the most important to show.

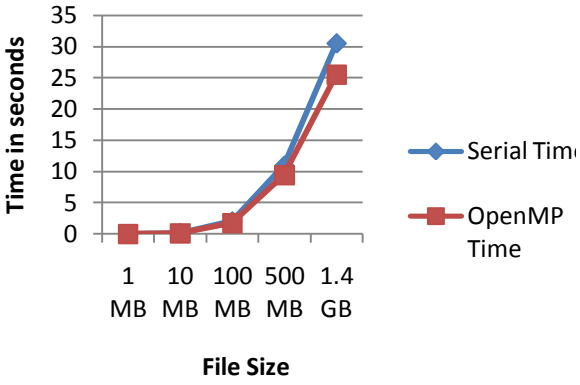

Figure 6. OpenMP versus serial implementation

Then we tested our AES implementation in OpenCL using the same set of input data and we obtained the results presented in Table 3.

TABLE III. OPENCL TIME RESULTS

FILE SIZE	TOTAL TIME	ENCRYPT TIME	READ TIME	WRITE TIME
1 MB	0,001	0,001	0,00001	0,00001
10 MB	0,050	0,030	0,010	0,010
100 MB	0,251	0,100	0,121	0,130
500 MB	5,02	3	1,52	0,5
1.4 GB	18	15	2	1

The graphic in Figure 7 emphasizes the fact that a parallel approach such as OpenCL gains time more than the serial implementation when the file size is growing, ensuring in this way the acceleration of the AES cryptographic algorithm.

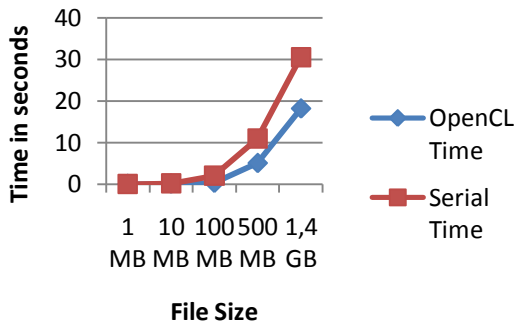

Figure 7. OpenCL versus serial implementation

The last implementation was realized in CUDA, for the same set of input data as before. The performance obtained for every file we tested can be observed in Table 4.

TABLE IV. CUDA TIME RESULTS

FILE SIZE	TOTAL TIME	ENCRYPT TIME	READ TIME	WRITE TIME
1 MB	0.0005	0.0005	0,00001	0,00001
10 MB	0.01	0.005	0.0025	0.0025
100 MB	0.0148	0.0071	0.0041	0.0036
500 MB	1.04	0.71	0.2	0.13
1.4 GB	11.29	9.56	1	0.73

In Figure 8, we can observe the comparison between the serial and CUDA's results. It can be seen that the best speedup of 2.70 was obtained for this implementation.

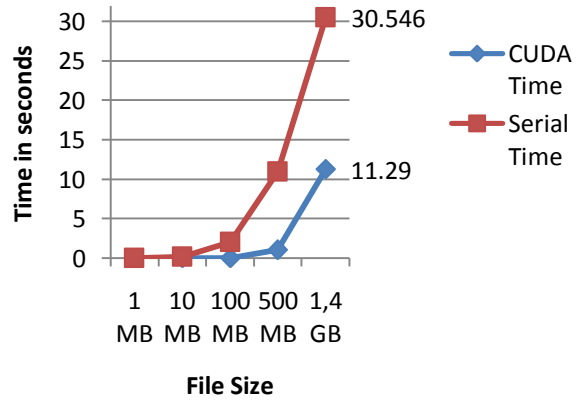

Figure 8. CUDA versus serial implementation

VI. CONCLUSION

In this paper we presented our implementation of the symmetric block cipher AES using CUDA, OpenMP and OpenCL. We chose this algorithm because it is a standard encryption algorithm implemented in the OpenSSL cryptographic library. Using the OpenSSL library as a support for parallel implementations makes this cipher available to software that already uses OpenSSL with very little effort.

The aim of this paper was to show the remarkable reduction in encryption and decryption time of cryptographic systems when using ICBC mode of AES and also when we have serial implementation versus parallel implementation. These preliminary, very good results, can lead to better performances on GPUs after a further optimization of the source codes.

Our future work will involve the following activities: optimization of AES code for every parallel approach used, development of the OpenCL, CUDA and OpenMP implementations of all cryptographic algorithms existent in OpenSSL that would lead into a guideline for evaluation and development of cryptographic algorithms on GPU platforms.

REFERENCES

[1] David B. Kirk, Wen-Mei W. Hwu , Programming Massive Parallel Processors,Elsevier, 2010, pp. 100-200.

[2] Matthew Scarpino , OpenCL in Action, O'Reilly Media, 2011, pp. 300-420.

[3] Wlodzimierz Bielecki, Darius Burak ,Parallelization of the AES Algorithm , Springer US, 2008, pp. 191-204.

[4] Praveen Dongara and T. N. Vijaykumar, Accelerating Private-Key Cryptography via Multithreading on Symmetric Multiprocessors, Springer, 2005, pp. 213-254 .

[5] Ted Huffmire, Application of Cryptographic Primitives to Computer Architecture, Springer, 2010, pp. 139-152.

[6] J. John Raybin Jose and E. George Dharma Prakash Raj, A Survey on the Performance of Parallelized Symmetric Cryptographic Algorithms,in *International Journal of Research and Reviews in Computer Science (IJRRCS)* , June 2012.

[7] Osvaldo Gervasi, Diego Russo, Flavio Vella, The AES Implementation based on OpenCL for Multi/Many Core architecture,in *International Conference on Computational Science and Its Applications*, 2010.

[8] M. Loukides and J. Gilmore, Eds., Cracking DES: Secrets of Encryption Research, Wiretap Politics and Chip Design. Sebastopol, CA, USA: Electronic Frontier Foundation , O'Reilly & Associates, Inc., 1998.

[9] T. Guneysu, T. Kasper, M. Novotny, C. Paar, and A. Rupp, Cryptanalysis with COPACOBANA, IEEE, Trans. Comput., vol. 57, , 2008, pp. 1498-1513 .

[10] Kahraman Akedmir et al, Breakthorugh AES performance with Intel AES new instructions, Intel, 2010.

[11] Raymond Keith Scott Manley, Program generation for Intel AES new instructions, *thesis submitted for the degree of Doctor in Philosophy*, 2011, pp. 70-80.

[12] Kris Gaj, Pawel Chodowiec , Hardware performance of the AES finalists - Survey and analysis of results, *Technical Report, George Mason University*, 2000.