ÇANKAYA UNIVERSITY
THE GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES
COMPUTER ENGINEERING

MASTER THESIS

PARALLEL IMPLEMENTATION OF AES ALGORITHM USING CUDA & MPI

ÖZGÜR PEKÇAĞLIYAN

SEPTEMBER 2013

Title of the Thesis: **Parallel Implementation of AES Algorithm Using CUDA & MPI**

Submitted by **Özgür Pekçağlıyan**

Approval of the Graduate School of Natural and Applied Sciences, Çankaya University

[illegible]

Prof. Dr. Taner Altunok
Director

I certify that this thesis satisfies all the requirements as a thesis for the degree of Master of Science

[illegible]

Assist. Prof. Dr. Murat Saran
Head of Department

This is to certify that I have read this thesis and that in my opinion it is fully adequate, in scope and quality, as a thesis for the degree of Master of Science.

[illegible]

Assist.Prof.Dr. Nurdan Saran
Supervisor

Examination Date: 17.09.2013

Examining Committee Members

Assist.Prof.Dr. Abdül Kadir Görür	(Çankaya Univ.)	[illegible]
Assist.Prof.Dr. Nurdan Saran	(Çankaya Univ.)	[illegible]
Prof.Dr. Mehmet Reşit Tolun	(TED Univ.)	M.R. [illegible]

STATEMENT OF NON-PLAGIARISM

I hereby declare that all information in this document has been obtained and presented in accordance with academic rules and ethical conduct. I also declare that, as required by these rules and conduct, I have fully cited and referenced all material and results that are not original to this work.

Name, Last Name	: Özgür Pekçağlıyan
Signature	: [illegible]
Date	: 24.09.2013

ABSTRACT

PARALLEL IMPLEMENTATION OF AES ALGORITHM USING CUDA & MPI

PEKÇAĞLIYAN, Özgür

M.Sc., Department of Computer Engineering

Supervisor: Assist.Prof.Dr. Nurdan Saran

September 2013, 72 pages

According to today's standards, life goes online. People do their shopping and buying electronics through online stores. They date online and banks are transferring money online. Even, bachelor courses are online [10, 37]. Also, governments may keep their sensitive data such as tactical information for troops or messages for embassies on cloud computing systems which might be located on other countries.

Because of its sensitivity, these type of data must be protected from unauthorized access and its integrity should be guaranteed [37]. Cryptography is based on mathematical techniques which concentrated on data confidentiality, integrity and origin authentication [41]. Advanced Encryption Standard (AES) is the national standard of U.S. which is accepted by U.S. government on October 2000 [19, 46]. Encryption is a good way to protect data integrity and confidentiality. Still, encryption requires time and computation power. Today, computers have reached high clocking speed measured by Gigahertz. If one tries to encrypt a data over 1GB it could take more 10 than minutes to finish the operation. Upon thinking of computers, they come with multiple processors. Also, today we have very expensive GPUs installed in our computer cases. These GPUs are almost powerful than CPUs. There are several libraries to get full advantage of CPUs and GPUs. Two examples for these libraries are OpenMPI and CUDA. While OpenMPI allows developer to use all CPUs parallelly, CUDA allows developer to submit his/her code to run on GPU. The application running on GPU might be a serial application or parallel application divided to GPU cores [13, 37]. This study aims to paralleling AES algorithm using both OpenMPI and CUDA libraries and comparing time differences between these two methods and classical serial method on a CPU.

Keywords: AES, Encryption, CUDA, MPI, Parallel Programming

ÖZ

AES ALGORİTMASININ CUDA & MPI KULLANILARAK PARALLELLEŞTİRİLMESİ

PEKÇAĞLIYAN, Özgür

M.Sc., Bilgisayar Mühendisligi Bolumu

Tez Yöneticisi: Assist.Prof.Dr. Nurdan Saran

Eylül 2013, 72 pages

Bugünün standarlarına göre hayat online olarak ilerlemektedir. İnsanlar mutfak alışverişlerini, elektronik eşyalarını internet üzerinden satın alıyorlar, internet üzerinden arkadaşlık kurup, flört ediyorlar, bankacılık işlemlerini online olarak gerçekleştiriyorlar. Hatta, bazı internet siteleri üzerinden, online dersler verilerek insanların lisans eğitiminde ihtiyaç duyduğu dersleri önceden almasının imkanı sağlanıyor [10, 37]. Bazı devletler, uzak ülkelerdeki askeri birliklerine veya elçiliklerine gönderilmesi gereken hassas bilgilerini internet üzerinde bulunan bulut çözümlerine yükleyebiliyorlar. Bu verilerin, hassaslıkları nedeniyle, yetkisiz erişime karşı korunması gerekiyor [37].
2000 yılının ekim ayında, Amerikan hükümeti tarafından ulusal standard olarak AES algoritması kabul edilmiştir [19, 46]. Kriptografi, temel olarak, verinin butunluğünü, gizliliğini ve kaynağınının doğrulanmasını hedefleyen matematik metodlardan oluşmaktadır [41]. Kriptografi, verinin gizliliğinin ve bütünlüğünün korunması icin guzel bir yöntem olmasına rağmen yine de zaman ve hesaplama gücüne ihtiyaç duymaktadır. Günümüzde bilgisayarlar Gigahertz olarak ölçülen yuksek işlemci frekanslarına ulaşmışlardır. Ancak, boyutu 1GB'dan büyük bir dosya şifrelenmek istendiginde, yaklaşık olarak 10 dakikadan daha uzun bir süreye

ihtiyac duyulabilmektedir. Bankacılık işlemleri göze alındıgında ise, zaman, pahalı bir kaynaktır. Ayrıca bireyler, günümüzde oldukça pahalı ekran kartları kullanmaktadırlar. Bu ekran kartlarından bazıları neredeyse normal işlemcilerden bile daha güçlüdürler. Bu donanımları (ekran kartları ve işlemciler) kullanabilmek için birtakım hazır kütüphaneler bulunmaktadır. OpenMPI ve CUDA bu kütüphanelerden ikisidir. OpenMPI, programcıya CPU üzerinde kodunu parallel olarak çalıştırma ve bilgisayarın bütün işlemcilerini kullanma imkanı sunarken, CUDA ise, aynı kodun ekran kartı üzerinde çalıştırılmasına imkan saglamaktadır. Ekran kartı üzerinde çalıştırılan bu kod, bütün çekirdekleri kullanarak, paralel olabilecegi gibi, seri olarak da geliştirilebilir [13, 37]. Bu çalışmanın amacı, AES algaritmasını hem OpenMPI, hem de CUDA kütüphanelerini kullanarak paralelleştirmek ve AES'in orjinal seri kodu ile paralelleştirilmiş kodları çalışma süreleri açısından kıyaslamaktır.

Anahtar Kelimeler: AES, Sifreleme, CUDA, MPI, Paralel Programlama

ACKNOWLEDGMENTS

I would like to express my sincere gratitude to my supervisor Assist.Prof.Dr. Nurdan Saran for her patience, advice, criticism and encouragements throughout this research.

I would like to thank Assoc.Prof.Dr.Cem Ozdogan for his guidance when I was an undergraduate student. He helped me to decide my field of study.

I wish to thank my family and my friends for their support and patience throughout my research.

Finally, I would like to thank the person I love. If it wasn't for her, I wouldn't be studying.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER I

INTRODUCTION

People live in a technological world. Everyday, news have been announcing that some people have been wired or listened by some other people. Today, it is easily possible to intercept a mobile phone communication or read others e-mails. People can think that their mails are secure on the providers servers. Provider may have taken necessary precautions, still, people may be victim of the man-in-the-middle attacks and their mails can be read during the transfer operation to mail server. In order to prevent eavesdroppers from reading e-mails or listening phone communications, one may use encryption algorithms [16].

1.1 ENCRYPTION

Encryption is a part of cryptography. Cryptography is the principle to convert a data to something which is almost impossible to reverse back to original form by unauthorized people. Cryptography is focused on 4 goals;

- **Confidentiality:** Data should be kept away from unauthorized eyes, it may include sensitive information and it may turn to be harmful to owner, if it it is exposed by other people.
- **Data Integrity:** Data integrity is also an important matter. For example, if someone changes some other people's medical record, they can have false treatment or no treatment at all. This might cause casualties of innocent people.
- **Authentication:** Money transactions are go through online. People have teleconferences and have mails over Internet. The origin server, sender, receiver and the data itself should be authenticated as permitted sender, receiver etc...

- **Non-Repudiation:** Operations shouldn't be deniable by the entities. For example, assume a money transaction over Internet, later on, sender claims that he/she didn't execute or approve for the transaction. In the other hand, receiver would be on the position of claiming that the transaction was approved by the sender. To be able to solve this kind of conflicts, procedures should involve a third party person or institution for transaction to investigate operation. Encryption might be used to solve this kind of conflicts.

Encryption is the method which used to convert data into an unreadable form (cipher) [21, 41]. Decryption is the reverse operation of encryption. Encryption and decryption operations can be seen on Figure 1.1. Here plaintext is the input data as being converted to ciphertext (encrypted) using an encryption function. Later on, ciphertext is used as an input for decryption function and the output is same as the input which is the string "Hello World!".

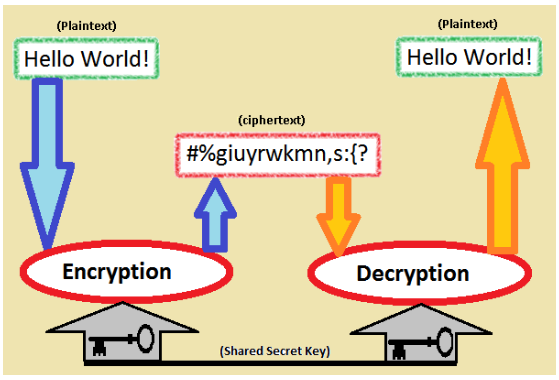

Figure 1.1: Encryption and Decryption [54]

There are basically two methods for cryptography. Symmetric and Asymmetric cryptography. In symmetric or shared key cryptography, there is only one key is being used for both encryption and decryption. Assume that lookup table in Table 1.1 is being used for encryption. With a specific key and using lookup table, it is possible to generate ciphertext. If the key is given, it is also possible to regain plaintext back. For example;

PlainText: It is rainy today.
Key: QWERTYUIOPASDFGHJ
CipherText: ZQ YM MJXDZ XUKII

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A
B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B
C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C
D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D
E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E
F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F
G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G
H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H
I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I
J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J
K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K
L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L
M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M
N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N
O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P
Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q
R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R
S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S
T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T
U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U
V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V
W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W
X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X
Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y
Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z

Table 1.1: Basic Lookup Table

In this example, spaces are ignored with the keys matching with space character. Crossing of other characters with their matching keys on lookup table have been found and assembled together to generate ciphertext. After getting ciphertext, if the key is known, it is possible to regenerate plaintext. Only thing which should be done is that checking key rows for the matching ciphertext columns [11, 21, 43].

This is the most basic encryption method and the most basic lookup table. Basically, encryption is just merging key with the plaintext. This can be also done with a lookup table, XOR'ing them or with any other specific merging method. The key point of shared key encryption is to use the same key for both decryption and encryption. Procedure can be seen on Figure 1.2.

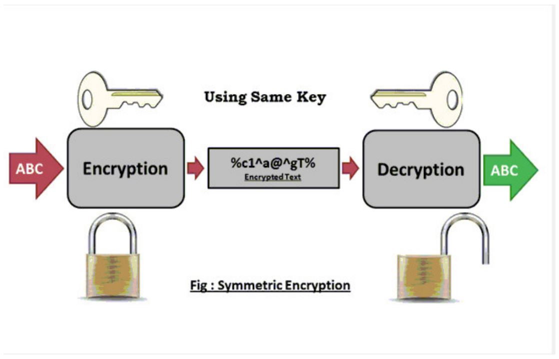

Figure 1.2: Shared Key Encryption [43]

In the other hand, asymmetric or public key cryptography, in simple manners, is based on different keys for encryption and decryption. Basically a person, who wants to communicate secretly, generates 2 different keys in his/her private and secure computer. One key is for encryption and the other one is for decryption. Later on, they will be called public and private keys. Person announce one of his keys for encryption on his/her own website for others use. That is the public key. Anyone can encrypt any message using the public key and then send to receiver. Only the person who has private key (key for decryption) can decrypt and read the message [26, 43]. Procedure is described simply on Figure 1.3.

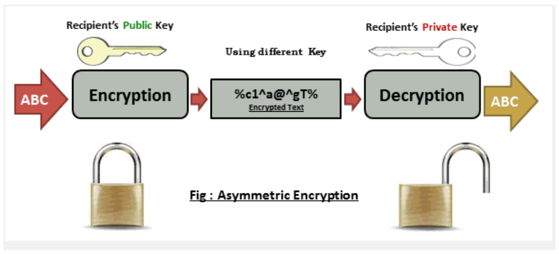

Figure 1.3: 2-Key Encryption [43]

1.1.1 HISTORY OF ENCRYPTION

History of encryption goes back to Egyptians. An Egyptian draw the hieroglyphs to his lord's tomb which tels a story about the lord's life. The Egyptian's intention was not to encrypt the writing. He/She probably just wanted to tell the story in a fascinating way. He/She wrote "In the year of Our Lord One thousand eight hundred and sixty three" instead just "1863". Encryption also didn't grow steadily later on. It only appeared in books and literature works. Invention of electronic communication has increased the development of cryptography. Most of the countries started to intercept radio and the espionage was a matter of fact.

Cryptography had been used during WW1, WW2 and cold war era by the military of almost every country. It is also known that, today, military is both using codes for communication with each other and encrypting the whole communication. Till the late 20th century, cryptography was only available to use of governments and military [12, 45].

In 1973 Horst Feistel has described cryptography in his paper [11] and in 1976 Diffie Whitfield and Martin E. Hellman have announced public key (2-key) authentication in their paper [26].

In 1984 Phil Zimmermann, who was a software engineer working for military and as a freelance developer, decided to develop PGP (Pretty Good Privacy). His intention was to protect rights of every human being on earth and he published the tool on Internet in 1991. During that time, according to U.S. laws, PGP was considered a munition and spreading it without a license has been accepted as illegal exportation of a military equipment. This put Zimmermann to target of three-years long criminal investigation. After these years, Zimmermann didn't found guilty and PGP became most widely used encryption software around the globe [42].

In 1997, U.S. government had announced a competition in order to select national encryption standard. Joan Daemen and Vincent Rijmen were applied to competition. Their algorithm was named after their names as Rijndael. U.S. National Institute of Standard and Technology (NIST) invited the cryptology community to perform attacks on candidate algorithms.

On October 2000, Rijndael algorithm officially selected as the winner and announced as Advanced Encryption Standard (AES) without any modification [37, 46].

1.1.2 AES

AES is based on Rijndael algorithm almost without any differences. Like all other encryption algorithms, AES is also based on mathematical techniques but still it has one of the simplest algorithms to understand. The main difference between AES and Rijndael is the block length. AES requires block length to be fixed to 128 bits while Rijndael allows multiples of 32 bits between 128 and 256 to be used as block length. In the other hand, key lengths are still multiples of 32 bits between 128 and 256 for both Rijndael and AES [46]. AES (or Rijndael) is a serial algorithm based on data blocks. It is described in the book [46] as "Rijndael is a key-iterated block cipher: it consists of the repeated application of a round transformation on the state". Round numbers are determined according to block length and key length. For example, if both key and block lengths are fixed to 128 bits, there should be 10 rounds to be executed. Rounds are sequenced as four transformations. These transformations are called steps. These steps are;

- **SubBytes:** This step is the only nonlinear part of the cipher. During SubBytes step, data block is being matched with the lookup table (S-Box) and the data in the block is being replaced with the matched values.
- **ShiftRows:** Every row should be shifted left with different offsets. Row 0 should be shifted C^0 bytes, row 1 should be shifted C^1 bytes and so on... If the block length has chosen as 128 bits then the offset values should be 0, 1, 2 and 3 respectively.
- **MixColumns:** There is a matrix multiplication of each column of data with the corresponding lookup table. Result of the multiplication is written on the place of the column which has been used [46]. MixColumns step is being applied on to data in first 9 steps but it is not applied in 10th step [7].
- **AddRoundKey:** The key is being added in the AddRoundKey step. The data is being XORed with a round key [37, 46].

A simple illustration of rounds in AES can be seen on Figure 1.4.

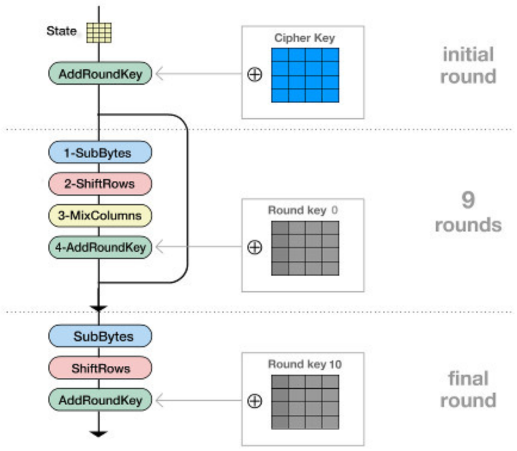

Figure 1.4: AES Rounds [7]

1.1.2.1 MODES OF OPERATION

Normally, plaintext with n-bit fixed-size blocks are being encrypted. For AES, block-length has been fixed to 128 bits, so, n is 128. If the size of data expands 128 bits (16 bytes), then, data is being divided into blocks with size of 128 bits. Then, each block should be encrypted separately [21]. Some of the examples of modes of operations are;

- **Electronic Code Book Mode (ECB):** ECB is the simplest one of the modes of operations. Divided blocks are just being encrypted separately [21, 44]. ECB is very simple and very easy to parallelization. Yet, its disadvantages tempts people to use other modes of operations. Every bit will be encrypted in the same way, so, same values in the data will be encrypted to same results. If someone do some statistical work on the encrypted data. it is possible to find plaintext without the key or it is possible to determine the key. ECB is very suitable for parallel implementation [22] as it can be seen on Figure 1.5.

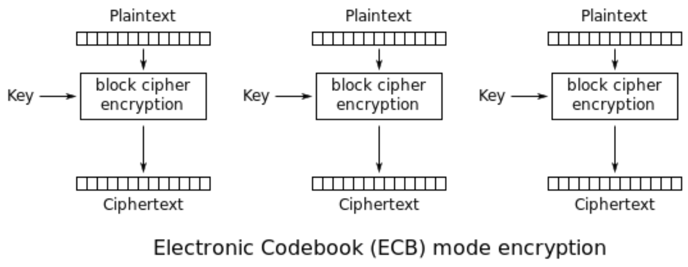

Figure 1.5: ECB Flow Chart [54]

- **Cipher Block Chaining Mode (CBC):** First of all, each data block is being XORed with initialization vector and the results enter encryption algorithm among with the key. Output is being used for the next initialization vector for the next data block and it goes so on. When every block has been encrypted, outputs of the operations are put together to generate ciphertext [21, 44]. Flow chart can be seen on Figure 1.6.

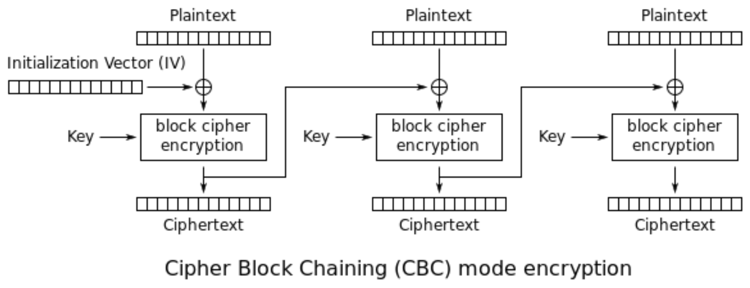

Figure 1.6: CBC Flow Chart [54]

- **Cipher Feedback Mode (CFB):** CFB has a different point of view than ECB and CBC. Instead of the data blocks, initialization vector has been encrypted during the procedure. Result of the encryption is XORed with the data block. Output will be used as initialization vector for the next data block and so on. Outputs are combined to produce ciphertext [21, 44]. Procedure is described in Figure 1.7.

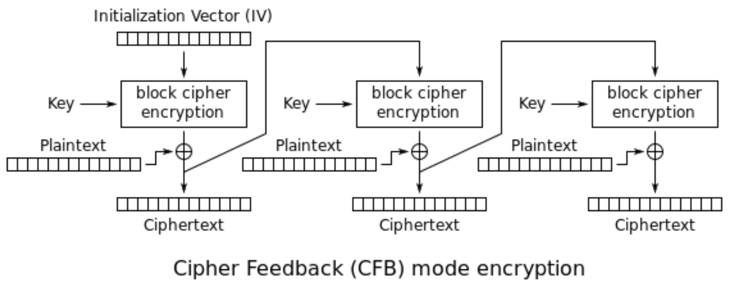

Figure 1.7: CFB Flow Chart [54]

- **Output Feedback Mode (OFB):** OFB works almost as same as CFB. There is slightly one difference. After encryption of the initialization vector, before XORing result with data block, result is also used as next initialization vector for next data block at the same time [21, 44]. Procedure and the data flows can be seen easily on Figure 1.8.

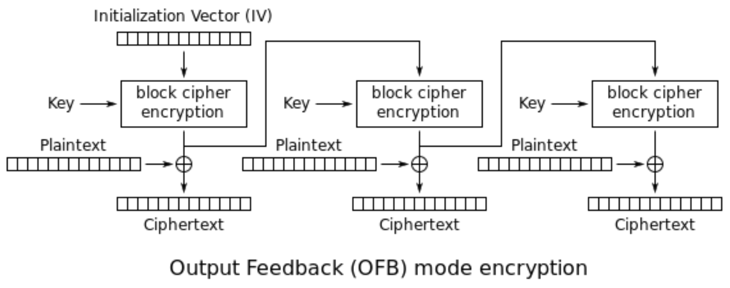

Figure 1.8: OFB Flow Chart [54]

- **Propagating Cipher-Block Chaining Mode (PCBC):** PCBC is a bit complicated. Before encryption, plaintext and initialization vector are being XORed and then they enter encryption. Result of the encryption procedure is a part of the ciphertext and it is also being XORed with plaintext to be initialization vector for next block [22, 44]. Procedure is described simply on Figure 1.9.

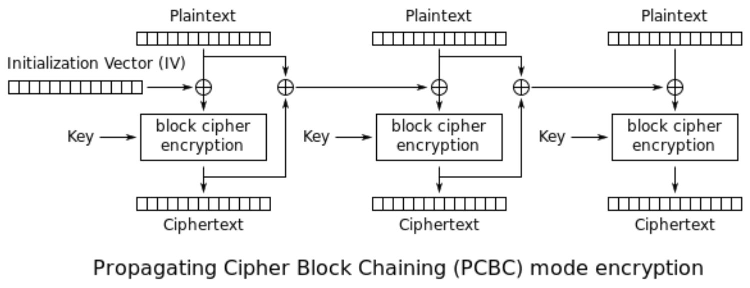

Figure 1.9: PCBC Flow Chart [54]

- **Counter Mode (CTR):** CTR requires a different counter value for each data block. This value should be combined with the initialization vector somehow, this combination can be earned by XOR'ing values or concatenating them. After combining values, the new counter value should be encrypted with the key and the result of encryption should be XORed with the data of the block in order to get ciphertext [9, 22, 37, 44]. Simple flow chart can be seen on Figure 1.10.

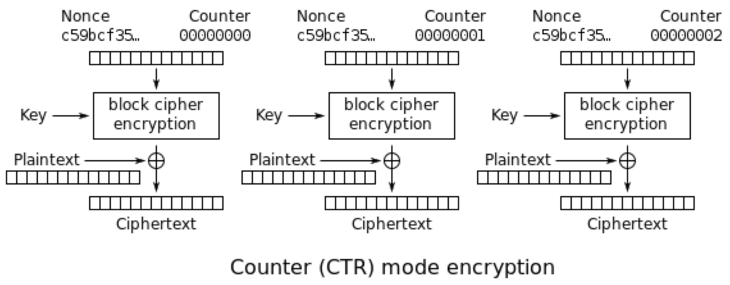

Figure 1.10: CTR Flow Chart [54]

It can be concluded that CTR Mode and ECB Mode are very suitable for parallel implementation. Rest of the modes are incompatible for parallel implementation because of their serial dependencies. In the modes (except CTR and ECB), every block should wait previous blocks to be processed because output of the previous blocks will be input for the current block. Kipper et al have also mentioned same thoughts in their page [22].

1.2 PARALLEL COMPUTING

Softwares used to developed in a serial manner. During these days multiprocessors performance were increasing 50% every year. After 2002, technology nearly reached its theoretical limits and development is slowed. Manufacturers changed their vision from single serial processors to multiple processors on a single circuit. But, increasing number of CPU doesn't affect the performance of softwares. Traditionally, softwares have been developed for a single CPU. They are not aware of multiple CPUs. Operating system's scheduling algorithms distributes different processes to different microprocessors. Even dough, every process runs sequentially on the assigned processor [23]. Parallel Programming allows developers to get full advantage of every microprocessor to solve the problems. Simply, parallel computing is taking advantage of every hardware which has compute capacity to solve a problem [25]. For example;

Assume that, there are one billion marbles and someone has to count them all. If the person tries to count them all just by himself, it might take months finish the operation. Instead, if the person calls his/her friends for help, divides a group of marbles to each of his/her friends and than each friend count the number of marbles in their own group and later, if the person sums all of the counts, it might just take hours to finish the work. Dividing marbles is an example of parallel computing.

Not every problem can be solved parallelly. Nature of the problem or solution should be suitable for parallelization. For example, reading a specific book can't be done parallelly. One can't ask his/her friends to read different chapters and explain himself/herself. Pages of the book should be read one by one in order to understand its content. It can't be parallelized. Situation is same for any problem. If a math problem in order to be solve requires other values to be found, then, it can't be parallelized. Still, most of the problems can be parallelized. Today, factories runs parallel, they produce many products in the same time. Weather changes parallelly, it might be rainy in London but in the mean time it might be sunny in Sydney.

Need of knowledge is increasing constantly and problems are becoming very complex. Parallel computing has many fields of study. It has been widely used in research projects. There are also examples of commercial usage. Main idea of the parallel computing is to solve hard problems in an acceptable time. Main advantage of the parallel computing is the reduction of time cost with a cost effective solution [25]. In order to achieve high speed computing power, there are some different approaches.

- **Multi-threaded Applications:** Today, operating systems are capable of executing several jobs in the same time with the help of scheduling algorithms. Every thread of the application are sharing same resources (CPU, memory etc) with a unique thread id of their own. Many applications use multi-threaded solutions, such as web browsers, instant messaging applications etc... Scheduling algorithm switch between these threads very efficiently, so, it is assumed that they run parallelly. In order to run threads parallelly, there should be a multi-core CPU and also OS scheduling algorithm should be able to manage multi-core CPUs [35]. This can be assumed as first attempts of parallel computing.

- **Cluster:** A cluster (or beowulf cluster) is a group of computers connected over a network (Ethernet, wifi etc). Computers in a cluster are called nodes. These nodes have been managed by a master/head node. People submit their jobs to head node and the head node divides jobs to available nodes, then collects results back [18, 48]. Clusters can be made out of junk computers, old PCs haven't been used daily or single board computers. A three-node home-made cluster made out of RaspberryPi single board computers can be seen on Figure 1.11 as an example.

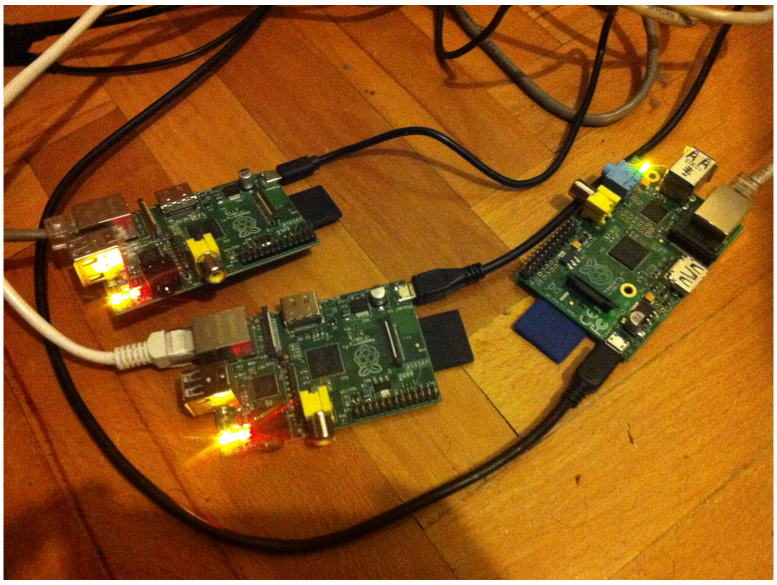

Figure 1.11: Home Made Cluster

- **Supercomputers:** Supercomputers are single machines with multiple processors and high computing capacity. They are much more expensive systems when compared to clusters [55]. Today, worlds most powerful supercomputer is Tianhe-2 which has been developed by China's National University of Defense [49].

- **Volunteer Computing:** There are lots of machines connected on Internet and most of them just in idle position most of the time. Volunteer Computing is focused on these idle times. In order to achieve volunteer computing, user should download an install client software to his/her computer. Software keeps running in background. When user stops using computer and computer enters idle mode, software becomes activated and receives packages through Internet, analyses packages and calculates results, then, sends findings back to server [8, 51]. There are some volunteer computing projects like seti@home which is aimed to find alien life forms [8].

Figure 1.12: Seti@Home Logo [8]

- **GPGPU:** Another approach of parallelization is using GPUs. Gaming industry has driven GPU technologies to be improved. Require of 3D, real-time, HD etc has forced GPUs to be multi-core processors, highly capable of parallelization and multi-threaded applications. In the other words, GPUs are specialized to run programs in parallel which are required to execute same instructions many times [13, 30].

- **Cloud Computing:** Cloud Computing actually used for sharing expenses with smaller companies. Companies with gigantic data centers have lots of expenses to operate it. To minimize expenses, companies rent available storage or service to other companies whom doesn't have any data centers [1, 53].

That much of computing power is usually used for projects like modeling, medical identification, scientific research, weather forecasting, alien life form searching, data redundancy etc [25]. Also, multinational companies may require power of parallel computing in other means. For example, companies like search engines need to have distributed server systems in order to keep their uptime high, serve as many customers as possible in the same time and survive from aggressive attacks like DDOS without losing control of the whole system [20].

Parallel computing is not a new concept. Researches have been started in 50s. But, till 2000s, CPUs were enough to respond performance requirements and there were no need for parallel computing. Reduction of time frequency in the development of CPUs had forced people to consider parallel computing [23, 47, 52]. During these years, parallel computing used to require supercomputers, computer clusters or expansive PCs with multi-core CPUs. In 2010, almost every home user had PCs with multi-core CPUs. This swept off the need for gigantic mainframes. Till 2006 every computation were done using CPUs. GPUs were only for processing and computing graphical operations. In November 2006, Nvidia introduced CUDA architecture for General Purpose Graphics Processing Unit (GPGPU) [13, 30]. This was another milestone for parallel computing. Almost after 1.5 years, Apple proposed OpenCL working group as an alternative of CUDA in June 08. Later, Khronos group has announced OpenCL 1.0 in December 08. So, OpenCL became alternative for proprietary libraries [24, 36]. Main difference between CUDA and OpenCL is that CUDA is specifically developed for Nvidia GPUs while OpenCL is compatible with any GPGPU vendor.

1.2.1 MPI

Message Passing Interface (MPI) is a standard for developing applications which are passing messages between each other. It has been defined by MPI forum. MPI is not a programming language instead it's a standard definition. So, MPI is defined as functions and implemented by different groups for different kind of languages like C/C++, Fortran etc. There are different libraries which implement MPI standard such as OpenMPI and MPICH. Because of its design of portability, applications developed according to MPI standard can be executed on any environment which has a MPI Library installed on it. This means that an application which has been developed for distributed memory parallel computers (such as clusters) can also be executed on a shared-memory computer (such as a multi-core CPU computer) [6, 27, 28]. Basic journey of a MPI application which has been submitted to a cluster, can be seen on Figure 1.13. During process, each node (CPUs or CPU cores) is executing very same serial code simultaneously. It is possible to execute different parts of the code on different nodes with specific keywords and functions. While execution of application, every node might be sending messages to each other till application complete its calculations [27].

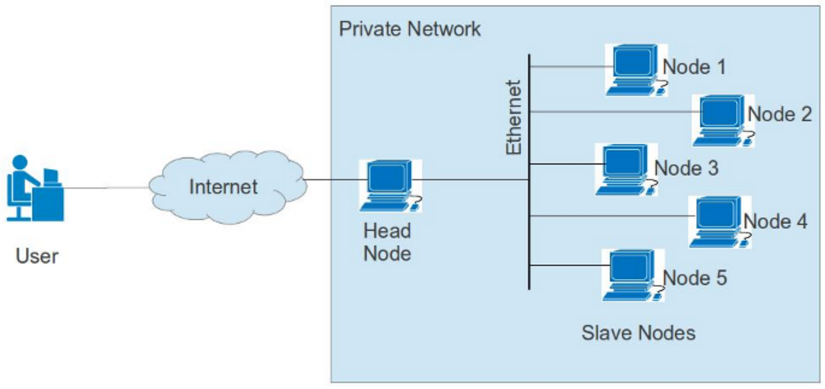

Figure 1.13: Basic Cluster

1.2.2 CUDA

CUDA is a parallel programming library which extends C to solve problems on a GPU more efficiently than on a CPU. Unlike OpenCL, CUDA can only be designed to work with Nvidia GPUs. Nvidia introduced CUDA in 2006. It has been widely used by researchers and developers since its announcement. Applications developed using CUDA are capable of being both serial and parallel. Just like all other applications written in C, CUDA applications also have a *main* function. The main function is the heart of the application and run by CPU. In terminology of CUDA, CPU part of the execution is called "Host" and GPU part is called "Device" and functions executed on GPU are called *kernel.*

Kernel is run by GPU and divided into blocks and threads. Blocks are parallel copies of kernel. Threads are located in a block with x and y coordinates, blocks are also located in a grid with x and y coordinates. There can be 8 blocks per streaming multiprocessors. Each block may have threads and these threads also runs simultaneously. Threads have access to shared memory. Shared memory is visible to every thread in the same block. Blocks have access only constant and global memories [13, 31, 15, 29]. CUDA memory architecture can be seen on Figure 1.14. CUDA grid and block architecture can be seen on Figure 1.15.

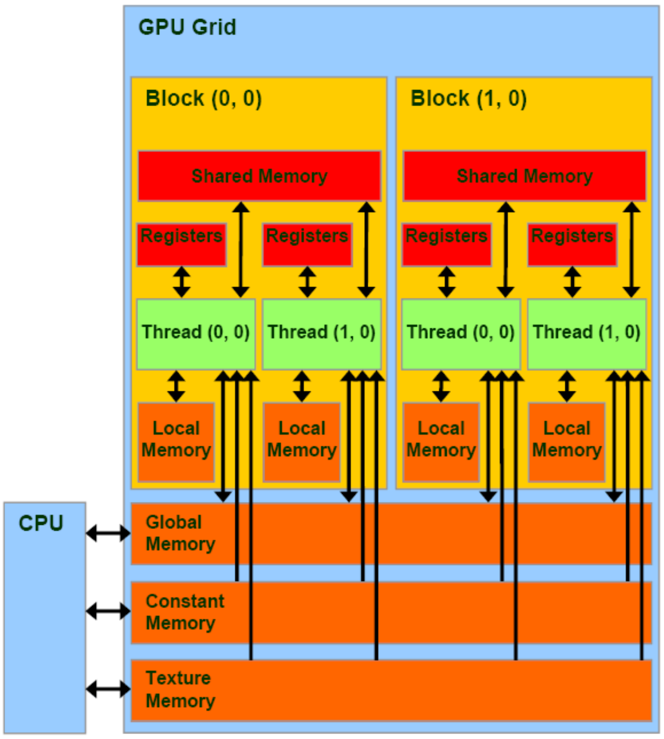

Figure 1.14: CUDA Memory Diagram [34]

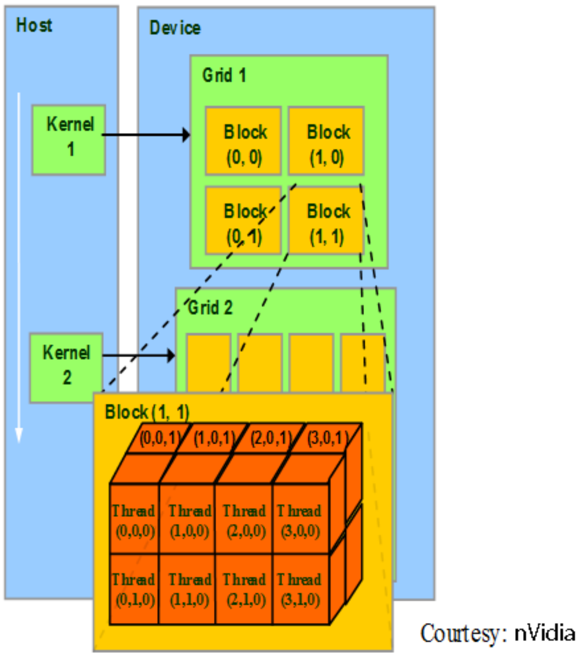

Figure 1.15: CUDA Grid Schema [3]

1.2.3 BACKGROUND

Michael Kipper et al ported an open source CPU implementation of AES algorithm to GPU. They used 128-bit AES for their testing, divided plaintext to 128-bit blocks and each block encrypted/decrypted parallelly. In order to boost the performance, they have done keyExpansion on CPU. They have encrypted and decrypted 64 MB size of message 200 times in order to measure the performance of algorithm. As a result, they have processed a total of 25.6 GB data in 23.187 seconds with a 9.26 Gbps throughput. Which means, research was able to accomplish 14.5x speedup when compared to CPU implementation [22].

Deguang Le et al are sending plaintext directly to kernel. In kernel, it is divided to blocks and each thread computes one AES block. Finally, encrypted blocks are combined on CPU to generate ciphertext. Implementation was able to achieve 7x speedup over a comparable CPU [38].

Wlodzimierz Bielecki et al have chosen OpenMP as a tool to parallelize AES algorithm. At first, they have determined data dependencies in loops using Petit program [50]. They have used AES algorithm with ECB mode of operation. Their strategy was to find most time consuming parts of the algorithm, minimizing data dependencies and building parallel loops. They have executed tests with different plaintexts with sizes are in the range of 1KB and 20MB [39]. They were able to observe almost 5x speedup for encryption and 12x speedup for decryption.

Keisuke Iwai et al have used as optimized AES C code which is a part of OpenSSL toolkit. AES algorithm with ECB mode of operation is used for this research and several approaches have been discussed. The approach was dividing plaintext to 16-bytes blocks and each block being computed by a thread. Another approach is making every thread to compute 1 byte of plaintext. In this approach, a total of 16 threads is required to compute a block. Results of the paper shows that, experiments were able to achieve a maximum of 28.39x speedup when compared to a i7 2.66GHz CPU [4].

Julian Ortega et al have made small changes in the code of AES algorithm which enables file reading/writing operations. They have used a fixed 256 KB chunk size, which limits application to work with file sizes multiples of 256 KB. OpenMP implementation is computing 16-bytes blocks parallelly. On the other hand, CUDA uses 256 KB chunks. Each chunk is copied to memory and divided into blocks and threads. They have observed 1.33x speedup with OpenMP implementation and 4x speedup with CUDA implementation [40].

Tomaiga Radu Daniel et al have used AES algorithm with CTR mode of operation and chosen Nvidia 8800 GT to execute their algorithm. Their approach was to optimizing access time to lookup tables. To do so, they have divided 128-bits sized aes blocks to threads and loaded whole data on the GPU's memory. They have executed algorithm 1.000.000 times in order to measure the performance. Results show that they were able to 1.3x speedup AES implementation [5].

Andrea Di Biagio et al have focused on paralleling of internal operations of AES algorithm with CTR mode of operation. They have used 4 threads to compute each block. They have tested their work on different platforms and as a result, for 32 KB data, they have achieved a maximum 2917 Mbps throughput with a Nvidia and 531 Mbps throughput with an Intel Premium D540. Results shows that they were successful to achieve almost 5.5x speedup [17].

Svetlin A. Manavski has developed two different approaches to execute AES on GPU. First approach was to use an OpenGL based implementation and second approach was to use CUDA. On CUDA approach, plaintext has been divided into 1024 bytes chunks and each CUDA blocks and threads were responsible to process these chunks. He has tested his approaches on an OpenSSL based AES algorithm and he has measured 19.6x speedup with CUDA compared to CPU [14].

Keisuke Iwai et al have used 128-bit AES algorithm with ECB mode of operation. They have improved their previous work with overlapping data transfer and kernel execution. The main idea was to execute encryption of several blocks on GPU while transferring rest of the data to GPU. They used 256 MB size plaintext and were able to achieve 22.5 Gbps throughput with overlapping and 13.4 Gbps throughput without overlapping. Their work shows that overlapping boosts performance up to 1.6x [2].

We have used OpenMPI to parallelize AES algorithm with CTR mode of operation in the paper of Parallelism of AES Algorithm via MPI. PlainText have been separated to 16-bytes blocks and divided blocks to different nodes. Each node computes its blocks and most of the blocks computed parallelly. We have observed 4.5x speedup with 12 nodes compared to one node [37].

1.2.4 CONTEXT OF THESIS

AES Algorithm with CTR Mode of operation has been implemented in this work. For implementation, several different approaches have been used. Implementation methodology, test environment and test data have been discussed in **Chapter II**. Results and suggestion have been discussed in **Chapter III**.

CHAPTER II

METHODOLOGY

In this study, 128-bit AES algorithm (from original submission) has been used. Fist of all algorithm have been modified to be able to read and write data file from HDD. Then, CTR mode is applied on the algorithm and test vectors in [44] is used to ensure of the application of CTR mode. The main function of the application with CTR mode can be seen in Appendix B.1. CTR mode is using a counter value for every block and every block is being encrypted individually. To encrypt a block, corresponding counter value should be merged with initialization vector (by summing, XORing etc) then, initialization vector should be encrypted. To find ciphertext each block should be XORed with the encrypted initialization vector and then merged together to generate ciphertext. Normally, all this work is done in a serial manner, but other blocks do not need to calculate counter value for previous blocks. It is possible to compute every block parallelly.

For this work, it is decided to use several different parallelization methods and in the conclusion section results are compared. AES algorithm is parallelized on two different clusters with OpenMPI implementation, on two different Nvidia GPUs with CUDA implementation, on three different multi-core CPUs with OpenMPI implementation and finally on three different PCs with multi-threaded implementation.

In this work, data is read from HDD and then it is divided to 16-byte AES blocks. Each block is divided to nodes/CPUs/threads and their counter value is calculated using thread/node/CPU id. Because of the limited number of resources (nodes/CPUs/GPU Cores), AES blocks are divided as equally as possible. If it is not possible to divide blocks equally, leap block is being left to be computed by the last node/CPU/thread. For example, if application reads a data with a size of 116 bytes, first of all, it will try to divide data into blocks. To do so, it will simply divide 116 bytes to 16 bytes to find number of 16 bytes-sized blocks. The calculation $116/16 = 7.25$ shows that it is not possible to divide data to equally balanced blocks. So, application will round up the value and it will assume that there are 8 blocks. As it can be understood that only $\frac{1}{4}$ of the last block is filled. So, application will act like rest of the block ($\frac{3}{4}$ of the block) is filled with zeros. Now, it is easy for application to divide blocks if the number of nodes/CPUs/GPUs are power of two (1, 2, 4, 8). But, if there are 3 nodes/CPUs/threads, application will try to divide 8 to 3, since it is not dividable, the result will be $8/3 = 2.6666667$ and application will round down the result to 2 and also it will round it up to 3. This means that every node/CPU/thread should compute 2 blocks, but in order to compute the rest of the blocks, last node/CPU/thread should compute 3 blocks.

This is the basic principle behind dividing blocks. The aim is to compute each block on a different resource. Since resources are limited, blocks are divided equally as possible. The implementation may differ from method to method and it will be detailed for every method in their own section.

2.1 TEST ENVIRONMENT

There are different test environments deployed for this work. Each test environment is deployed using same machines, so, results are comparable. Specifications of the test machines are described below;

- One of Çankaya University's clusters, which will be named as PC Cluster from now on is used. Tasks are divided into 12 nodes, specifications of the nodes are as follows;
  - Every node has RedHat Linux installed on them
  - Five of twelve nodes have 4 CPU cores (2.4 GHz) and 8 GB memory for each
  - Six of twelve nodes have 4 CPU cores (2.8 GHz) and 16 GB memory for each
  - The last nodes have 2 CPU cores (2.9 GHz) and 2 GB memory
  - Each node has openMPI version 1.4.3
- An ordinary PC, which will be named as I7PC from now on is used. It has following configuration;

  - Intel i7 - 4 CPU cores (3.4 GHz) with hyper-threading 8 in total
  - 8 GB memory
  - Ubuntu Desktop Linux 12.04 installed
  - openMPI version 1.5.4
  - CUDA Toolkit Release 5.5

- Two ordinary PCs, which will be named as I5PCs from now on are used. They have following configuration;
  - Intel(R) Core(TM) i3 - 2 CPU cores (3.33 GHz) with hyper-threading 4 in total
  - 4 GB memory
  - Ubuntu Server Linux 12.04 installed
  - openMPI version 1.5.4
  - CUDA Toolkit Release 5.5

- One Research PC, which will be named RSPC from now on is used. It has following configuration;
  - Intel(R) Xeon(R) 16 CPU cores (2.00 GHz) with hyper-threading 32 in total
  - 125 GB memory
  - RedHat Linux installed
  - openMPI version 1.4.3

2.2 DATA

Several different test data have been used for this work. Used data is randomly generated files with following sizes; 50MB, 150MB and 250MB. Each file has been read from disk, encrypted and written back to the disk 100 times. In this way, it was possible to compute mean time of the encryption procedure. Also, there is a problem with CUDA for files bigger than 250 MB. Problem will be detailed in Section 2.4.3. On the other hand, in clusters, only head node has the test data and it divides data to the rest of the nodes through network.

2.3 SERIAL IMPLEMENTATION OF AES

Serial implementation is the basic algorithm. It reads data from file and divides it into 16-bytes sized AES blocks. After dividing into blocks, it encrypts every block one-by-one and then writes them to output file. Encryption works as follows; Application reads a data file from HDD as 16-bytes sized blocks. For every block there is an incrementing counter value starting from 1 (counter value for block one is 1, block two is 2 and so on...). Each counter value is being XORed with initialization vector, then, initialization vector is being encrypted with the key. Finally, encrypted initialization vector is being XORed with the corresponding block and the result is being written back to HDD as output.

This is the very basic implementation. Application reads each block and computes them sequentially. After computation of every block, application halts and the written output file is the encrypted equivalent of the input data.

2.4 PARALLEL IMPLEMENTATION OF AES

In order to parallelize AES algorithm, there are three different parallelization approaches in this work. Each approach has been tested on every suitable test environment. These approaches are;

- Multi-threaded Implementation of AES
- Parallel Implementation of AES using OpenMPI
- Parallel Implementation of AES Using CUDA

Each approach will be detailed in following sections.

2.4.1 MULTI-THREADED IMPLEMENTATION OF AES

In theory, a slightly better implementation than serial one should be multi-threaded implementation. Again, data is read from a file and divided into 16-bytes sized blocks. Then, application executes several threads and separates blocks to these these threads. Threads encrypt blocks parallelly but application writes blocks to the output file in a serial manner. Writing output serially would create a bottleneck but running parallel threads might reduce some time on powerful CPUs. Yet, this method is highly dependent on operating systems scheduling algorithm. Scheduling algorithm can assign each thread to a different CPU or it can assign all threads to the same CPU. Scheduling algorithm's behavior will affect the results [35].

2.4.2 MPI

MPI Implementation is a parallel approach to encrypt data and an example of embarrassingly parallel applications. MPI implementation is based on a head node/CPU which divides the job to all other nodes/CPUs. Thanks to MPI standard, MPI implementation can be applied on a cluster or on a multi-core computer without any change on the code itself [28]. In this work, MPI algorithm has been tested on both clusters and multi-core CPUs. Performances have been compared and mentioned in the Section 3.1.3.

If the application is running on a cluster, head node divides jobs to nodes of cluster, otherwise jobs are being divided to other CPU cores. After the start of the application, it initializes MPI immediately. Head node (or core, depending on the environment) reads data from HDD and sending it through network to all other nodes. Each node (excluding head node) is receiving the data through network and storing it on the local memory (RAM). Application uses MPI_Bcast method rather than MPI_Send. By this way, it only sends the data once through network and reduces unnecessary network latency. After receiving of the data, nodes are (for head node after sending it) calculates its own start value for counter (it is hard-coded as 1 for head node) and end value of the counter. Calculation of the counter value has been described in the beginning of this Chapter.

Nodes read the data from their local memory as 16-bytes sized blocks, then, they encrypt and store them back on local memory. Every node selects data blocks according to current calculated counter values. Every node has whole data but each of them computes different blocks. To do so, blocks are being located on the data buffer, as counter value being used as coordinate pointer. Encryption of blocks are done sequentially by nodes. Nodes (except last node) encrypt blocks only if they are in the calculated counter range. Last node encrypts blocks, which are starting from calculated counter start value to the end of file size. The reason is that, last node is the responsible one of the fraction in the block calculation, which has been described earlier. Nodes which finish encryption of the blocks, immediately send the encrypted results back to the head node. If, the head node didn't finish its job, then, sent data waits to be received by the head node. When the head node finishes its job, it starts writing its own encryption result to the output file and then starts receiving of the sent data by other nodes in a sequential order (node 1 comes first, nodes 2 second and so on). Head node appends every received data to the end of the output file and halts the application. Output of the application is below;

```
I7PC$ mpirun -np 4 crypt data250
Time spent in seconds: 24
```

The MPI implementation can be found in Appendix C.1.

2.4.3 CUDA

CUDA implementation is also parallel another approach to encrypt the data and it is an example of coarse-grained parallelism. Unlike MPI standard, CUDA only lets developer to run application on selected GPUs (Which are Nvidia products, for other brands OpenCL is a good alternative). CUDA is a framework to use GPU for computing purposes in a parallel manner. CUDA implementation is based on blocks and threads within kernels (these blocks are different than AES blocks).

User should be aware of the size of data and how to divide application to kernels, blocks and threads in order to use CUDA implementation. Application starts with asking five questions, which are;

```
 PATH to data file
 # of Kernels
 # of Blocks in X coordinate
 # of Blocks in Y coordinate
 # of Threads in X coordinate
 # of Threads in Y coordinate
```

First of all, application requires number of kernels, as an input, to be opened. These numbers are dependent to the size of data and the limits of the device. Limits can be seen on table 2.1. Both of the GPU cards can run at most 8 blocks per kernel and 1024 threads for each block parallelly. These limitation can only be suppressed by increasing number of kernels. First of all, the application reads whole data to RAM at once and starts KeyExpansion procedure on CPU. The main reason is that result of the KeyExpansion procedure is only calculated once and it is same for every node. There is no need to consume valuable GPU computing power with this procedure over and over again. This procedure may look like a bottleneck, but actually it is enhancing performance by saving a valuable resource. The application then checks the values which have been entered, multiplication of the values shouldn't be smalled then the data size. If the numbers are smaller or too big from the size of the data, then application will raise an error and quit. If there is no error, application calculates the size of the data to be copied on GPU and copies the first part of data to GPUs global memory. Shared memory didn't used in this work because, threads doesn't require to use same data. Before execution of kernels, application calculates the initializing counter value for that kernel. For example, the value will be 1 for the first kernel.

During kernel, each thread calculates the block's counter value according to initial kernel counter value with using thread id x/y, block id x/y, grid dimensions and block dimensions. In order to calculate counter values, position of blocks and threads are assumed they are in a matrix. At first, thread calculates position of the block on the grid and multiplies its value with the total number of threads in a block. With that, it finds initial value for thread with thread id X=0 and Y=0 of that block. Then, it calculates position of thread in the block, adds this calculation to the previously calculated value. In the end, it has raw value of the counter, after adding initial counter value to sum, finally, thread finds its counter value.

Calculated counter value is being used by threads to find the data block in global memory and use as counter value for CTR mode encryption. Each thread only calculates one block of data and after encryption of the data block, it has been copied back to global memory. Kernel terminates its existence after each thread has been halted. Host copies encrypted data from global memory back to RAM and calculates counter value for the next kernel and then triggers it. After execution of every kernel, host flushes the encrypted data from RAM to HDD as output.

Kernels can run sequentially or concurrently with using streams in CUDA. Streams also allows developer to copy some data to the device while executing a kernel. This increases parallelism and speed.Kernel implementation can be found on Appendix D.1.

Property	**GTX 480**	**GTX 650**
CUDA Driver Version / Runtime Version	5.5 / 5.5	5.5 / 5.5
CUDA Capability Major/Minor version number	2.0	3.0
Total amount of global memory	1536 MBytes (1610153984 bytes)	2047 MBytes (2146762752 bytes)
CUDA Cores/MP	480 CUDA Cores	384 CUDA Cores
GPU Clock rate	1401 MHz (1.40 GHz)	1189 MHz (1.19 GHz)
Memory Clock rate	1848 Mhz	2800 Mhz
Memory Bus Width	384-bit	128-bit
L2 Cache Size	786432 bytes	262144 bytes
Total amount of constant memory	65536 bytes	65536 bytes
Total amount of shared memory per block	49152 bytes	49152 bytes
Warp size	32	32
Maximum number of threads per multiprocessor	1536	2048
Maximum number of threads per block	1024	1024
Max dimension size of a thread block (x,y,z)	(1024, 1024, 64)	(1024, 1024, 64)
Max dimension size of a grid size (x,y,z)	(65535, 65535, 65535)	(2147483647, 65535, 65535)

Table 2.1: Specifications of GPUs

Based on output of deviceQuery application in CUDA Toolkit

During CUDA implementation, some interesting problems have been encountered. First of all, it have been observed that application responded differently on different GPUs. It has been proven that the configuration of I5PCs and I7PC are same. Kernel versions, library versions and driver versions have been checked. All of them had the same release version. Application have been compiled on both test environments and also it has been compiled on just one environment and binary file copied to another. Application again showed different behavior. Results were acceptable on I7PC but unacceptable on I5PCs. Only solution found was to compile application with debugging symbols on I5PCs. When it was compiled with debugging symbols, it was giving acceptable results. Times should be slower than no-debugging symbol compilation. So, results in Section 3.1.4 have been normalized. For normalization, GTX650 have been executed with both debugging and no-debugging symbols, then percentage of the time difference have been computed and applied to the results of GTX480 with multiplication of a constant value 0.93. Value has been chosen randomly according to experiences in order to reduce the difference of cache sizes and bus speeds.

Another problem which have been encountered was large data encryption. Problem was occurring for files bigger than 250MB. I've tried to run profiler on the application but the output was saying that application returning non-zero error code. Next thing I've done was to check error code. Its value was clearly zero as it can be seen below;

```
cudaadmin@cudalab2:~$ nvprof -s ./main
==23055== NVPROF is profiling process 23055, command: ./main
Maximum block numbers: 65535 65535 65535
Please enter the path of file: mega
Please enter kernel call count: 2400
Please enter number of X blocks per kernel: 2
Please enter number of Y blocks per kernel: 4
Please enter number of X threads per block: 32
Please enter number of Y threads per block: 32
Internal profiler error (12884901897:999)
==23055== Profiling application: ./main
==23055== Profiling result:
Time(%)      Time     Calls       Avg       Min       Max  Name
100.00%  211.09us         3  70.363us  1.1520us  208.15us
[CUDA memcpy HtoD]
======== Error: Application returned non-zero code 1
cudaadmin@cudalab2:~$ ./main
Maximum block numbers: 65535 65535 65535
Please enter the path of file: mega
Please enter kernel call count: 2400
Please enter number of X blocks per kernel: 2
Please enter number of Y blocks per kernel: 4
Please enter number of X threads per block: 32
Please enter number of Y threads per block: 32
Time spent in seconds: 5
cudaadmin@cudalab2:~$ echo $?
0
```

I was unable to solve this problem, so, I have decided to encrypt files with specific sizes with many iterations. This way, it is possible to measure performance for every approach equally with bigger data sizes.

CHAPTER III

CONCLUSION

3.1 RESULTS

During this work, every approach has been tested on every capable machine with a total of 48 different tests. Results and related discussions about results can be found for any approach on its own sections below.

3.1.1 SERIAL

Serial implementation of AES algorithm is the slowest one and mostly dependent on the performance of CPU. This performance can be affected by any other running processes, because, OS has to schedule between processes and at the single time, only one job can be processed by CPU.

Test Environment	One Node of PC Cluster			I5PC			I7PC			RSPC		
Data Size	50MB	150MB	250MB	50MB	150MB	250MB	50MB	150MB	250MB	50MB	150MB	250MB
Mean Time (in seconds)	37.55	108.91	182.94	27.69	84.48	138.85	16.83	50.44	86.13	64.22	195.93	332.89
Average of the Results												
Data Size	50MB				150MB				250MB			
Average Time (in seconds)	36.56				109.94				182.20			

Table 3.1: Serial Implementation Results

Serial implementation of AES algorithm requires long time to be processed. As it can be seen on Table 3.1, best time could only be achieved with a new generation, high frequency CPU (16.83 seconds for 50MB, 50.44 seconds for 150MB and 86.14 seconds for 250MB) and the worst time could only be achieved with a low frequency CPU (64.13 seconds for 50MB, 195.193 seconds for 150MB, 332.89 seconds for 250MB). Performance comparison of the machines can be seen on Figure 3.1. Highest bars are showing worst performance.

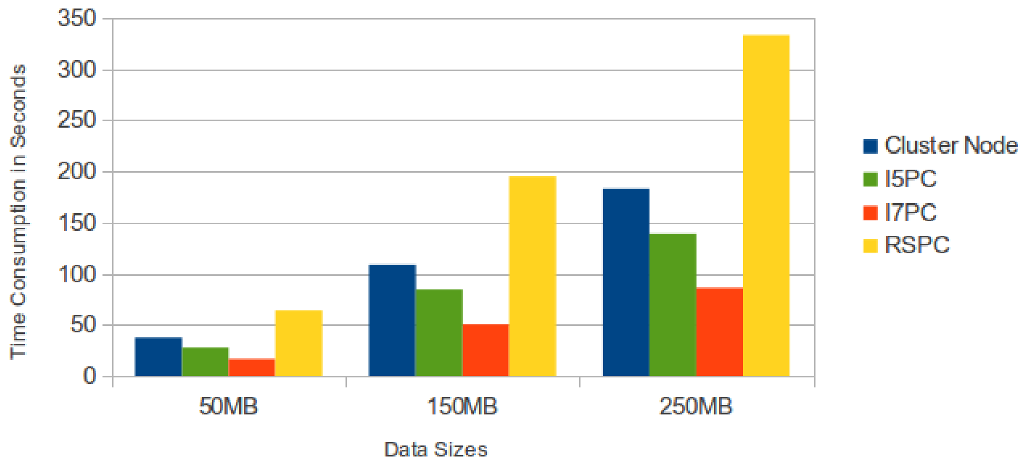

Figure 3.1: Serial Implementation Performance Comparison

3.1.2 MULTI-THREADED

Multi-threaded implementation of AES algorithm is the very basic way to parallelize it. Algorithm shows potential of implementation, yet as its described in Section 2.3, it is highly dependent on operating system's scheduling algorithm and it will show no difference on a single-core CPU. Results can be seen on Table 3.2.

Test Environment	One Node of PC Cluster			I5PC			I7PC			RSPC		
Data Size	50MB	150MB	250MB	50MB	150MB	250MB	50MB	150MB	250MB	50MB	150MB	250MB
Number of Threads	4			2			4			16		
Mean Time (in seconds)	9.71	29.07	48.69	14.26	42.88	72.29	4.80	14.43	24.11	4.47	13.72	21.88
Average of the Results												
Data Size	50MB				150MB				250MB			
Average Time (in seconds)	8.31				25.02				41.74			

Table 3.2: Multi-Threaded Implementation Results

By results, it can be understand that it is possible to have equivalent performance of MPI implementations with a higher frequency, new generation CPU, if taking advantage of every CPU core is being considered while determining thread count.

3.1.3 MPI

MPI Implementation has the potential of increasing performance since its nature is taking advantage of every possible CPU core. Results show that increasing number of CPUs slightly decreases the time requirement of the algorithm. For different data sizes and different environment several tests have been executed and the results compared in the tables below;

Test Environment	Cluster	I5PC	I7PC	RSPC
Number of CPUs	12	2	4	16
Mean Time (in seconds)	2.44	14.44	4.82	1.80
Average Time (in seconds)	5.87			

Table 3.3: MPI Implementation Results for Data Size 50MB

Test Environment	Cluster	I5PC	I7PC	RSPC
Number of CPUs	12	2	4	16
Mean Time (in seconds)	6.97	24.56	14.38	6.49
Average Time (in seconds)	13.1			

Table 3.4: MPI Implementation Results for Data Size 150MB

Test Environment	Cluster	I5PC	I7PC	RSPC
Number of CPUs	12	2	4	16
Mean Time (in seconds)	13.29	41.17	23.99	10.77
Average Time (in seconds)	22.30			

Table 3.5: MPI Implementation Results for Data Size 250MB

It can be seen on the results that large and cheap cluster can easily compete with the new generation CPUs. It is possible to obtain better performance values if many CPUs brought together to build a cluster, even if the CPUs are cheap and nasty. It can be said that beowulf clusters have very good potential in computation, they can be used to increase performance of AES algorithm. Performance comparison of the machines can be seen on Figure 3.2. Highest bars are showing the worst performances.

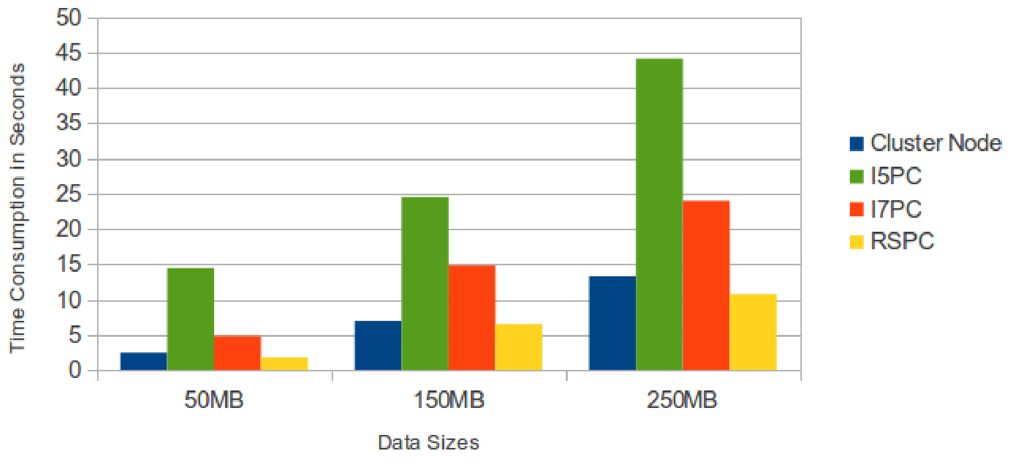

Figure 3.2: MPI Implementation Performance Comparison

3.1.4 CUDA

CUDA is a brand-new and increasing technology. GPUs are very young in the field of computation, but they have already proved themselves with their image processing abilities.

Test results of this work show that with a higher GPU clock rate and more cores it is possible to decrease required time of the application. Still, both GPUs, which are used for tests, are limited with the same thread and block counts. Results of tests can be seen in tables below;

Test Environment	GTX480		GTX650	
Data Size	50MB		50MB	
Mean Time (seconds)	1.27		1.35	
Kernel Count	400		400	
Block Count X&Y	2	4	2	4
Thread Count X&Y	32	32	32	32
Average	1.31			

Table 3.6: CUDA Implementation Results for 50MB-Sized Data

Test Environment	GTX480		GTX650	
Data Size	150MB		150MB	
Mean Time (seconds)	3.88		4.07	
Kernel Count	1200		1200	
Block Count X&Y	2	4	2	4
Thread Count X&Y	32	32	32	32
Average	3.97			

Table 3.7: CUDA Implementation Results for 150MB-Sized Data

Test Environment	GTX480		GTX650	
Data Size	250MB		250MB	
Mean Time (seconds)	6.36		6.82	
Kernel Count	2000		2000	
Block Count X&Y	2	4	2	4
Thread Count X&Y	32	32	32	32
Average	6.59			

Table 3.8: CUDA Implementation Results for 250MB-Sized Data

3.2 COMPRESSION OF RESULTS

Test results point out that all of the methods show different level of improvements. Serial implementation requires the most time as expected. In the meaning of parallelism, if we compare each method with each other, multi-threaded implementation is a good start to increase performance, yet, as it mentioned before, it is highly dependent on operating systems scheduling algorithm. So, it may not give the best results always. MPI is also another good approach for multi-core computers. Both MPI and Multi-Threaded implementations are giving similar results and the increase in performances are almost 78% for Multi-Threaded and 84% for MPI solution. Performance change can be seen on Figure 3.3

Also when compared MPI approaches between each other, it is easily seen that core number has a very important factor on the results. This was also expected, but, if results compared between RSPC and the cluster, RSPC has 16 cores running and cluster using 12 nodes, clusters performance is in a very close range to RSPCs especially for data size 150MB. So if we increase the number of nodes in cluster, it

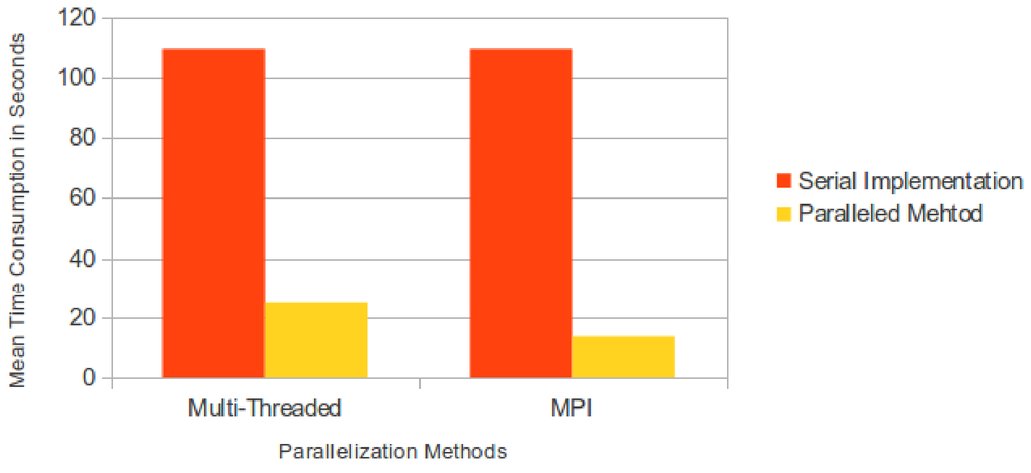

Figure 3.3: Performance Change of Multi-Threaded and MPI Implementations

is possible get better results than RSPC, but network connection between nodes would create a bottleneck. So, if it is possible to obtain a multi-core computer with many cores, it should have better performance than a cluster. Yet, costs of buying a multi-core computer would be higher.

If GPU compared with the other approaches, test results show that parallelization on a GPU has the best performance values. Performance change of every approach can be seen on Figure 3.4. Also, when compared GPU with MPI, it shows 11% better performance for data size 150MB. CUDA is new in the field of parallelization, but tests show that it really shows a very good performance in the field. The speedup values are 27.64 for 250MB, 27.69 for 150MB and 27.90 for 50MB and the average speedup value for GPU is 27.74.

Speedup values have been calculated with the following formula;

$$S_p = \frac{T_s}{T_p} \tag{3.1}$$

In the formula, $\mathrm{T_p}$ is the time spend of parallel algorithm, $\mathrm{T_s}$ is the time spend of serial algorithm and $\mathrm{T_p}$ is the speedup value. Figure 3.5 shows speedup differences between approaches. In the figure, speedup values are based on average of the speedup values of all three data sizes.

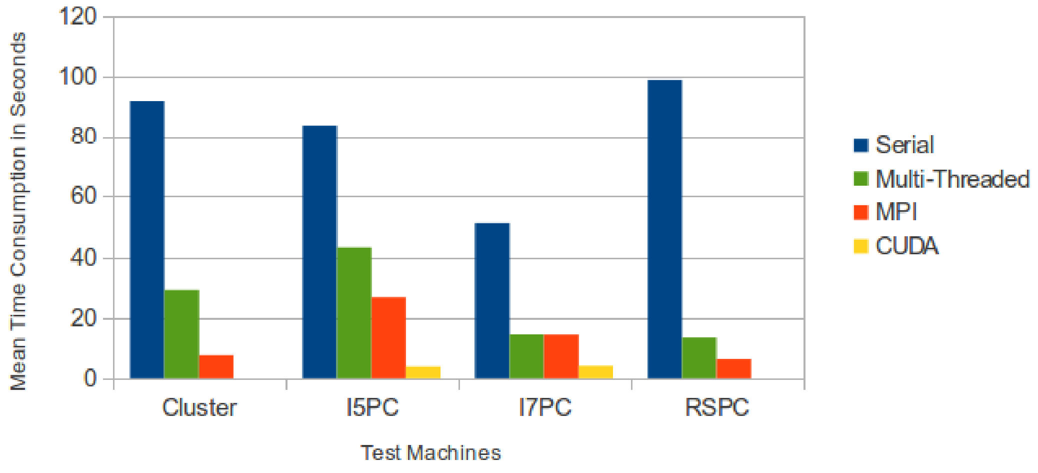

Figure 3.4: Performance Change throughout Test Machines

This work shows that GPU is a good choice to amplify performance of AES algorithm. It is easy to build-up a beowulf cluster using second hand cheap computers. Many nodes might give great performance but increasing number of nodes won't increase speedup linearly. In parallel computing, speedup increases exponentially growth but at some peak point its growth will start slowing exponentially according to Amdahl's law[48]. So, buying a lot of nodes for a cluster won't be efficient. Also today, end users can easily afford to buy a brand new GPU. Prices are in an acceptable range and GPUs show better performance than clusters. If the aim is to increase performance of an every day application's encryption (like web browsers etc), GPU is a good choice, but, if the purpose of encryption is much more important (such as military documents) and the data has very big sizes, buying a supercomputer could be an alternative option.

3.3 SUGGESTIONS FOR FUTURE WORK

In order to increase performance, instead of Rijndael algorithm in Appendix A.1, an optimized version should be used for parallelization. Also, optimization can be done using assembly commands.

Stream library should be added to CUDA implementation, parallelly running kernels should also increase performance.

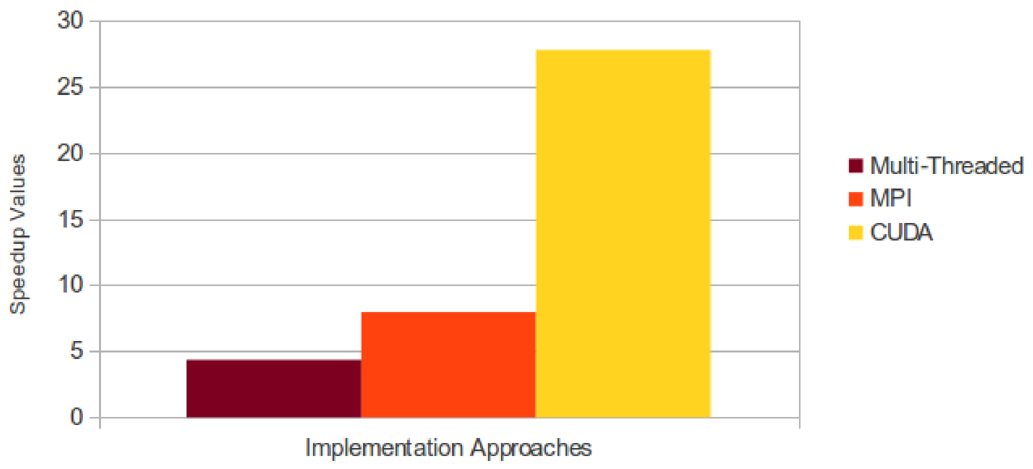

Figure 3.5: Speedup Change Throughout Implementation Approaches

Multiple graphics cards with SLI technology should be used and multiple kernels should be run on different cards parallelly [33]. After successful implementation, stream should also be implemented, then, algorithm can take full advantage of hardware.

CUDA aware MPI library should also be tested. CUDA aware MPI library allows developer to use CUDA abilities with MPI extensions. Through this library, it is possible to build-up a GPU powered cluster. Developer might use both power of GPUs and CPUs of every node in cluster. Implementation of CUDA aware MPI might possibly increase performance of AES algorithm [32].

REFERENCES

[1] **M.ARMBRUST, A.FOX, R.GRIFFITH, A.D.JOSEPH, R.H.KATZ, A.KONWINSKI, G.LEE, D.A.PATTERSON, A.RABKIN, I.STOICA, M.ZAHARIA**, (2009), *Above the Clouds: A Berkeley View of Cloud Computing*, Technical Report No. UCB/EECS-2009-28, University of California, Berkeley

[2] **K.IWAI, N.NISHIKAWA, T.KURUKAWA**, (2012), *Acceleration of AES Encryption on CUDA GPU*, International Journal of Networking and Computing, Vol.2 No.1, 131-145

[3] Retrieved from *http://www.arcos.inf.uc3m.es/˜ii_ac2_en/dokuwiki/lib/exe/f-etch.php?id=slides&cache=cache&media=lecture2_cuda_spring_2010.ppt* on 2013-08-27

[4] **K.IWAI, T.KURUKAWA, N.NISHIKAWA**, (2010), *AES Encryption Implementation on CUDA GPU and Its Analysis*, First International Conference on Networking and Computing, Hangzhou

[5] **T.R.DANIEL, S.MIRCEA**, (2010), *AES on GPU Using CUDA*, European Conference for the Applied Mathematics and Informatics, Minnesota

[6] Retrieved from *http://www.mcs.anl.gov/research/projects/mpi/* on 2013-08-13

[7] Retrieved from *http://www.cs.bc.edu/˜straubin/cs381-05/blockciphers/rijndael_ingles2004.swf* on 2013-02-13

[8] Retrieved from *http://boinc.berkeley.edu/* on 2013-02-17

[9] **H.LIPMAA, P.ROGAWAY, D.WAGNER**, (2000), *Comments to NIST concerning AES Modes of Operations: CTR-Mode Encryption* Symmetric Key Block Cipher Modes of Operation Workshop, Baltimore, Maryland

[10] Retrieved from *https://www.coursera.org/* on 2013-07-23

[11] **H.FEISTEL**, (May 1973), *Cryptography and Computer Privacy*, Scientific American Vol.288, Number 5

[12] Retrieved from *http://cryptozine.blogspot.com/2008/05/brief-history-of-cryptography.html* on 2013-07-19

[13] **J.SANDERS, E.KANDROT**, (2011), *CUDA by Example*, Addison-Wesley, Michigan

[14] **S.A.MANAVSKI**, (2007), *CUDA Compatible GPU as an Efficient Hardware Accelerator for AES Cryptography*, International Conference on Signal Processing and Communications, Dubai

[15] **S.COOK**, (2013), *CUDA Programming a Developer's Guide to Parallel Computing with GPUs*, Elsevier Inc., Waltham

[16] Retrieved from *http://www.daviddfriedman.com/Libertarian/Why_Crypto_Matters.html* on 2013-07-22

[17] **A.D.BIAGIO, A.BARENGHI, G.AGOSTA, G.PELOSI**, (2009), *Design of a Parallel AES for Graphics Hardware Using the CUDA Framework*, Parallel & Distributed Processing, Rome

[18] Retrieved from *http://www.phy.duke.edu/~rgb/Beowulf/beowulf_book/beowulf_book* on 2013-08-17

[19] **NIST** (2001), *Federal Information - Processing Standards Publication 197*, National Institute of Standards and Technology

[20] Retrieved from *http://research.google.com/pubs/DistributedSystemsandParallelComputing.html* on 2013-05-12

[21] **A.J.MENEZES, P.C.V.OORSCHOT, S.A.VANSTONE**, (2001), *Handbook of Applied Cryptography*, CRC Press, Florida

[22] **M.KIPPER, J.SLAVKIN, D.DENISENKO**, (2009), *Implementing AES on GPU Final Report*, University of Toronto, Toronto

[23] **P.PACHECO**, (2011), *Introduction to Parallel Programming* Elsevier Inc., Burlingron

[24] Retrieved from *http://www.khronos.org/opencl/* on 2013-03-22

[25] Retrieved from *https://computing.llnl.gov/tutorials/parallel_comp/* on 2013-08-02

[26] **W.DIFFIE, M.E.HELLMAN**, (1976), *Multiuser Cryptographic Techniques*, Stanford University, National Computer Conference

[27] **Message Passing Interface Forum** (2009), *MPI: A Message-Passing Interface Standard*, University of Tennessee, Tennessee

[28] **M.SNIR, S.OTTO, S.HUSS-LEDERMAN, D.WALKER, J.DONGARRA**, (1996), *MPI: The Complete Reference*, MIT Press, London

[29] **Nvidia CUDA C Programming Guide Version 5.5**, (2013), Nvidia, California

[30] **Nvidia OpenCL Programming Guide for the CUDA Architecture Version 2.3**, (2009), Nvidia, California

[31] Retrieved from *https://developer.nvidia.com/what-cuda* on 2013-01-29

[32] Retrieved from *https://developer.nvidia.com/content/introduction-cuda-aware-mpi* on 2013-08-27

[33] Retrieved from *http://www.nvidia.com/object/sli-campaign.html* on 2013-01-17

[34] Retrieved from *http://cs.nyu.edu/courses/spring12/CSCI-GA.3033-012/lecture6.pdf* on 2013-08-30

[35] **A.SILBERSCHATZ, P.B.GALVIN, G.GAGNE**, (2010), *Operating System Concepts*, John Wiley & Sons, Jefferson City

[36] **White Paper, OpenCL^{TM} : The Future of Accelerated Application Performance Is Now** (2011), AMD, California

[37] **Ö.PEKÇAĞLIYAN**, **N.SARAN** (2013), *Parallelism of AES Algorithm via MPI*, 6th MTS Seminar, Ankara

[38] **D.LE, J.CHANG, X.GOU, A.ZHANG, C.LU**, (2010), *Parallel AES Algorithm for Fast Data Encryption on GPU*, 2nd International Conference on Computer Engineering and Technology, Chengdu

[39] **W.BIRLECKI, D.BURAK**, (2005), *Parallelization of the AES Algorithm*, 4th WSEAS International Conference on Information Security, Canary Islands

[40] **J.ORTEGA, H.TREFFTZ, C.TREFFTZ**, (2011), *Parallelizing AES on Multi-cores and GPUs*, IEEE International Conference on Electro/Information Technology (EIT), Minnesota

[41] **U.K.PRODHAN, A.H.M.S.PARVEZ, Md I.HUSSAIN, Y.F.RUMI, Md A.HOSSAIN** (2012), *Performance Analysis Of Parallel Implementation Of Advanced Encryption Standard (Aes) Over Serial Implementation* IJITS Vol.2, No.6, November 2012

[42] Retrieved from *http://www.philzimmermann.com/* on 2013-08-20

[43] Retrieved from *http://raviranjankr.wordpress.com/* on 2013-08-22

[44] **M.DWORKIN**, (2001), *Recommendation for Block Cipher Modes of Operation Methods and Techniques*, NIST Special Publication 800-38A 2001 Edition

[45] **D.KAHN**, (1973) *The Codebreakers*, Sphere; New edition

[46] **J.DAEMEN, V.RIJMEN**(2002), *The Design of Rijndael: AES - The Advanced Encryption Standard* Springer, Germany.

[47] **P.J.DENNING, J.B.DENNIS**, (2010), *The Profession of IT The Resurgence of Parallelism*, Communications of the ACM, June 2010 Vol.53 No.6, 30-32

[48] Retrieved from *http://siber.cankaya.edu.tr/ozdogan/* on 2013-03-12

[49] Retrieved from *http://www.top500.org* on 2013-09-02

[50] Retrieved from *http://www.cs.umd.edu/projects/omega/* on 2013-08-30

[51] Retrieved from *http://www.volunteer-computing.org/EN/volunteer-computing-in-30-sec.html* on 2013-08-12

[52] Retrieved from *http://ei.cs.vt.edu/ history/Parallel.html* on 2013-07-20

[53] **Y.CHEN, V.PAXSON, R.H.KATZ**, (2010), *What's New About Cloud Computing Security?*, Technical Report No. UCB/EECS-2010-5, University of California, Berkeley

[54] Retrieved from *http://www.wikipedia.org* on 2013-09-04

[55] Retrieved from *http://www.wisegeek.org/what-is-a-supercomputer.html* on 2013-08-13

APPENDIX A

RIJNDAEL ALGORITHM

Listing A.1: Rijndael Algorithm

```
//  Rijndael code August 01
//
//  author:Vincent Rijmen
//  This code is based on the official reference code
//  by Paulo Barreto and Vincent Rijmen
//
//  This code is placed in the public domain.
//  Without any warranty of fitness for any purpose
//=====================================================================
#include<stdio.h>

#include<stdlib.h>
using namespace std;
//=====================================================================
typedef unsigned char word8;
typedef unsigned int word32;
//=====================================================================
//  The table Logtable and Alogtable are used to perform
//  multiplication in GF(256)
//=====================================================================
word8 Logtable[256] = {
    0,   0,   25,  1,  50,  2,  26,  198,
    75,  199,  27,  104,  51,  238,  223,  3,
    100, 4,  224, 14,  52,  141, 129, 239,
    76,  113, 8,   200, 248,  105,  28,  193,
    125, 194, 29, 181, 249, 185, 39, 106,
    77, 228, 166, 114, 154, 201, 9, 120,
    101, 47, 138, 5, 33, 15, 225, 36,
    18, 240, 130, 69, 53, 147, 218, 142,
    150, 143, 219, 189, 54, 208, 206, 148,
    19, 92, 210, 241, 64, 70, 131, 56,
    102, 221, 253, 48, 191, 6, 139, 98,
    179, 37, 226, 152, 34, 136, 145, 16,
    126, 110, 72, 195, 163, 182, 30, 66,
    58, 107, 40, 84, 250, 133, 61, 186,
    43, 121, 10, 21, 155, 159, 94, 202,
    78, 212, 172, 229, 243, 115, 167, 87,
    175, 88, 168, 80, 244, 234, 214, 116,
    79, 174, 233, 213, 231, 230, 173, 232,
    44, 215, 117, 122, 235, 22, 11, 245,
    89, 203, 95, 176, 156, 169, 81, 160,
    127, 12, 246, 111, 23, 196, 73, 236,
    216, 67, 31, 45, 164, 118, 123, 183,
    204,  187, 62, 90, 251, 96, 177, 134,
    59, 82, 161, 108, 170, 85, 41, 157,
    151, 178, 135, 144, 97, 190, 220, 252,
    188, 149, 207, 205, 55, 63, 91, 209,
    83, 57, 132, 60, 65, 162, 109, 71,
    20, 42, 158, 93, 86, 242, 211, 171,
    68, 17, 146, 217, 35, 32, 46, 137,
    180, 124, 184, 38, 119, 153, 227, 165,
    103, 74, 237, 222, 197, 49, 254, 24,
    13, 99, 140, 128, 192, 247, 112, 7
};
//=====================================================================
word8   Alogtable[256] = {
    1,  3, 5, 15, 17, 51, 85, 255,
    26, 46, 114, 150, 161, 248, 19, 53,
    95, 225, 56, 72, 216, 115, 149, 164,
    247, 2, 6, 10, 30, 34, 102, 170,
    229, 52, 92, 228, 55, 89, 235, 38,
    106, 190, 217, 112, 144, 171, 230, 49,
    83, 245, 4, 12, 20, 60, 68, 204,
    79, 209, 104, 184, 211, 110, 178, 205,
    76, 212, 103, 169, 224, 59, 77, 215,
    98, 166, 241, 8, 24, 40, 120, 136,
    131, 158, 185, 208, 107, 189, 220, 127,
    129, 152, 179, 206, 73, 219, 118, 154,
    181, 196, 87, 249, 16, 48, 80, 240,
    11, 29, 39, 105, 187, 214, 97, 163,
    254, 25, 43, 125, 135, 146, 173, 236,
    47, 113, 147, 174, 233, 32, 96, 160,
    251, 22, 58, 78, 210, 109, 183, 194,
    93, 231, 50, 86, 250, 21, 63, 65,
    195, 94, 226, 61, 71, 201, 64, 192,
    91, 237, 44, 116, 156, 191, 218, 117,
    159, 186, 213, 100, 172, 239, 42, 126,
    130, 157, 188, 223, 122, 142, 137, 128,
    155, 182, 193, 88, 232, 35, 101, 175,
    234, 37, 111, 177, 200, 67, 197, 84,
    252, 31, 33, 99, 165, 244, 7, 9,
    27, 45, 119, 153, 176, 203, 70, 202,
    69, 207, 74, 222, 121, 139, 134, 145,
    168, 227, 62, 66, 198, 81, 243, 14,
    18, 54, 90, 238, 41, 123, 141, 140,
    143, 138, 133, 148, 167, 242, 13, 23,
    57, 75, 221, 124, 132, 151, 162, 253,
    28, 36, 108, 180, 199, 82, 246, 1
};
//=====================================================================
word8 S[256] = {
    99, 124, 119, 123, 242, 107, 111, 197,
    48, 1, 103, 43, 254, 215, 171, 118,
    202, 130, 201, 125, 250, 89, 71, 240,
    173, 212, 162, 175, 156, 164, 114, 192,
    183, 253, 147, 38, 54, 63, 247, 204,
    52, 165, 229, 241, 113, 216, 49, 21,
    4, 199, 35, 195, 24, 150, 5, 154,
    7, 18, 128, 226, 235, 39, 178, 117,
    9, 131, 44, 26, 27, 110, 90, 160,
    82, 59, 214, 179, 41, 227, 47, 132,
    83, 209, 0, 237, 32, 252, 177, 91,
    106, 203, 190, 57, 74, 76, 88, 207,
    208, 239, 170, 251, 67, 77, 51, 133,
    69, 249, 2, 127, 80, 60, 159, 168,
    81, 163, 64, 143, 146, 157, 56, 245,
    188, 182, 218, 33, 16, 255, 243, 210,
    205, 12, 19, 236, 95, 151, 68, 23,
    196, 167, 126, 61, 100, 93, 25, 115,
    96, 129, 79, 220, 34, 42, 144, 136,
    70, 238, 184, 20, 222, 94, 11, 219,
    224, 50, 58, 10, 73, 6, 36, 92,
    194, 211, 172, 98, 145, 149, 228, 121,
    231, 200, 55, 109, 141, 213, 78, 169,
    108, 86, 244, 234, 101, 122, 174, 8,
    186, 120, 37, 46, 28, 166, 180, 198,
    232, 221, 116, 31, 75, 189, 139, 138,
    112, 62, 181, 102, 72, 3, 246, 14,
    97, 53, 87, 185, 134, 193, 29, 158,
    225, 248, 152, 17, 105, 217, 142, 148,
    155, 30, 135, 233, 206, 85, 40, 223,
    140, 161, 137, 13, 191, 230, 66, 104,
    65, 153, 45, 15, 176, 84, 187, 22
};
//=====================================================================
word8 Si[256] = {
    82, 9, 106, 213, 48, 54, 165, 56,
    191, 64, 163, 158, 129, 243, 215, 251,
    124, 227, 57, 130, 155, 47, 255, 135,
    52, 142, 67, 68, 196, 222, 233, 203,
    84, 123, 148, 50, 166, 194, 35, 61,
    238, 76, 149, 11, 66, 250, 195, 78,
    8, 46, 161, 102, 40, 217, 36, 178,
    118, 91, 162, 73, 109, 139, 209, 37,
    114, 248, 246, 100, 134, 104, 152, 22,
    212, 164, 92, 204, 93, 101, 182, 146,
    108, 112, 72, 80, 253, 237, 185, 218,
    94, 21, 70, 87, 167, 141, 157, 132,
    144, 216, 171, 0, 140, 188, 211, 10,
    247, 228, 88, 5, 184, 179, 69, 6,
    208, 44, 30, 143, 202, 63, 15, 2,
    193, 175, 189, 3, 1, 19, 138, 107,
    58, 145, 17, 65, 79, 103, 220, 234,
    151, 242, 207, 206, 240, 180, 230, 115,
    150, 172, 116, 34, 231, 173, 53, 133,
    226, 249, 55, 232, 28, 117, 223, 110,
    71, 241, 26, 113, 29, 41, 197, 137,
    111, 183, 98, 14, 170, 24, 190, 27,
    252, 86, 62, 75, 198, 210, 121, 32,
    154, 219, 192, 254, 120, 205, 90, 244,
    31, 221, 168, 51, 136, 7, 199, 49,
    177, 18, 16, 89, 39, 128, 236, 95,
    96, 81, 127, 169, 25, 181, 74, 13,
    45, 229, 122, 159, 147, 201, 156, 239,
    160, 224, 59, 77, 174, 42, 245, 176,
    200, 235, 187, 60, 131, 83, 153, 97,
    23, 43, 4, 126, 186, 119, 214, 38,
    225, 105, 20, 99, 85, 33, 12, 125
};
//=================================================================
word32 Rc[30] = {
    0x00, 0x01, 0x02, 0x04, 0x08, 0x10, 0x20, 0x40, 0x80, 0x1B,
    0x36, 0x6C, 0xD8, 0xAB, 0x4D, 0x9A, 0x2F, 0x5E, 0xBC, 0x63,
    0xC6, 0x97, 0x35, 0x6A, 0xD4, 0xB3, 0x7D, 0xFA, 0xEF, 0xC5
};
//=================================================================
#define MAXBC       8
#define MAXKC       8
#define MAXROUNDS   14
//=================================================================
static word8 shifts[5][4] = {
    0,1,2,3,
    0,1,2,3,
    0,1,2,3,
    0,1,2,4,
    0,1,3,4};
//=================================================================
static int numrounds[5][5] = {
    10,11,12,13,14,
    11,11,12,13,14,
    12,12,12,13,14,
    13,13,13,13,14,
    14,14,14,14,14};
//=================================================================
int     BC;
int     KC;
int     ROUNDS;
//=================================================================
//   multiply two elements of GF(256)
//   required for MixColumns and InvMicolumns
//=================================================================
word8 mul(word8 a, word8 b)
{

    if( a&&b )

        return Alogtable[(Logtable[a] + Logtable[b]) % 255];

    else

        return 0;

}
//=================================================================
// XOR corresponding text input and round key input bytes
//=================================================================
void AddRoundKey(word8 a[4][MAXBC],word8 rk[4][MAXBC])
{
    int i;
    int j;
    for(i=0; i<4; i++)
    {
        for(j=0; j<BC; j++)
        {
            a[i][j] ^= rk[i][j];
        }
    }
}
//===================================================================
// Replace every byte of the input by the byte at that place
// in the non-linear S-box
//===================================================================
void SubBytes( word8 a[4][MAXBC], word8 box[256] )
{
    int i;
    int j;

    for(i=0; i<4; i++)
    {
        for(j=0; j<BC; j++)
        {
            a[i][j] = box[a[i][j]];
        }
    }
}
//===================================================================
//  Row 0 remains unchanged
//  The other three rows are shifted a variadle amount
//===================================================================
void ShiftRows(word8 a[4][MAXBC], word8 d)
{
    word8   tmp[MAXBC];
    int i;
    int j;


    if( d==0 )
    {
        for( i=1; i<4; i++ )
        {
            for( j=0; j<BC; j++ )
            {

                tmp[j] = a[i][( j + shifts[BC-4][i] ) % BC];
            }
            for(j=0; j<BC; j++)
            {
                a[i][j] = tmp[j];
            }
        }
    }
    else
    {
        for( i=1; i<4; i++ )
        {
            for( j=0; j<BC; j++ )
            {

                tmp[j]=a[i][( BC + j - shifts[BC-4][i] ) % BC];
            }

            for(j=0; j<BC; j++)
            {
                a[i][j] = tmp[j];
            }
        }
    }
}
//===================================================================
//  Mix the four bytes of every column in a linear way
//===================================================================
void MixColumns(word8 a[4][MAXBC])
{
    word8 b[4][MAXBC];
    int i;
    int j;
    for(j=0; j<BC; j++)
    {
        for(i=0; i<4; i++)
        {
            b[i][j] = mul(2,a[i][j])^mul(3,a[(i+1)%4][j])^a[(i+2)%4][j]^a[(i+3)%4][j];
        }
    }

    for(i=0; i<4; i++)
    {
        for(j=0; j<BC; j++)
        {
            a[i][j] = b[i][j];
        }
    }
}
//=====================================================================
//  Mix the four bytes of every column in a linear way
//  This is the opposite operation of Mixcolumns
//=====================================================================
void InvMixcolumns( word8 a[4][MAXBC] )
{
    word8 b[4][MAXBC];
    int i;
    int j;


    for(j=0; j<BC; j++)
    {
        for(i=0; i<4; i++)
        {
            b[i][j] = mul(0xe, a[i][j]) ^ mul(0xb, a[(i + 1) % 4 ][j])
                ^ mul(0xd, a[(i + 2) % 4 ][j]) ^ mul(0x9, a[(i + 3) % 4 ][j]);
        }
    }
    for( i=0; i<4; i++ )
    {
        for( j=0; j<BC; j++ )
        {
            a[i][j] = b[i][j];
        }
    }
}
//=====================================================================
int KeyExpansion( word8 k[4][MAXKC],word8 w[MAXROUNDS+1][4][MAXBC] )
{
//  Calculate the required round keys
    int i;
    int j;
    int t;
    int RCpointer = 1;
    word8 tk[4][MAXKC];
    for( j=0; j<KC; j++ )
    {
        for( i=0; i<4; i++ )
        {
            tk[i][j] = k[i][j];
        }
    }
    t=0;
    //  copy values into round key array
    for( j=0; (j < KC) && (t < (ROUNDS + 1) * BC ); j++, t++ )
    {
        for( i=0; i<4; i++ )
        {
            w[t / BC][i][t % BC] = tk[i][j];
        }
    }
    while( t<(ROUNDS+1)*BC )
    {
```

```
    //  while not enough round key material calculated,
    //  calculate new values
        for( i=0; i<4; i++ )
        {
            tk[i][0] ^= S[tk[(i+1)%4][KC-1] ];
        }
        tk[0][0] ^= Rc[ RCpointer++ ];
        if( KC <= 6 )
        {
            for( j=1; j<KC; j++ )
            {
                for( i=0; i<4; i++ )
                {
                    tk[i][j] ^= tk[i][j-1];
                }
            }
        }
        else
        {
            for( j=1; j<4 ;j++ )
            {
                for( i=0; i<4; i++ )
                {
                    tk[i][j] ^=tk[i][j-1];
                }
            }
            for( i=0; i<4; i++ )
            {
                tk[i][4] ^= S[ tk[i][3] ];
            }
            for( j=5; j<KC; j++ )
            {
                for( i=0; i<4; i++ )
                {
                    tk[i][j] ^= tk[i][j-1];
                }
            }
        }
    //  copy values into round key array
        for( j=0; (j<KC) && (t<(ROUNDS+1)*BC); j++,t++ )
        {
            for( i=0; i<4; i++ )
            {
                w[t/BC][i][t%BC] = tk[i][j];
            }
        }
    }
    return 0;
}

int Encrypt( word8 a[4][MAXBC],word8 rk[MAXROUNDS+1][4][MAXBC] )
{
    int r;
    AddRoundKey(a,rk[0]);
    for( r=1; r<ROUNDS; r++ )
    {
        SubBytes(a,S);
        ShiftRows(a,0);
        MixColumns(a);
        AddRoundKey(a,rk[r]);
    }
    //  last round is special:there is no MixColumns
    SubBytes(a,S);
    ShiftRows(a,0);
    AddRoundKey(a,rk[ROUNDS]);

    return 0;
}

int Decrypt(word8 a[4][MAXBC], word8 rk[MAXROUNDS+1][4][MAXBC] )
{
    int r;
    AddRoundKey(a,rk[ROUNDS]);
    SubBytes(a,Si);
    ShiftRows(a,1);
    for( r=ROUNDS-1; r>0; r-- )
    {
        AddRoundKey(a,rk[r]);
        InvMixcolumns(a);
        SubBytes(a,Si);
        ShiftRows(a,1);
    }
    AddRoundKey(a,rk[0]);
    return 0;
}

int main()
{
    int i,mmm;
    int j;
    word8 a[4][MAXBC];
    word8 rk[MAXROUNDS+1][4][MAXBC];
    word8 sk[4][MAXKC];

    for( KC=4; KC<=8; KC++ )
    {
        for( BC=4; BC<=8; BC++ )
        {
            {
                ROUNDS = numrounds[KC-4][BC-4];
            }
            for( j=0; j<BC; j++ )
            {
                for( i=0; i<4; i++ )
                {
                    a[i][j] = 0;
                }
            }
            for( j=0; j<KC; j++ )
            {
                for( i=0; i<4; i++ )
                {
                    sk[i][j] = 0;
                }
            }
            KeyExpansion(sk,rk);
            Encrypt(a,rk);
            printf("block length %d key length %d\n", 32*BC, 32*KC );
            for( j=0; j<BC; j++ )
            {
                for( i=0; i<4; i++ )
                {
                    printf("%02X", a[i][j] );
                }
            }
            printf("\n");
            Decrypt(a,rk);
            for( j=0; j<BC; j++ )
            {
                for( i=0; i<4; i++ )
                {
                    printf("%02X", a[i][j]);
                }
            }
            printf("\n");
            printf("\n");

            for( j=0; j<BC; j++ )
            {
                for( i=0; i<4; i++ )
                {
                    a[i][j] = i;
                }
            }
            Encrypt(a,rk);
            for( j=0; j<BC; j++ )
            {
                for( i=0; i<4; i++ )
                {
                    printf("%02X", a[i][j]);
                }
            }
            printf("\n");
            Decrypt(a,rk);
            for( j=0; j<BC; j++ )
            {
                for( i=0; i<4; i++ )
                {
                    printf("%02X", a[i][j]);
                }
            }
            printf("\n");
            printf("\n");
        }
    }
    //      system("pause");
    return 0;
}
```

APPENDIX B

CTR MODE

Listing B.1: CTR Mode

```
int main()
{
        int i;
        int j;
        unsigned char rk[MAXROUNDS+1][4][MAXBC];
        unsigned char sk[4][MAXKC] = {{0x2b,0x28,0xab,0x09},{0x7e,0xae,0xf7,0xcf},{0x15,0
            xd2,0x15,0x4f},{0x16,0xa6,0x88,0x3c}};
//CASE 1 Encrypt
//      unsigned char a[4][MAXBC] = {{0x6b,0x2e,0xe9,0x73},{0xc1,0x40,0x3d,0x93},{0xbe,0
    x9f,0x7e,0x17},{0xe2,0x96,0x11,0x2a}};
//      unsigned char iv[4][MAXBC] = {{0xf0,0xf4,0xf8,0xfc},{0xf1,0xf5,0xf9,0xfd},{0xf2,0
    xf6,0xfa,0xfe},{0xf3,0xf7,0xfb,0xff}};
//      unsigned char iv2[4][MAXBC] = {{0xf0,0xf4,0xf8,0xfc},{0xf1,0xf5,0xf9,0xfd},{0xf2,0
    xf6,0xfa,0xfe},{0xf3,0xf7,0xfb,0xff}};
//CASE 2
//      unsigned char a[4][MAXBC] = {{0xae,0x2d,0x8a,0x57},{0x1e,0x03,0xac,0x9c},{0x9e,0
    xb7,0x6f,0xac},{0x45,0xaf,0x8e,0x51}};
//      unsigned char iv[4][MAXBC] = {{0xf0,0xf4,0xf8,0xfc},{0xf1,0xf5,0xf9,0xfd},{0xf2,0
    xf6,0xfa,0xff},{0xf3,0xf7,0xfb,0x00}};
//      unsigned char iv2[4][MAXBC] = {{0xf0,0xf4,0xf8,0xfc},{0xf1,0xf5,0xf9,0xfd},{0xf2,0
    xf6,0xfa,0xff},{0xf3,0xf7,0xfb,0x00}};
//CASE 3
//      unsigned char a[4][MAXBC] = {{0x30,0xc8,0x1c,0x46},{0xa3,0x5c,0xe4,0x11},{0xe5,0
    xfb,0xc1,0x19},{0x1a,0x0a,0x52,0xef}};
//      unsigned char iv[4][MAXBC] = {{0xf0,0xf4,0xf8,0xfc},{0xf1,0xf5,0xf9,0xfd},{0xf2,0
    xf6,0xfa,0xff},{0xf3,0xf7,0xfb,0x01}};
//      unsigned char iv2[4][MAXBC] = {{0xf0,0xf4,0xf8,0xfc},{0xf1,0xf5,0xf9,0xfd},{0xf2,0
    xf6,0xfa,0xff},{0xf3,0xf7,0xfb,0x01}};
//CASE 4
        unsigned char a[4][MAXBC] = {{0xf6,0x9f,0x24,0x45},{0xdf,0x4f,0x9b,0x17},{0xad,0
            x2b,0x41,0x7b},{0xe6,0x6c,0x37,0x10}};
        unsigned char iv[4][MAXBC] = {{0xf0,0xf4,0xf8,0xfc},{0xf1,0xf5,0xf9,0xfd},{0xf2,0
            xf6,0xfa,0xff},{0xf3,0xf7,0xfb,0x02}};
        unsigned char iv2[4][MAXBC] = {{0xf0,0xf4,0xf8,0xfc},{0xf1,0xf5,0xf9,0xfd},{0xf2,0
            xf6,0xfa,0xff},{0xf3,0xf7,0xfb,0x02}};
        BC = 4, KC = 4;
        ROUNDS = numrounds[KC-4][BC-4];
        KeyExpansion(sk,rk);
        Encrypt(iv, rk);
        printf("block length %d key length %d\n\nEncrypted data: ", 32*BC, 32*KC );

        for( j=0; j<4; j++ )
        {
                for( i=0; i<4; i++ )
                {
                        iv[i][j] ^= a[i][j];
                        printf("%02X", iv[i][j]);
                }
        }

        printf("\n\n\nDecrypted data: ");
        Encrypt(iv2, rk);
        for( j=0; j<4; j++ )
        {
                for( i=0; i<4; i++ )
                {
                        iv[i][j] ^= iv2[i][j];
                        printf("%02X", iv[i][j]);
                }
        }
        printf("\n");
        printf("\n");
        printf("\n");
        return 0;
}
```

APPENDIX C

HEAD NODE PART OF THE MPI IMPLEMENTATION

Listing C.1: Head Node Part of the MPI Implementation

```
void ctrivxor(double ctr, unsigned char iv[4][MAXBC], int i = 0, int j = 0, int p = 0)
{
        if(j != BC)
        {
                double power;
                int res = 0;
                int newctr;
                int newp;
                if(p == 0 && j == (BC - 1) && i == 3)
                {
                        newp = 1;
                        power = 10;
                        res = fmod(ctr, power);
                        iv[i][j] ^= res;
                        i = 0;
                        j++;
                        return;
                }
                if(p == 0)
                {
                        newp = 4 * BC - 1;
                }
                else
                {
                        newp = p;
                }
                power = (int)pow((double)10, newp);
                res = ctr / power;
                iv[i][j] ^= res;
                newctr = fmod(ctr, power);
                i++;
                if(i == 4)
                {
                        i = 0;
                        j++;
                }
                ctrivxor(newctr, iv, i, j, --newp);
        }
}


int main(int argc, char**argv)
{
        int i;
        int j;
        int myid;
        unsigned char rk[MAXROUNDS+1][4][MAXBC];
        unsigned char sk[4][MAXKC] = {{0x2b,0x28,0xab,0x09},{0x7e,0xae,0xf7,0xcf},{0x15,0
             xd2,0x15,0x4f},{0x16,0xa6,0x88,0x3c}};
        unsigned char a[4][MAXBC];
        unsigned char iv[4][MAXBC] = {{0xf0,0xf4,0xf8,0xfc},{0xf1,0xf5,0xf9,0xfd},{0xf2,0
             xf6,0xfa,0xfe},{0xf3,0xf7,0xfb,0xff}};
        BC = 4, KC = 4;
        ROUNDS = numrounds[KC-4][BC-4];
        int stime = time(0);
        KeyExpansion(sk,rk);
        unsigned char iv2[4][MAXBC];
        MPI_Init( &argc, &argv );
        MPI_Comm_size( MPI_COMM_WORLD, &worldSize );
        MPI_Comm_rank(MPI_COMM_WORLD,&myid);


        if(myid == 0)
        {
                int iter = 0;
                while(iter < 100)
                {
                        ofstream out;
                        char tinyBuffer[4*BC];
                        int fsize;
                        char *buffer;
                        //read data and send it to the nodes
                        ifstream in;
                        in.open(argv[1], ios::in);
                        in.seekg(0,ios::end);
                        fsize = in.tellg();
                        in.seekg(0, ios::beg);
                        buffer = (char*)malloc(fsize * sizeof(char));
                        in.read(buffer, fsize);
                        MPI_Bcast((void*)&fsize, 1, MPI_INT, 0, MPI_COMM_WORLD);
                        MPI_Bcast((void*)buffer, fsize, MPI_BYTE, 0, MPI_COMM_WORLD);


                        //Do own calculations
                        int ctr = 1; //There should be a counter interval for every node.
                             This helps to create that interval
                        out.open("testFile1outCrypt", ios::out);
                        int finalctr = ((int)ceil(((double)fsize / (4*BC))) / worldSize) *
                             (myid + 1);
                        while(ctr <= finalctr)
                        {
                                for(j = 0; j < BC; j++)
                                {
                                        for(i = 0; i < 4; i++)
                                        {
                                                iv2[i][j] = iv[i][j];
                                        }
                                }
                                ctrivxor(ctr, iv2);
                                Encrypt(iv2, rk);
                                for(i = 0; i <= 4*BC; i++)
                                {
                                        tinyBuffer[i] = '\0';
                                }
                                j = 0;
                                for(i = 0; i < 4*BC && j < fsize; i++)
                                {
                                        j = ((ctr - 1) * 4*BC) + i;
                                        tinyBuffer[i] = buffer[j];
                                }
                                for(j = 0; j < BC; j++)
                                {
                                        for(i = 0; i < 4; i++)
                                        {
                                                a[i][j] = tinyBuffer[j*BC+i];
                                                iv2[i][j] ^= a[i][j];
                                                tinyBuffer[j*BC+i] = iv2[i][j];
                                        }
                                }
                                out.write(tinyBuffer, 4*BC);
                                ctr++;
                        }
                        free(buffer);
                        //Start receiving from nodes
                        int nfsize;
                        char *recieveBuffer;
                        MPI_Status status;
                        for(i = 1; i < worldSize; i++)
                        {
                                MPI_Recv((void*)&nfsize, 1, MPI_INT, i, 0, MPI_COMM_WORLD,
                                     &status);
                        recieveBuffer = (char*)malloc(sizeof(char) * nfsize);
                        MPI_Recv((void*)recieveBuffer, nfsize, MPI_BYTE, i, 0,
                            MPI_COMM_WORLD, &status);
                        out.write(recieveBuffer, nfsize);
                        free(recieveBuffer);
                }

                //clean the rest
                out.flush();
                out.close();
                iter++;
        }
        int etime = time(0);
        cout<<"Time spent in seconds: "<<etime - stime<<endl;
}
else if (myid != worldSize - 1) //machines other than first and last
{
        int iter = 0;
        while(iter < 100)
        {
                char tinyBuffer[4*BC];
                int fsize;
                char *buffer;

                //receive data from master
                MPI_Bcast((void*)&fsize, 1, MPI_INT, 0, MPI_COMM_WORLD);
                buffer = (char*)malloc(sizeof(char) * fsize);
                MPI_Bcast((void*)buffer, fsize, MPI_BYTE, 0, MPI_COMM_WORLD);

                //Do Own calculations
                int ctr = ((int)ceil(((double)fsize / (4*BC))) / worldSize) * myid
                     + 1; //There should be a counter interval for every node.
                     This helps to create that interval
                int finalctr = ((int)ceil(((double)fsize / (4*BC))) / worldSize) *
                     (myid + 1);
                while(ctr <= finalctr)
                {
                        for(j = 0; j < BC; j++)
                        {
                                for(i = 0; i < 4; i++)
                                {
                                        iv2[i][j] = iv[i][j];
                                }
                        }
                        ctrivxor(ctr, iv2);
                        Encrypt(iv2, rk);
                        for(i = 0; i <= 4*BC; i++)
                        {
                                tinyBuffer[i] = '\0';
                        }
                        j = 0;
                        for(i = 0; i < 4*BC && j < fsize; i++)
                        {
                                j = ((ctr - 1) * 4*BC) + i;
                                tinyBuffer[i] = buffer[j];
                        }
                        for(j = 0; j < BC; j++)
                        {
                                for(i = 0; i < 4; i++)
                                {
                                        a[i][j] = tinyBuffer[j*BC+i];
                                        iv2[i][j] ^= a[i][j];
                                        tinyBuffer[j*BC+i] = iv2[i][j];
                                }
                        }
                        j = 0;
                        for(i = 0; i < 4*BC && j < fsize; i++)
                        {
                                j = ((ctr - 1) * 4*BC) + i;
                                buffer[j] = tinyBuffer[i];
                        }
                        ctr++;
                }
                //Start sending data
                int ectr = ctr;
```

```
            ctr = ((int)ceil(((double)fsize / (4*BC))) / worldSize) * myid +
                 1; //There should be a counter interval for every node. This
                 helps to create that interval
            int nfsize = ((ectr - 1) * (4*BC)) - ((ctr - 1) * (4*BC)); //
                 difference of end seek point and head seek point. This gives
                 total data count
            char *sendBuffer = (char*)malloc(nfsize * sizeof(char));
            j = 0;
            for(i = 0; i < nfsize && j < fsize; i++)
            {
                j = ((ctr - 1) * 4*BC) + i;
                sendBuffer[i] = buffer[j];
            }
            MPI_Send((void*)&nfsize, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
            MPI_Send((void*)(sendBuffer), nfsize, MPI_BYTE, 0, 0,
                 MPI_COMM_WORLD);
            free(buffer);
            free(sendBuffer);
            iter++;
        }
    }
    else //Man, I am the last node in cluster
    {
        int iter = 0;
        while(iter < 100)
        {
            char tinyBuffer[4*BC];
            int fsize;
            char *buffer;
            char *sendBuffer;
            //receive data from master
            MPI_Bcast((void*)&fsize, 1, MPI_INT, 0, MPI_COMM_WORLD);
            buffer = (char*)malloc(sizeof(char) * fsize);
            MPI_Bcast((void*)buffer, fsize, MPI_BYTE, 0, MPI_COMM_WORLD);

            //Do own calculations
            int ictr = ((int)ceil(((double)fsize / (4*BC))) / worldSize) *
                 myid + 1; //There should be a counter interval for every node
                 . This helps to create that interval
            int ctr = ictr;
            int tread = 0;
            int read = (worldSize - 1) * ((int)ceil(((double)fsize / (4*BC)))
                 / worldSize) * (4*BC);
            int nfsize = ((fsize - read) / (4 * BC)) * 4 * BC + ((fsize - read
                 ) % (4 * BC) == 0?0:16);
            sendBuffer = (char*)malloc(nfsize * sizeof(char));
            while (read < fsize)
            {
                for(j = 0; j < BC; j++)
                {
                    for(i = 0; i < 4; i++)
                    {
                        iv2[i][j] = iv[i][j];
                    }
                }
                ctrivxor(ctr, iv2);
                Encrypt(iv2, rk);
                tread = fsize - read;
                for(i = 0; i <= 4*BC; i++)
                {
                    tinyBuffer[i] = '\0';
                }
                if(tread <= 4*BC && tread > 0)
                {
                    for(i = 0; i < tread && j < fsize; i++)
                    {
                        j = ((ctr - 1) * 4*BC) + i;
                        tinyBuffer[i] = buffer[j];
                    }
                }
                else
                {
                    for(i = 0; i < 4*BC && j < fsize; i++)
                    {
                        j = ((ctr - 1) * 4*BC) + i;
```

```
                                        tinyBuffer[i] = buffer[j];
                                }
                        }
                        read += 4*BC;
                        for(j = 0; j < BC; j++)
                        {
                                for(i = 0; i < 4; i++)
                                {
                                        a[i][j] = tinyBuffer[j*BC+i];
                                        iv2[i][j] ^= a[i][j];
                                        tinyBuffer[j*BC+i] = iv2[i][j];
                                }
                        }
                        for(i = 0; i < 4*BC && j < fsize; i++)
                        {
                                j = ((ctr - ictr) * 4 * BC) + i;
                                sendBuffer[j] = tinyBuffer[i];
                        }
                        ctr++;
                }

                //Start sending data
                MPI_Send((void*)&nfsize, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
                MPI_Send((void*)(sendBuffer), nfsize, MPI_BYTE, 0, 0,
                    MPI_COMM_WORLD);
                free(buffer);
                free(sendBuffer);
                iter++;
            }
        }

        MPI_Finalize();
        return 0;
}
```

APPENDIX D

KERNEL FUNCTION

Listing D.1: Kernel Function

```
__global__ void kernelCrypt (float d, char *buffer, int bsize) //d is the initializer of
    the counter
{
        char tinyBuffer[4*BC];
        unsigned char iv2[4][MAXBC];
        unsigned char a[4][MAXBC];
        int i;
        int j;
        float ctr = ((blockIdx.x *  gridDim.y + blockIdx.y) * blockDim.x * blockDim.y) + (
            threadIdx.x * blockDim.y + threadIdx.y) + d; //calculate ctr value
        for(j = 0; j < BC; j++)
        {
                for(i = 0; i < 4; i++)
                {
                        iv2[i][j] = iv[i][j]; //save original IV value
                        tinyBuffer[j*BC+i] = '\0';//clean tinyBuffer before use
                }
        }
        ctrivxor(ctr, iv2); //counter ile ivyi xorla
        Encrypt(iv2, rk); //encrypted new iv value with rk
        int zgrw = (int)((4*BC) * (ctr - d)); //the variable which holds the coordinates
            on the buffer
        if(zgrw < bsize) //if it is bigger or equal it means we are out of bounds and this
             might cause problems
        {
                for(j = 0; j < BC; j++)
                {
                        for(i = 0; i < 4; i++)
                        {
                                tinyBuffer[j*BC+i] = buffer[zgrw + (j*BC+i)];
                                a[i][j]  = tinyBuffer[j*BC+i];
                                iv2[i][j] ^= a[i][j]; //xor iv with Buffer
                                tinyBuffer[j*BC+i] = iv2[i][j]; //Fill buffer with
                                    encrypted data
                                buffer[zgrw + (j*BC+i)] = tinyBuffer[j*BC+i];
                        }
                }
        }
}
```

APPENDIX E

CURRICULUM VITAE

PERSONAL INFORMATION

Surname, Name: Pekçağlıyan, Özgür
Nationality: Turkish (TC)
Date and Place of Birth: 12 December 1985, Ankara
Marital Status: Single
Phone: +90 535 364 22 00
e-mail: ozgur.pekcagliyan@gmail.com

EDUCATION

Degree	**Institution**	**Year of Graduation**
B.Sc.	Çankaya Univ.Computer Engineering	2008
High School	Private Evrensel High School	2003

WORK EXPERIENCE

Year	**Place**	**Enrollment**
2011-Present	Turkish Coast Guard Command	Project Manager
2010	Labris Technologies	System Administrator & C++ Developer
2009	Alcatel-Lucent	C++ Developer
2008-2009	Alcatel-Lucent	System Administrator

FOREIGN LANGUAGES

English - Advanced
German - Beginner

PUBLICATIONS

1. Özgür Pekçağlıyan, Nurdan Saran, *Parallelism of AES Algorithm via MPI*, 6th MTS Seminar, April 2013

2. Özgür Pekçağlıyan, Yusuf Soyman, *Parallel Password Cracking Using Brute Force B.Sc. Thesis Report*, Cankaya University, February 2008

HOBBIES

Movies, Karting, Animes.