Master Thesis
Electrical Engineering
September 2014

Performance analysis of GPGPU and CPU
On AES Encryption

Akash Kiran Neelap

Master Thesis in
Electrical Engineering

School of Electrical Engineering
Blekinge Institute of Technology
SE-371 79 Karlskrona
Sweden

Internet: www.bth.se/ing
Phone: +46 455 38 50 00
Fax: +46 455 38 50 57

This thesis is submitted to the School of Engineering at Blekinge Institute of Technology in partial fulfillment of the requirements for degree of Master of Science in Electrical Engineering with emphasis on Telecommunication Systems.

Contact Information

Author:	Akash Kiran Neelap Email: aks.akashkiran@gmail.com
University Supervisor:	Prof. Lars Lundberg Department of Computer Science and System Engineering Blekinge Institute of Technology Karlskrona, Sweden Contact No: 0045-385833 Email: lars.lundberg@bth.se
Industrial Supervisor:	Ross W Tsagalidis Project Manager, POC Swedish Armed Forces (SWAF) Sweden Contact No: 0046-733 666982 Email: wross@tele2.se
Examiner:	Prof. Kurt Tutschku Department of Communication Systems Blekinge Institute of Technology Karlskrona, Sweden Contact No: 0455-385872 Email: kurt.tutschku@bth.se

School of Electrical Engineering
Blekinge Institute of Technology
SE-371 79 Karlskrona
Sweden

Internet: www.bth.se/ing
Phone: +46 455 38 50 00
Fax: +46 455 38 50 57

ABSTRACT

The advancements in computing have led to tremendous increase in the amount of data being generated every minute, which needs to be stored or transferred maintaining high level of security. The military and armed forces today heavily rely on computers to store huge amount of important and secret data, that holds a big deal for the security of the Nation. The traditional standard AES encryption algorithm being the heart of almost every application today, although gives a high amount of security, is time consuming with the traditional sequential approach.

Implementation of AES on GPUs is an ongoing research since few years, which still is either inefficient or incomplete, and demands for optimizations for better performance. Considering the limitations in previous research works as a research gap, this paper aims to exploit efficient parallelism on the GPU, and on multi-core CPU, to make a fair and reliable comparison. Also it aims to deduce implementation techniques on multi-core CPU and GPU, in order to utilize them for future implementations.

This paper experimentally examines the performance of a CPU and GPGPU in different levels of optimizations using Pthreads, CUDA and CUDA STREAMS. It critically exploits the behaviour of a GPU for different granularity levels and different grid dimensions, to examine the effect on the performance. The results show considerable acceleration in speed on NVIDIA GPU (QuadroK4000), over single-threaded and multi-threaded implementations on CPU (Intel® Xeon® E5-1650).

Keywords: *AES algorithm; CUDA; GPU computing; Pthreads*

ACKNOWLEDGEMENT

I would like to express my deep gratitude to Prof. Lars Lundberg my supervisor without whom this research would not have been possible. He has been an excellent mentor who has been patiently guiding me throughout this research work.

I am grateful to Ross W Tsagalidis (SAF), my external supervisor for his consistent support and encouragement.

I owe particular thanks to Stefan Peterson (BTH), for his invaluable guidance and assistance. I also thank him for providing me necessary equipments to accomplish this thesis work.

Special thanks to Md. Altaf Ahmed and C. Gautam Krishna, my friends for their consistent support.

Finally, I would like to thank my parents for their kind love and support, words cannot convey.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF GRAPHS

LIST OF ABBREVIATIONS

Acronyms	**Description**
AES	Advanced Encryption Standard
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
GPU	Graphics Processing Unit
GPGPU	General Purpose Graphics Processing Unit
MIMD	Multiple Instruction Multiple Data
NIST	National Institute of Standards and Technology
SIMD	Single Instruction Multiple Data
SIMT	Single Instruction Multiple Thread

CHAPTER 1 INTRODUCTION

This chapter describes briefly the motivation behind the research work, aim and objectives, contributions of the research and organization of the paper.

1.1 OVERVIEW

Cryptography has been the hot topic of research in the field of information security since ancient times. Beginning from the ancient Caesar's cipher that used manual encryption of data, followed by enigma machine used in the World War II, after the advancement in mechanics, it has been the urge of researchers to find solutions for better performance in terms of speed and security. With the invention of transistors, which gave rise to microprocessors, began the era of digital automation. This new advancement together with contemporary processors, gave rise to a new generation of encryption technology. Since then, various encryption algorithms have been developed and utilized. Encryption finds its usage in almost every field where data protection is concerned, including scientific institutions, corporate offices, social networking, and more specifically in military and government affairs in order to facilitate secret communication. The military and armed forces hold huge amount of secret information which relates to the protection of the country and its people, making itself one of the most important area to use the encryption technology.

The Advanced Encryption Standard (AES) is a specification for the encryption of electronic data established by the U.S. National Institute of Standards and Technology (NIST) in 2001[25]. This standard today is used worldwide in almost every field including government organizations, military, educational institutions, ATM cards, computer password security and electronic commerce. The present state of art technology generates exponential amounts of confidential data which needs to be encrypted before being stored or transferred to authorized destination. Encryption being heavily based on intense mathematical computations is a time taking process [4]. Fast encryption has been an important subject of research and discussion since many years. Various implementations techniques and optimization methods have been incorporated to achieve better speedups. Yet the answer to "Which method can give the best solution?" remains partially solved [19]. The advancement in the computer industry directly affects the performance of these algorithms. This is a positive hint for researchers, to focus on the improvement in the processor utility, in order to achieve better processing speeds.

Microprocessors based on a single central processing unit (CPU) have given high performance increase since decades. Various techniques have been followed to increase the speed of the processor which includes increase in the clock speed. The culture of increasing the clock speed has flattened out, due to issues such as excessive power consumption, heat dissipation and current leakage. Also, excessive heat raises the need of expensive cooling equipments, increasing the cost of overall system. The single core microprocessors served for decades providing considerable functionality and user interfaces. As the law of science demands, the users in turn long for more improvements and faster computation. The computer industry moved from single core processors to multi-core processors in which more than one processing units or processing cores are used in a single chip. The model of multi-core processor has exerted a tremendous impact on the software community. According to the Moore's law, "the number of transistors in a dense integrated circuit doubles approximately every two years" [37]. In other words, the number of processing core doubles with every new generation of processors. However the increase in the number of cores in a processor does not mark much impact on such systems, as a sequential program can run on only one of the processor cores leaving other cores idle. The introduction of multi threaded programming languages such as POSIX threads, however promises to help utilizing multiple cores of the system effectively in such applications.

The addition of more cores into a single chip demands for higher chip area, which leads to increased cost besides the power consumption and heat dissipation problems, which continues to increase with increasing processing units. Hence, development in the hardware cannot continue to be an effective solution to increase the processing speed of the system. The application software here after can continue to feed on high performance achievements only by the use of parallel programming, i.e., improvement in the software instead of addition in the hardware. However, this theory holds good for specific algorithms and its level of parallelizability. Algorithms are programmed in such a way that its parallel parts can be executed simultaneously or concurrently on different processor cores. The practice of parallel programming is by no means new. Although it has been in practise since quite before, the application area to use this technique had been limited due to the need for expensive systems. Nowadays all micro processors are parallel computers, and it is possible and also necessary to parallelise many applications to accelerate performance. This again demands the need for programmers to learn more about parallel computing, the technique of utilizing cores efficiently to gain maximum performance.

After the introduction of General Purpose Graphical Processing Units (GPGPU), many parallel applications have been shifted to graphic cards, or implemented with coordination with CPU in a heterogeneous environment. The architecture of Graphical Processing Unit (GPU), which has hundreds of independent processing cores, is best suitable for applications that contain excessive parts that are independent of each other, and can be implemented simultaneously. This however requires sufficient knowledge in many-core programming, GPU architecture and memory design. A programmer also needs to understand which part of the program can be best implemented on a GPU and on a CPU. With the introduction of Compute Unified Device Architecture (CUDA) parallel computing platform and programming model created by NVIDIA, various parallel programming languages such as Direct Compute, OpenCL and CUDA have been developed using which, a programmer need not master the depth of the hardware constrains of the system.

1.2 MOTIVATION

Many research works have been performed previously, in order to compare the performance of GPU and CPU, for the implementation of various encryption algorithms including AES. Where few researches claim that GPUs outperform CPUs, the implementation technique used is either improper or incomplete. Also, the performance improvement is quite small or almost negligible, to promise that GPUs can outperform CPUs on every scenario. The devices used are older GPUs with Tesla and Fermi architecture, and not much research has been done on the latest Kepler architecture GPUs which is used in this research work. The GPUs are not efficiently exploited, to be able to deduce the factual standards to be followed in order to get efficient performance, keeping the programmer diffident while implementing a new research using a GPU. The GPUs have been working on the principle of Single Instruction Multiple Data (SIMD), and further parallelising the processing was not possible until the CUDA STREAMS have been recently introduced to

GPU computing. The usage of CUDA STREAMS towards general purpose computing is new and not much research has been done towards its utility.

Where GPUs try to spread its mark on every general purpose application, the developments in the multi-core trajectory continues to increase. As the number of cores in the multi-core processors increase, the scope for faster execution on the CPUs considerably increases opening the doors to new results and inferences. These improvements on multi-core processors pose challenges to hold back general purpose applications to rely on CPUs for better performance, instead of completely switching into GPU computing. Parallel computing on multi-core processors is still an ongoing research and demands for more intricate exploration.

This research aims to experiment the different versions of AES algorithm (AES-128, AES-192 and AES-256) on the state of art CPU and GPU on different levels of optimizations to identify the best implementation technique to achieve fastest execution. It focuses on understanding how effectively multiple threads in a GPU can be utilized to achieve best performance of an algorithm. It evaluates parallel implementation on the CPU using Pthreads, and compares its performance with that of single threaded program. It also aims to differentiate the performance of optimised version of programming on GPU using CUDA STREAMS, in order to understand its effects on the performance of the algorithm. Finally this research proposes protocols, which can be considered while implementing an algorithm on the GPU by future researchers. This research being an active addition to ongoing research on parallel computing, introduces valuable ideas and techniques, which has not been efficiently stated earlier, and proves to be a definite contribution to the world of computing.

1.3 AIM AND OBJECTIVES

The aim of this research is to evaluate and compare the performance of CPU and GPU on the three different versions of AES encryption algorithm, on different levels of parallelism.

The objectives to achieve the aims are:

a) Develop Single threaded C program on the CPU for AES 128, AES 192 and AES 256 algorithms and record the execution time of each.
b) Develop CUDA program on the GPU for AES 128, AES 192 and AES 256 algorithms and record the execution time of each. Examine the performance of each implementation on different levels of granularities to identify the fastest execution.
c) Compare the performance of the CPU and GPU based objective a) and b).
d) Develop Multi threaded C program on the CPU using POSIX thread for AES 128, AES 192 and AES 256 algorithms and record the execution time of each.
e) Compare the performance of the single threaded C program and the multi threaded C program on the CPU and analyse the result.
f) Compare the performance of the multi threaded C program on the CPU and the CUDA program on the GPU and analyse the result.
g) Optimize the CUDA program using CUDA STREAMS on the GPU and analyse the performance compared to the CUDA program.

1.4 RESEARCH QUESTIONS

RQ1:
a) Does the GPU outperform the single-core CPU in the implementation of AES algorithm?
b) Does the GPU outperform the multi-core CPU in the implementation of AES algorithm?

RQ2:
Does the use of CUDA STREAMS have a positive impact on the performance of AES algorithm?

RQ3:
Which implementation gives the fastest AES execution?

1.5 MAIN CONTRIBUTIONS

- This research work is a valuable addition to ongoing research on exploring parallelism on a GPU. It experimentally explains the effects of varied thread-block usage on an algorithm.
- The main focus of this paper is to provide the performance comparison of a CPU and a GPGPU on different levels of parallelism considering all possible programming constraints and combinations to implement cryptographic algorithm.
- It proposes the best method to implement AES algorithm to achieve least execution speed.
- The observations in this research work can be used by future researchers to make better use of parallel programming.

Contribution to Armed Force:

GPUs holds its usage in various areas of the defence and armed forces which includes image and video processing applications like image stabilization, cockpit and commander display, video tracking, digital mapping and radar processing, and data protection applications including encryption, compression etc. Identifying a better way to exploit parallelism in GPU directly supports fast execution of these applications.

AES encryption algorithm is widely used by military and armed force due to its dependable security and popularity. Improvement in the execution speed of AES can help reducing the time to encrypt the continuously increasing huge amount of confidential data.

1.6 THESIS ORGANISATION

The report has been meticulously organized to provide the reader the ease to understand and follow.
Hereafter, the report is organized as follows.

Chapter 2 begins with the description of the AES algorithm. It then reviews the history of parallel computing beginning with single core processors followed by multi-core and many-core/GPU computing followed by an introduction to CUDA programming model. It finally reviews the previous related work done on this area of research. A good understanding of these will help the reader to better comprehend this research work.

Chapter 3 is an important section of the document which describes the methodology used to accomplish this research work. Describing the research approach, it explains the implementation details and specifications.

Chapter 4 presents the results obtained by the experiments conducted during the research. It involves tables and graphs which is later used to analyse and compare the outcome of the implementation.

Chapter 5 discusses the observations and inference through the research and obtained results. It also presents the verification and validation of the implementation and result. It is an important section of the document as it helps to answer the present research questions and proves to be a reference for future research work.

Chapter 6 presents the conclusion which includes the answers to the research questions and future work.

CHAPTER 2 BACKGROUND

This section firstly describes briefly the design of AES algorithm. Secondly it introduces the concept of parallel computing followed by the description of GPU using the CUDA programming model. It then presents the usage of CUDA STREAMS.

2.1 ADVANCED ENCRYPTION STANDARD

The Advanced Encryption Standard (AES) is a symmetric block cipher. A symmetric block cipher uses the same secret key or cipher-key to encrypt and decrypt information. The AES takes as input a sequence of 128-bits and generates an output sequence of 128-bits. This sequence can also be called as blocks. The number of bits inside a block determines its length. AES algorithm can choose a cipher-key of length 128-bits, 192-bits or 256-bits. Depending upon the cipher-key size selected, the AES version is referred to as AES-128, AES-192 or AES-256 respectively. A number of parameters depend on the cipher-key size. The cipher-key size selected for an AES cipher decides the number of repetitions of transformation rounds that would be used for the encryption or decryption of the data. If the cipher-key size is 128-bits then the number of rounds or cycles of repetition in both encryption and decryption is 10, whereas it is 12 rounds and 14 rounds for 192-bits and 256-bits respectively. This section describes the computational rounds in AES explaining the various operations in each round in detail.

AES operates on an input data of 128-bits i.e., 16-bytes. These bytes can be represented as finite field elements in the form of polynomial representation [25]:

$$b_7x^7 + b_6x^6 + b_5x^5 + b_4x^4 + b_3x^3 + b_2x^2 + b_1x + b_0$$

For example: A set of data {10011100} will be represented as

$$x^7 + x^4 + x^3 + x^2$$

This element can be represented in hexadecimal notation as {9c}.

Input/output:

The basic entities that the AES algorithm operates on are the input data (plain-text) and the cipher-key. The 16-byte input data is arranged in a 4×4 column-major order matrix known as the input matrix and is denoted as ‘‘in_matrix’’as shown below.

$$\begin{bmatrix} byte_0 & byte_4 & byte_8 & byte_{12} \\ byte_1 & byte_5 & byte_9 & byte_{13} \\ byte_2 & byte_6 & byte_{10} & byte_{14} \\ byte_3 & byte_7 & byte_{11} & byte_{15} \end{bmatrix}$$

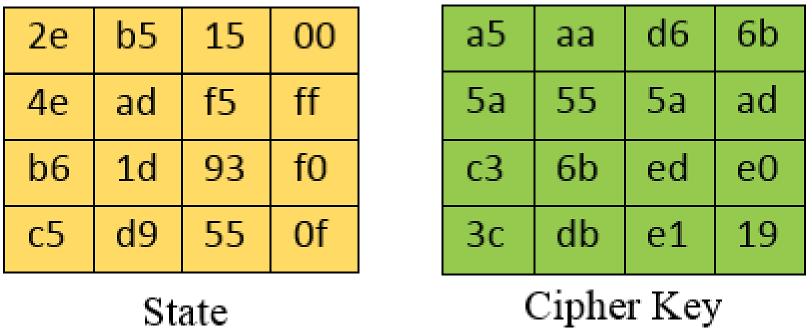
The matrix is organized such that the columns are stored one after the other. The first four bytes of the 128-bit input block occupy the first column in the 4×4 matrix of bytes. The next four bytes occupy the second column, and so on. The same method is used in order to arrange any matrix here forth. This matrix is then copied into another two dimensional array of bytes known as input state which is directly given as input to the AES algorithm. This two dimensional array of bytes is known as the input state, state array or simply state and is denoted as ‘‘in_state’’. The in_state undergoes various transformation rounds and generates subsequent outputs after each operation which is also referred to as state. After the completion of all the rounds, the final output of the algorithm known as the output state denoted as ‘‘out_state’’is copied to a 16-byte 4×4 column-major matrix known as the output matrix denoted as ‘‘out_matrix’’. The data in the output matrix is the required cipher-text.

Rounds:

The AES algorithm has several transformation rounds to transform any information into encoded format and vice versa. AES-128 has 10 rounds; whereas AES-192 and AES-256 has 12 and 14 rounds respectively. Each round consists of several processing steps. Except for the last round all other rounds are identical. A set of reverse rounds are applied to transform cipher-text back into the original plain-text using the same encryption key. The decryption process is out of the scope of this dissertation and hence will not be considered. To understand the working of AES encryption in depth let us consider the case of AES-128.

Algorithm with example

This section provides the detailed working of AES algorithm [39]. Consider an example input matrix and cipher-key matrix as shown below. The input matrix is first copied to the state array.



2e	b5	15	00
4e	ad	f5	ff
b6	1d	93	f0
c5	d9	55	0f

State

a5	aa	d6	6b
5a	55	5a	ad
c3	6b	ed	e0
3c	db	e1	19

Cipher Key

Figure 1: Example state and cipher-key

The AES algorithm can be divided into two main processes.

A. Key expansion/ Key scheduling
B. Encryption/ Decryption

The state matrix is given directly as input to the encryption process, whereas the cipher-key is processed through the key schedule before being used in encryption.

<u>Key Expansion</u>

The key expansion mechanism takes as input the cipher-key and generates a series of round-keys that would be used during the encryption process. The round-keys are derived from the cipher-key by using Rijndael's key schedule. If the total number of words in the cipher-key is ‘‘n_i’’ and the number of rounds in AES algorithm is N, then the key expansion will generate a round-key of a total of $n_i(N + 1)$words. The cipher-key is arranged in the form of a matrix of 4×4byes as shown below. The matrix is arranged such that the first 4 bytes of the key fills the first column of the matrix, the second 4 bytes fills the second column and so on.

$$\begin{bmatrix} k_0 & \mathrm{k}_4 & \mathrm{k}_8 & \mathrm{k}_{12} \\ \mathrm{k}_1 & \mathrm{k}_5 & \mathrm{k}_9 & \mathrm{k}_{13} \\ \mathrm{k}_2 & \mathrm{k}_6 & \mathrm{k}_{10} & \mathrm{k}_{14} \\ \mathrm{k}_3 & \mathrm{k}_7 & \mathrm{k}_{11} & \mathrm{k}_{15} \end{bmatrix}$$

The figure below depicts the arrangement of the cipher-key and the expansion of the key into a key schedule consisting of "44" 4 byte words. Let the expanded key be denoted as ''w'' such that$w = (w_0, w_1, w_2 \ldots w_{43})$.Each round in encryption consumes four words from the key schedule generated.

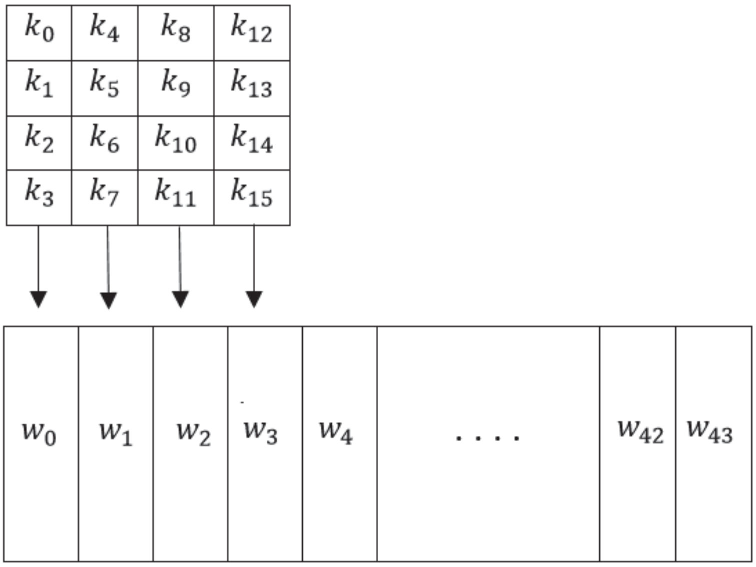
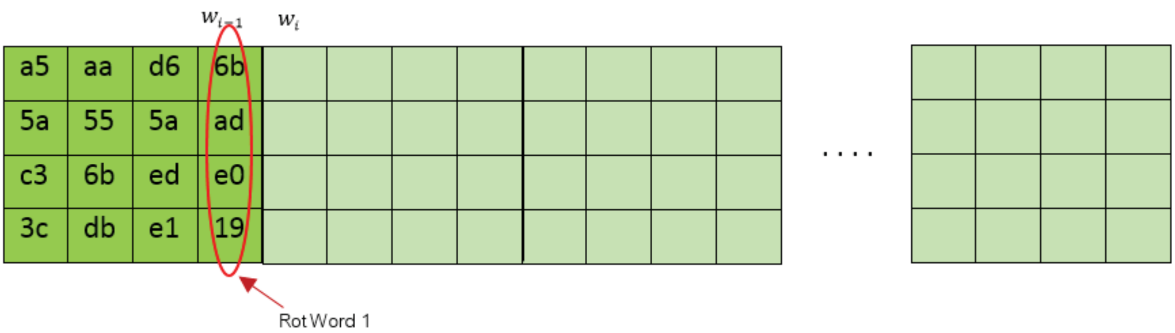


Figure 2: Key scheduling model

The round-key generation from the cipher-key can be clearly understood by considering a random cipher-key as example shown in figure 1. As mentioned earlier, the cipher-key is expanded into a key schedule of 44 words.

The steps involved in generating a round-key matrix are given below.

I. Firstly, the four columns of the cipher-key are copied to the first four columns of the key schedule as shown below.



a5	aa	d6	6b
5a	55	5a	ad
c3	6b	ed	e0
3c	db	e1	19

Figure 3: Rot word

II. The right-most column of the cipher-key circled in the table above denoted as ‘‘w_{i-1}’’ acts as the first ‘‘Rot word’’. Note that here forth the right-most column of every round-key would act as a rot word and contribute in generating the next round-key.

III. The Rot word is rotated upwards such that each byte of the column takes the place of the byte above it, as shown below.

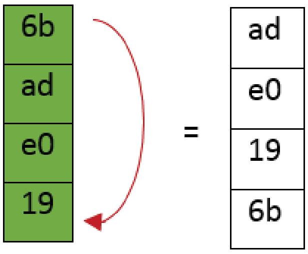


Figure 4: Rot word rotation

IV. Each byte of the obtained column is replaced with its substitute byte in the S-Box. This method is called ‘‘substitute-byte’’ or ‘‘sub-byte’’. For example: The first byte of the column "ad" should be substituted by the element at the a^{th}row and the d^{th}column of the S-Box. Similarly the second byte of the column "4f" should be substituted by the element of the 4^{th}row and f^{th}column of the S-Box, and so on. After the sub-byte operation, the obtained column is shown below.

95
e1
d4
7f

Figure 5: Column after sub-byte

V. The obtained column is then added to the column which is two columns away from the rot-word to the left, denoted as ‘‘w_{i-4}’’. The output of this addition is then added to the first column of the R-con matrix known as R-con(4).

Note that each column of the R-con matrix contributes to the generation of a new round-key. The addition is obtained by the XOR operation and is shown below.

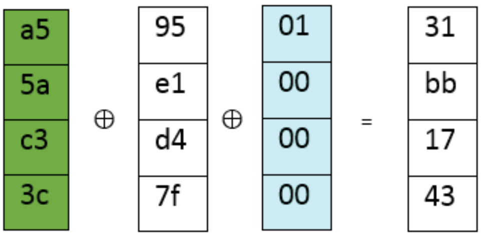


Figure 6: First column generation

The output obtained from the above operation is the next column of the key schedule matrix, and also the first column of the first round-key matrix.

VI. The second, third and the fourth column of the first round key matrix can be obtained by simply performing XOR operation between columns to its left as shown below.

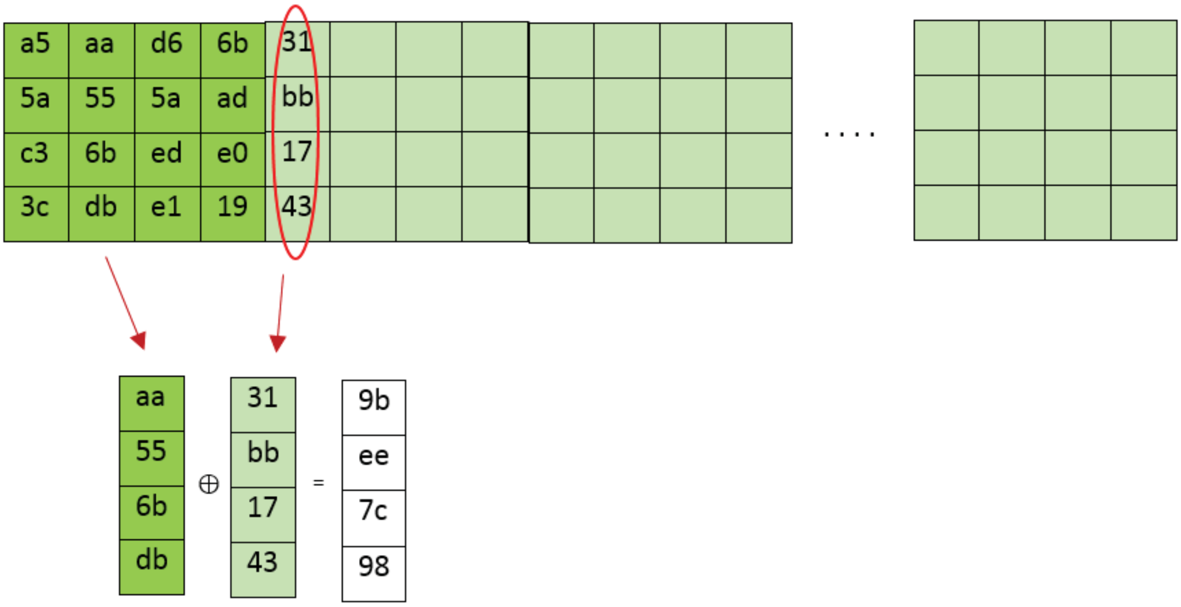
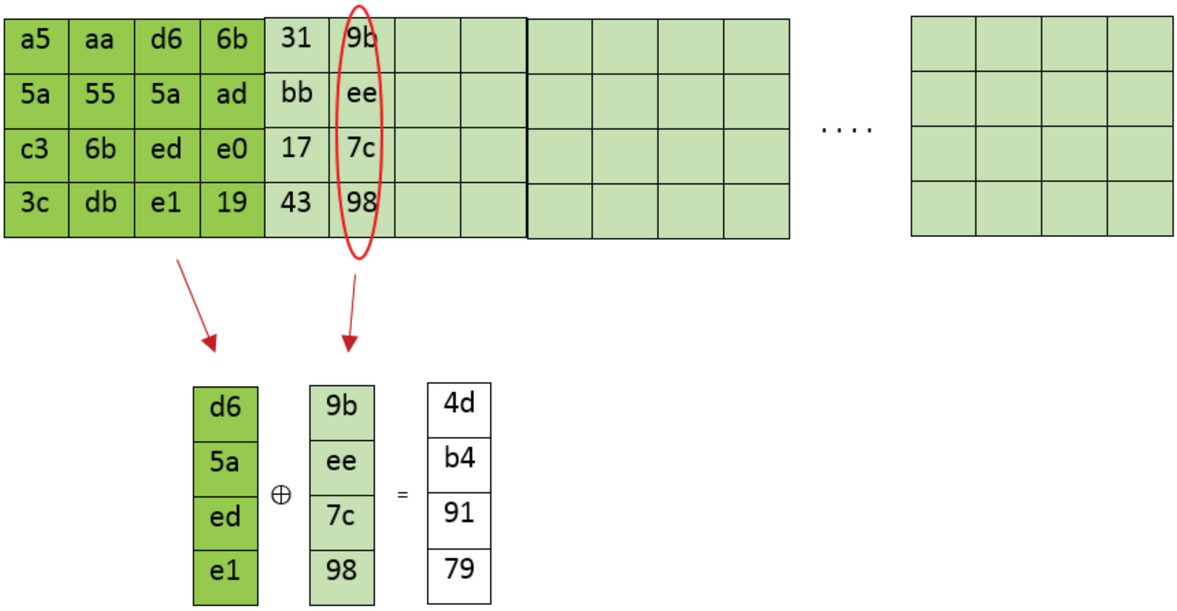


Figure 7: Second column generation

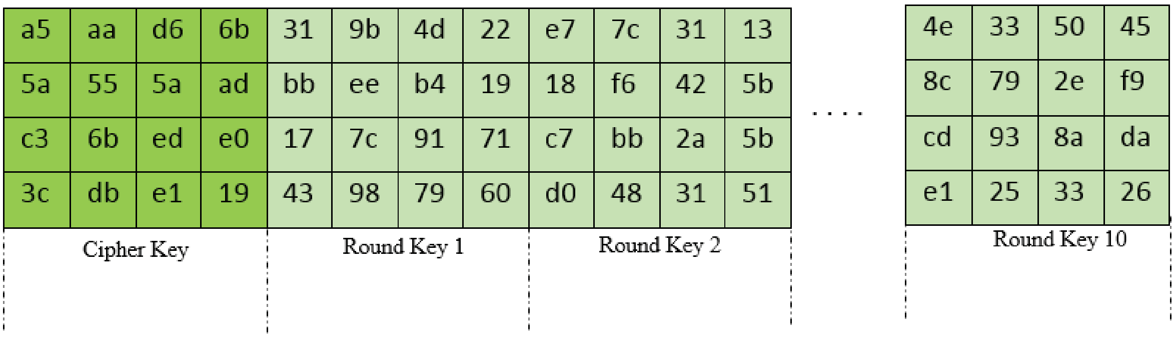


a5	aa	d6	6b	31	9b							....				
5a	55	5a	ad	bb	ee											
c3	6b	ed	e0	17	7c											
3c	db	e1	19	43	98											

d6		9b		4d
5a	⊕	ee	=	b4
ed		7c		91
e1		98		79

Figure 8: Third column generation

The above 6 steps needs to be repeated in order to generate all the 10 round keys, therefore generating the 44 byte key schedule. Finally the key schedule that obtained is shown below.



a5	aa	d6	6b	31	9b	4d	22	e7	7c	31	13	....	4e	33	50	45
5a	55	5a	ad	bb	ee	b4	19	18	f6	42	5b		8c	79	2e	f9
c3	6b	ed	e0	17	7c	91	71	c7	bb	2a	5b		cd	93	8a	da
3c	db	e1	19	43	98	79	60	d0	48	31	51		e1	25	33	26
Cipher Key				Round Key 1				Round Key 2					Round Key 10			

Figure 9: Generated round keys

Each round-key of the above key schedule is used in each transformation round of the encryption process described in the next section.

Encryption process

An encryption process involves the following processing steps to convert a plain-text into a secure cipher-text.

1. Initial round
2. Rounds
   i. Substitute byes or Sub Bytes
   ii. Shift Rows
   iii. Mix Columns
   iv. Add Round Key

3. Final round (No Mix Columns)
   i. Sub Bytes
   ii. Shift Rows
   iii. Add Round Key

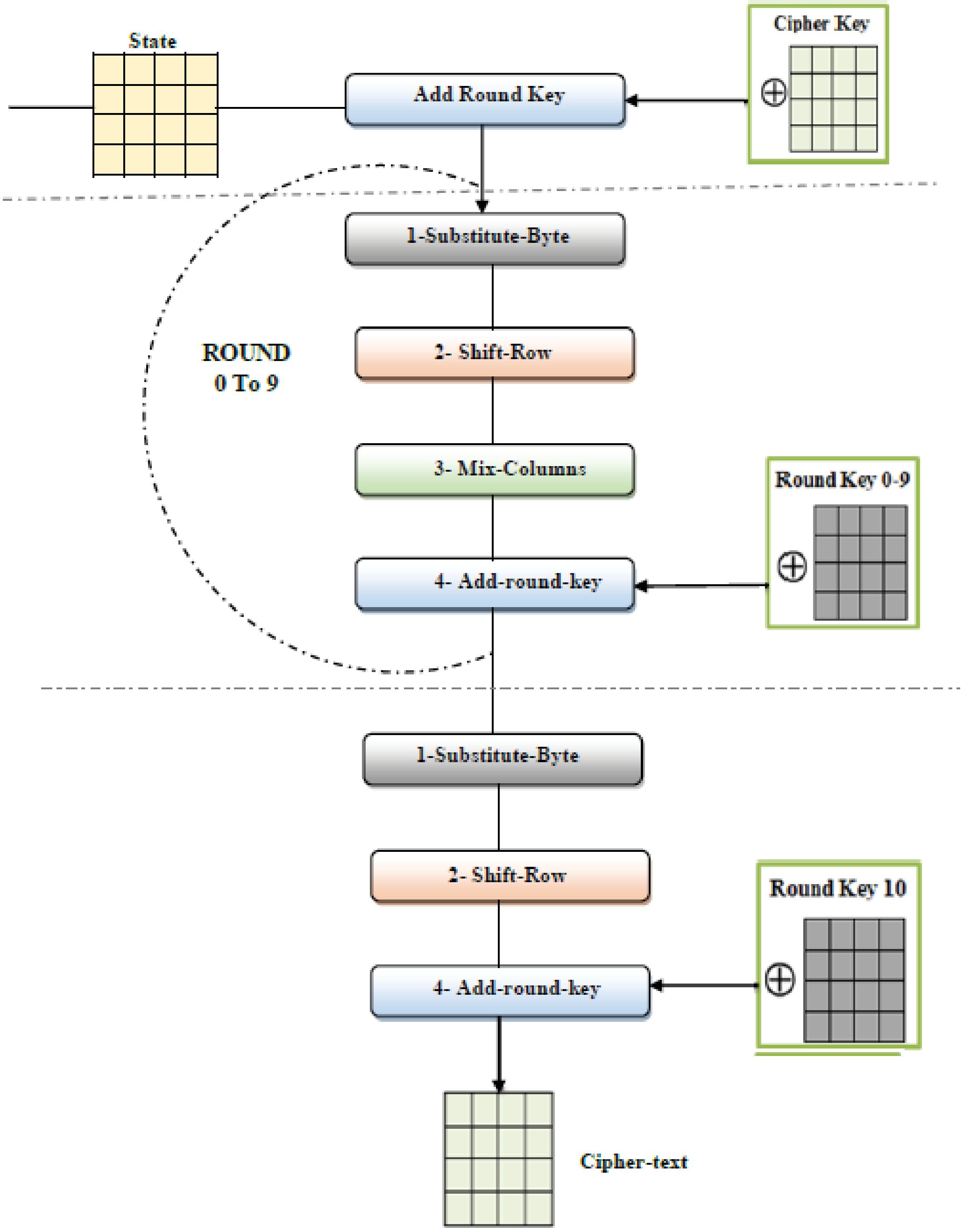
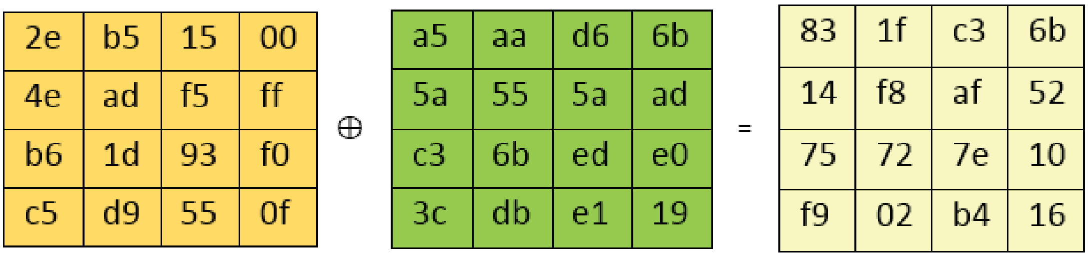


Figure 10: AES encryption [39]

Figure 10 shown above clearly depicts the encryption process. As can be observed in the figure, the encryption process consists of 10 rounds, with the first 9 rounds consisting of four different transformations. The last round does not involve one of the four transformations namely the mixing of columns. (As mentioned earlier, the number of rounds is 10, 12 and 14 for 128-bit, 192-bit and 256-bit long key size respectively). The description of each step in the encryption process is produced in detail below with the considered example matrix shown in figure 1.

Initial round:

In the initial round, the state matrix is processed through the Add Round Key transformation function. In this Add Round Key transformation, the cipher-key is added to the state. Note that here the cipher-key acts as the round-key. The cipher- key is added to the state by combining each byte of the state with the corresponding byte of the cipher-key using bitwise XOR as shown below.



$$\begin{array}{|c|c|c|c|}\hline 2e & b5 & 15 & 00 \\ \hline 4e & ad & f5 & ff \\ \hline b6 & 1d & 93 & f0 \\ \hline c5 & d9 & 55 & 0f \\ \hline \end{array} \oplus \begin{array}{|c|c|c|c|}\hline a5 & aa & d6 & 6b \\ \hline 5a & 55 & 5a & ad \\ \hline c3 & 6b & ed & e0 \\ \hline 3c & db & e1 & 19 \\ \hline \end{array} = \begin{array}{|c|c|c|c|}\hline 83 & 1f & c3 & 6b \\ \hline 14 & f8 & af & 52 \\ \hline 75 & 72 & 7e & 10 \\ \hline f9 & 02 & b4 & 16 \\ \hline \end{array}$$

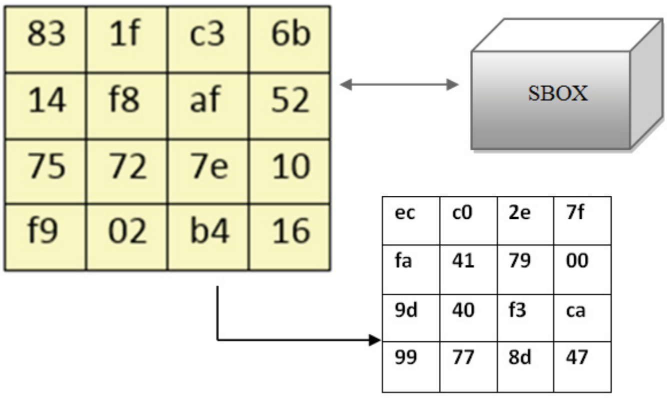
Figure 11: Initial round

Rounds:

It consists of 9 rounds in the case of AES-128. Each round consists of four different transformation steps explained below.

1. Substitute-Byte/Sub-byte function ‘‘sub-byte ()’’:

In this step, each byte in the state matrix (obtained from the previous transformation step) is replaced with its substitute byte using an 8-bit substitution box, the Rijndael’s S-Box. [See section] The figure below depicts the substitute function. This provides non linearity to the cipher.



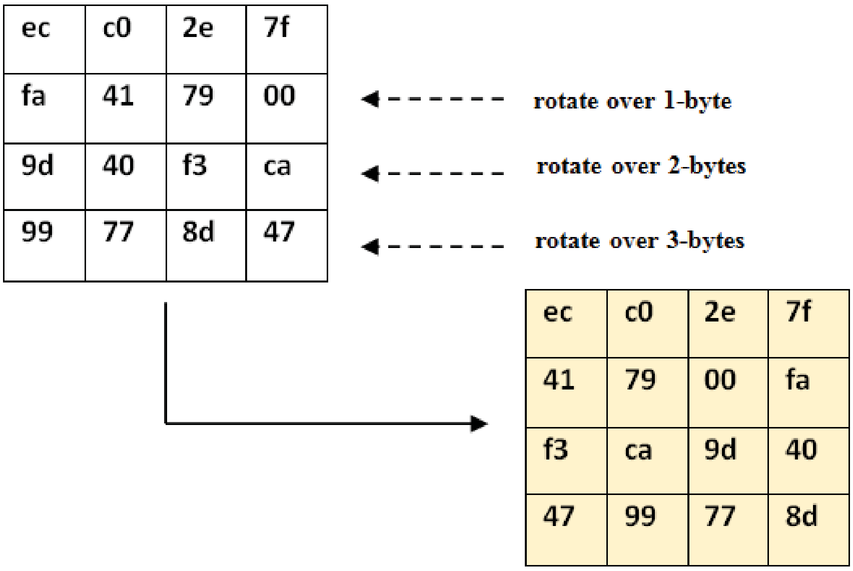
83	1f	c3	6b
14	f8	af	52
75	72	7e	10
f9	02	b4	16

ec	c0	2e	7f
fa	41	79	00
9d	40	f3	ca
99	77	8d	47

Figure 12: Substitute-byte function

2. Shift-Rows function/shift-rows ():

The shift-rows function operates on the rows of the state (obtained from the previous transformation step). It cyclically shifts the byte in each row by a certain offset. The first row is left unchanged. Each byte of the second row is shifted once to the left. The third and fourth rows are shifted by offsets of two and three respectively as shown in figure below.



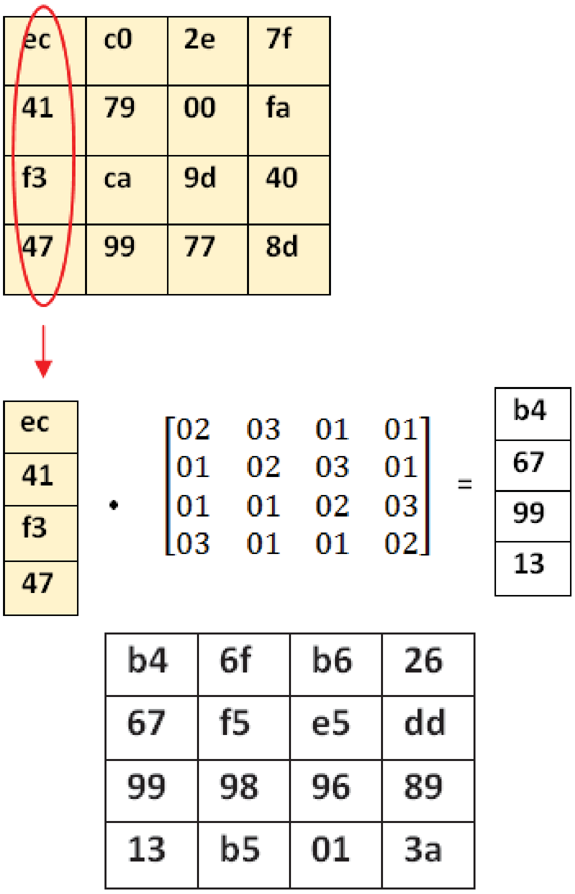
ec	c0	2e	7f
fa	41	79	00
9d	40	f3	ca
99	77	8d	47

ec	c0	2e	7f
41	79	00	fa
f3	ca	9d	40
47	99	77	8d

Figure 13: Shift-Rows function

3. Mix Columns function/mix-columns ():

In this function, the four bytes of each column of the state (obtained from the previous transformation step) are transformed using an invertible linear transformation. It takes four bytes as input and outputs four bytes. During the operation, each column of the state is multiplied with a 128-bit key as shown in the figure 14. The multiplication operation is defined as, multiplication by 1 indicates no change, multiplication by 2 indicates a left shift, and multiplication by 3 indicates shift to the left and then performing XOR with the initial un-shifted value. If the shifted value is larger than 0XFF then a conditional XOR with 0X1B should be performed after shifting.



ec	c0	2e	7f
41	79	00	fa
f3	ca	9d	40
47	99	77	8d

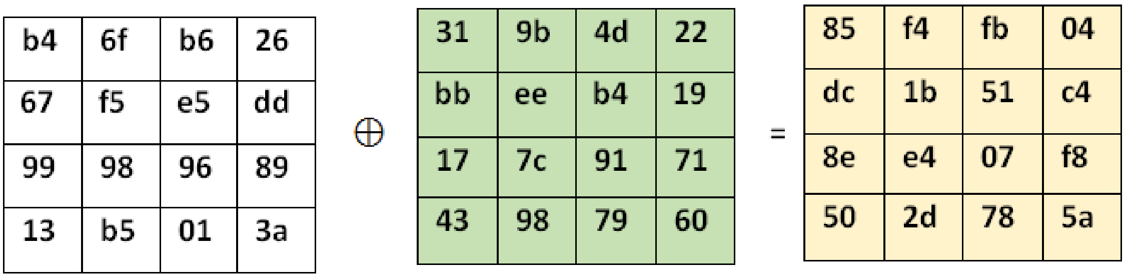
$$\begin{bmatrix} ec \\ 41 \\ f3 \\ 47 \end{bmatrix} \bullet \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} = \begin{bmatrix} b4 \\ 67 \\ 99 \\ 13 \end{bmatrix}$$

b4	6f	b6	26
67	f5	e5	dd
99	98	96	89
13	b5	01	3a

Figure 14: Mix columns function

4. Add Round Key Step

As mentioned previously in initial round, in Add Round Key step the round-key is combined with the state (obtained from the previous transformation step) using XOR operation. For each round, a sub-key/round-key is derived from the main cipher-key using Rijndael's key schedule. The key schedule has already been explained in the previous sections. The Round-keys generated through the key scheduling process shall be used in the encryption process for Add Round Key transformation. For the first round i.e. after the initial round, the Round-Key 1 is used. Similarly, for second round, the round- key 2 is used and so on. The Add Round Key is defined as adding each byte of the state with the corresponding byte of the sub-key/round-key using bitwise XOR.



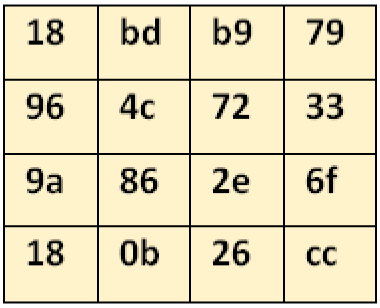
$$\begin{bmatrix} b4 & 6f & b6 & 26 \\ 67 & f5 & e5 & dd \\ 99 & 98 & 96 & 89 \\ 13 & b5 & 01 & 3a \end{bmatrix} \oplus \begin{bmatrix} 31 & 9b & 4d & 22 \\ bb & ee & b4 & 19 \\ 17 & 7c & 91 & 71 \\ 43 & 98 & 79 & 60 \end{bmatrix} = \begin{bmatrix} 85 & f4 & fb & 04 \\ dc & 1b & 51 & c4 \\ 8e & e4 & 07 & f8 \\ 50 & 2d & 78 & 5a \end{bmatrix}$$

Figure 15: Add round key

The above four transformation functions are repeated 9 times to cover 9 rounds. The output of the 9th round is processed through the final transformation round.

Final round:

The final round is similar to the previous rounds, except that this round skips the third transformation step i.e. the Mix Column function. Therefore the transformation steps involved in the final round are; The Sub-byte, the Shift-Row and Add Round Key function. The output of the final round is the required cipher-text. The cipher-text generated for the considered plaintext is shown in figure 16.



18	bd	b9	79
96	4c	72	33
9a	86	2e	6f
18	0b	26	cc

Figure 16: Final round

2.2 PARALLEL COMPUTING

2.2.1 Why parallel computing?

With a faster computer one can solve problems consisting of heavy computations faster. In the case of interactive applications it can give better responsiveness. Second important thing once can do is get better solutions in the same amount of time. It helps increase the resolution of models and allows adding extra sophistication to the process. Parallel computing is an attempt to speed up solution of a particular task through certain techniques, one of the techniques being, dividing the particular task into sub tasks and then executing them simultaneously.

Microprocessors based on single central processing unit (CPU), have driven rapid performance improvement since years and still continue the trend. Many hardware and software improvements have been made in order to get faster performance. These improvements include increase in the clock speed, optimizing hardware by techniques like instruction prefetching, reordering, pipelined functional units, branch predictions, hyper-threading etc. A famous quote comes from Herb Shutter which says “the free lunch is over”. What this means is that, the clock speed are no longer going to be increasing exponentially like before when it used to double every 18 months to 2 years according to Moore’s Law [37]. The reason for clock speeds being flattering out is, excessive power consumption which leads to heat dissipation and current leakage. This leads to increase in the need for additional cooling hardware which in turn increases the cost of sophistication. Additionally it requires difficult design and verification. Improvement in the hardware demands for more silicon to be devoted to control hardware. Due to many such complications computer industry could not any more rely on improvements in hardware and software, instead it needed a change in the approach of improving speed. One of such approach was to shift to multi core CPUs and this gave rise of parallel computing.

2.2.2 Single core Vs Multi core processor

A single core CPU implies a computing component with a single computing unit or core. Figure 17 below shows the architecture of a single core CPU.

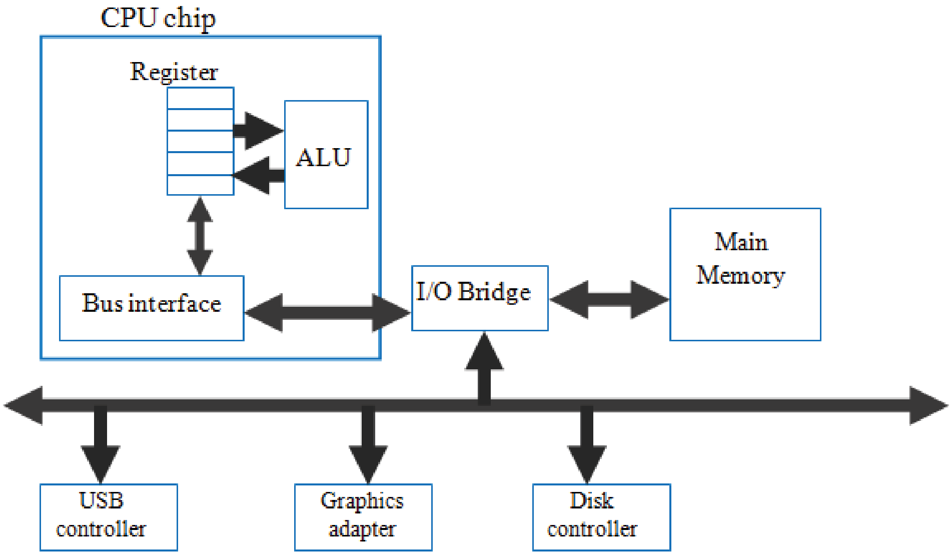


Figure 17: Single-core processor [40]

A multi core processor is a computer system with more than one core on a single chip. The cores can be identical, referred to as homogeneous multi-core system or a heterogeneous multi-core system with different types of core. A homogeneous multi-core processor contains the replica of the CPU chip on the same silicon die. These cores run in parallel. Each core contains several threads and within each core the threads are time-sliced. The operating system comprehends each core as a separate processor and each thread in a processor as a separate virtual processor. Figure 18 below shows dual-core processor architecture with a private L1 cache and shared L2 cache.

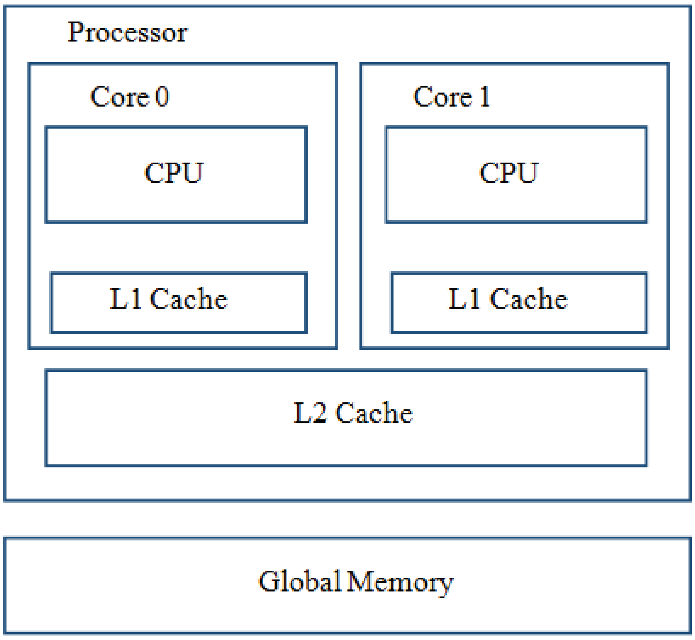


Figure 18: Dual-core processor [36]

A multi-core processor is shared memory multiprocessor which works on the principal of Multiple Instructions Multiple Data (MIMD). In MIMD different threads can operate on different parts of the memory. This principal converts the sequential process into a simultaneous process where the tasks are independent of each other increasing the speed of execution.

There are various ways of exploiting parallelism namely Doman decomposition, Task decomposition and Pipelining. In domain decomposition a large amount of data is divided into chunks of small data which is processed by each thread independently. Each thread executes the same instruction with its set of data. After all the data is executed the output from all the threads together forms the final data. Another way of parallelising is task decomposition in which a large sophisticated task is divided into sub tasks and each task is executed independently by each thread. Two threads of a single core cannot work simultaneously on the same functional unit. Pipelining is another way of parallelising a task, where two threads actually executes two tasks concurrently overlapping the execution and making it faster. This research work is a case of Domain decomposition, also known as Data decomposition. In order to exploit parallelism efficiently a programmer needs to make few considerations.

- Check if the application is suitable to be parallelised i.e., it consists of parts that are independent of each other and can be executed simultaneously.
- Identification of which part of the algorithm or task can be parallelised.
- Distribution of tasks or data such that all the threads have their required part of data
- Avoid cache coherence problem.
- Synchronisation of execution.

With the introduction of POSIX Threads, usually referred to as Pthreads which is a POSIX standard for threads it has become easy and flexible to efficiently parallelise an algorithm on a multi-core CPU. The Pthreads is a standardized C language threads programming interface specified by the IEEE POSIX 1003.1c standard [34].

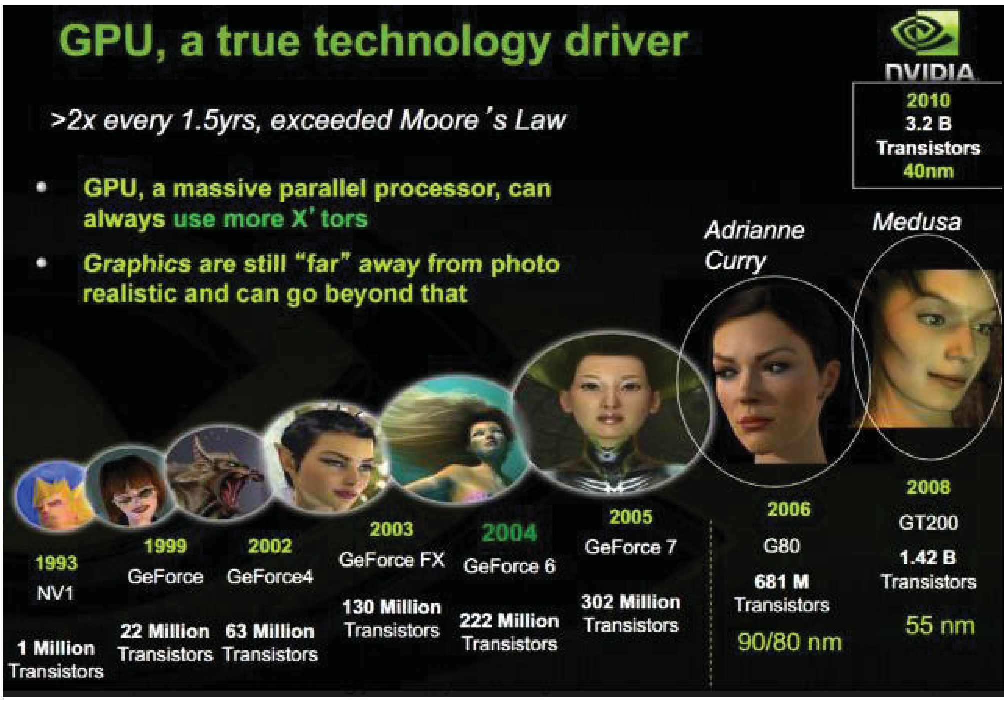
2.3 GPU COMPUTING

2.3.1 Evolution of GPGPU

Before one begins to talk about GPUs in high performance parallel computing it is worthwhile to look back and see the evolution history of GPU. Even today GPU is considered to have a marked job i.e., the entertainment industry.

Early GPUs were designed specifically for graphics applications. It settled out somewhere around early 90s with a fixed pipeline/function. Those were the devices that had specific silicon for specific operation like shading a triangle and so on. Then at one point it was realized that there is much more flexibility if one can add programmability on these devices. And eventually there were the vertex shaders and the pixel shaders which were basically either pieces of programs that operated on 2D data structures or pieces of programs that operated on 3D data structures. It was not until 2008, when it was realized that one can use the same programming interface on the same silicon to do both 2D and 3D operations. And that is basically where the Compute Unified Device Architecture (CUDA) was born. Since then all the newer generation GPUs are CUDA capable and will remain CUDA capable in the future. Size of the device is proportional to the amount of transistors that is located on the device.

With the improvement in programmability, programmers started exploring GPUs for scientific computing. At the beginning it actually had the same architecture that had the split between the pixel and vertex shaders. But with the advent of CUDA there was just much more activity in academia for doing GPU computing. And today it has become the main stream in the world of scientific computing. These new trends of GPU that are used to general purpose computing applications are usually referred to as General Purpose Graphical Processing Unit (GPGPU).



GPU, a true technology driver

>2x every 1.5yrs, exceeded Moore's Law

- GPU, a massive parallel processor, can always use more X'tors
- *Graphics are still "far" away from photo realistic and can go beyond that*

Figure 19: Evolution of GPU [19]

2.3.2 Architectural Difference in GPU and CPU

CPU and GPU have fundamentally different design philosophies as illustrated in figure 20.

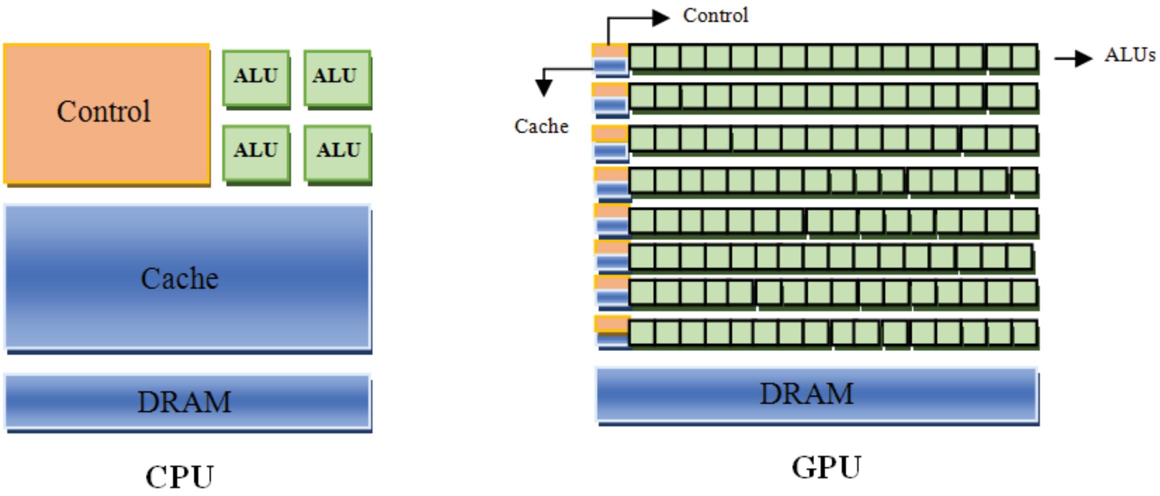


Figure 20: CPU and GPU architectural difference [42]

The CPU is optimized for minimal latency where one works towards being able to quickly switch between different operations. GPUs are optimized for throughput so what one can push as many operations through the device as possible. In order to get low latency on the CPU there are lot of infrastructure on the chip such as large caches to make sure that one has massive amount of data readily available. There is lot of control flow silicon. There are actually few parts dedicated towards computing. On the other hand, in the GPU the balance has shifted, wherein one needs tons of arithmetic and logic unit. It is specialized for compute intensive, highly parallel computations and therefore is designed such that more transistors are devoted to computing rather than memory and control [9]. The L2 cache can shrink because the amount of time that a GPU takes to get data from the DRAM is not a major concern as long as there is sufficient amount of work in the application. This sufficient amount of data is required to hide the latency what is introduced while fetching data from the global memory or while processing a complex operation. That means a GPU needs to use massive number of threads in order to tolerate latencies.

The traditional GPU used OpenGL platform is not suitable for AES computation as it is confined to floating point data and unavailability of bitwise logical operations unlike AES that involves huge mathematical computations and uses bitwise logical operations in every transformation round [9]. OpenCL can be a good platform to implement AES encryption. However, CUDA is optimized for NVIDIA GPUs. This research uses the latest NVIDIA GPU that supports CUDA platform, as it is flexible and efficient to be used for AES encryption.

2.3.3 CUDA programming model

Speed of the program accelerates only if the software efficiently exploits the parallelism provided by the hardware architecture underlying in the multiprocessor. There is hence a need to develop the algorithm in such a way, that it can effectively use the multithreaded structure and contribute towards potentially increasing speed of execution. To develop such an algorithm, a parallel programming model is required that would support the parallel programming environment. CUDA is a parallel computing platform and programming model introduced by NVIDIA to be used in the GPUs that they produce [10]. This model provides developers access to the memory of CUDA compatible GPUs and the virtual instruction set. The CUDA C compiler efficiently manages hardware resources and provides functions libraries which hide the hardware resources making it possible for the programmer to efficiently exploit parallelism in the program without being an expert in the lower-level hardware architecture of GPU [9]. Programmers who are familiar with C programming language can develop the GPGPU applications easily.

2.3.3.1 CUDA Thread organisation

The GPU is a single instruction multiple thread (SIMT) computing device for CUDA. A thread is the fundamental unit of CUDA programming model. Every thread executes the same instruction with a specific data. The instruction is known as CUDA kernel or simply kernel. The CUDA threads are hierarchically organized. Figure 21 presents the organization of threads in a grid. A grid consists of a 3-Dimensional array of thread-blocks each intern consisting of a 3-Dimensional array of threads. Every block within a grid holds a unique block ID to be differentiated from other blocks. Similarly every thread within a block holds a unique thread ID so as to be differentiated from other threads in the block. The grid has a total of N*M threads, where N represents the number of blocks and M represent the number of threads in each block. Interestingly, every thread and blocks holds together a unique ID known as the thread-block ID to be differentiated from any other thread in the entire device, i.e., by the ID of a thread it can be easily identified which thread of which block it belongs to.

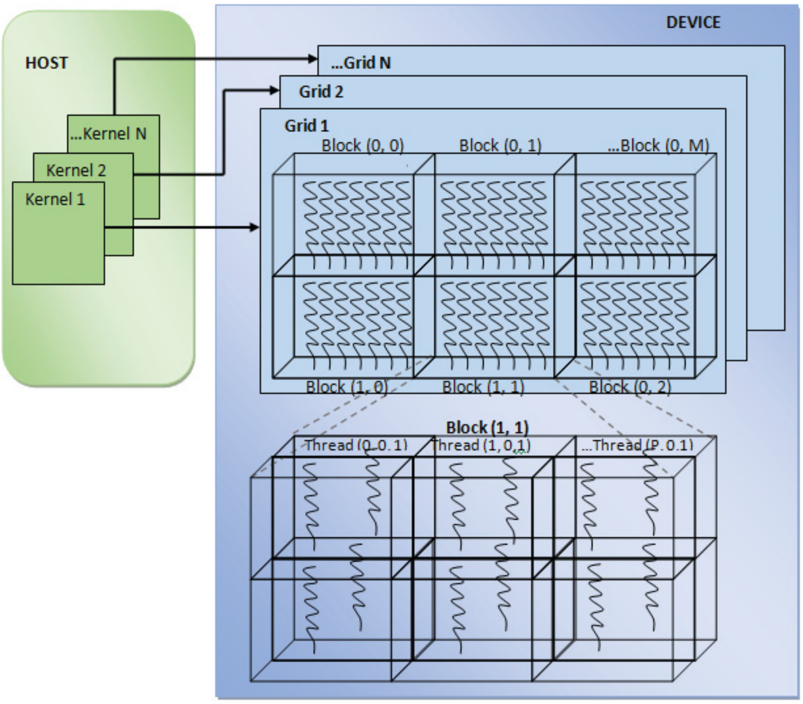
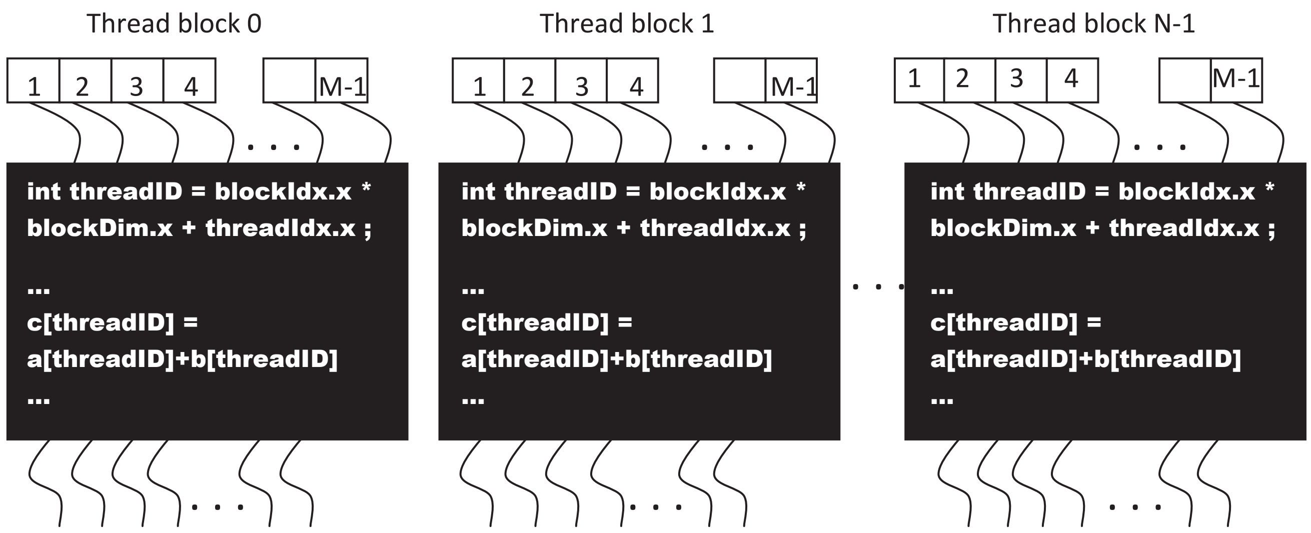


Figure 21: CUDA thread organization [37]

A grid represents a single device (GPU) and the Host (CPU) can handle multiple such grids. Figure 22 depicts the distribution of thread IDs and block IDs. All the threads in a grid execute the same kernel function. They depend on their unique IDs to compute memory addresses and make control decisions.



```
int threadID = blockIdx.x *
blockDim.x + threadIdx.x ;

...
c[threadID] =
a[threadID]+b[threadID]
...
```

```
int threadID = blockIdx.x *
blockDim.x + threadIdx.x ;

...
c[threadID] =
a[threadID]+b[threadID]
...
```

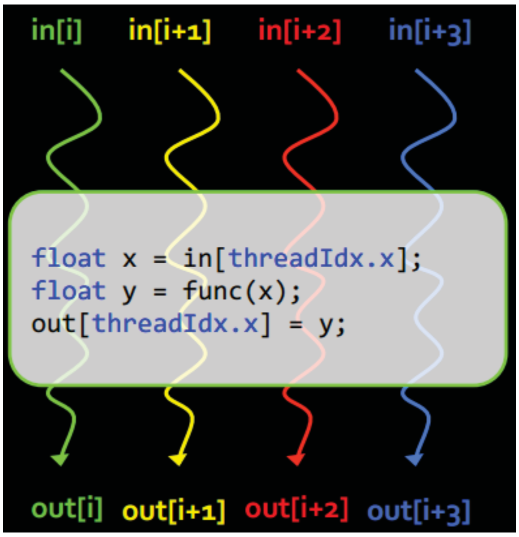
```
int threadID = blockIdx.x *
blockDim.x + threadIdx.x ;

...
c[threadID] =
a[threadID]+b[threadID]
...
```

Figure 22: CUDA thread distribution [37]

The thread organization is determined through the configuration provided during the kernel launch.

$$threadID = blockIdx.x \times blockDim.x + threadIdx.x$$

The first dimension $'blockIdx.x'$ indicates the dimension of the grid in terms of number of blocks. The parameter $'blockDim.x'$ represents the dimension of each block in terms of number of threads. For example thread ‘2’ of block ‘4’ has a $'threadID'$ value $(4 \times M + 2)$. Each thread executes the same code, but with different parts of data and may take different paths.



```
float x = in[threadIdx.x];
float y = func(x);
out[threadIdx.x] = y;
```

Figure 23: CUDA thread functioning [37]

Different combinations of the threads and blocks of a device can affect the execution of a kernel. This research work examines the effect of different thread block combinations on the algorithm.

2.3.3.2 CUDA Memory model

In CUDA, the host and the device have separate memory spaces. GPUs are typically hardware cards that come with its own memory. In order to execute a kernel in the device, the programmer needs to allocate memory within the device to store the necessary data for execution. Figure 24 below shows the CUDA device memory model. It includes registers, shared memory, constant and texture memory and Global memory. The communication between the device and host takes place through the global memory. When a data is transferred by the host through the PCI bus, it is stored in the global memory and the GPU can access this memory for data to be used during execution. In the same way the GPU after execution deliver the data back to the CPU via the PCI bus. The global memory and the constant memory are shared by all the blocks in the device. Every block has its own shared memory which is shared by all the threads within that block. Each thread within a block has its own private memory or local memory and registers.

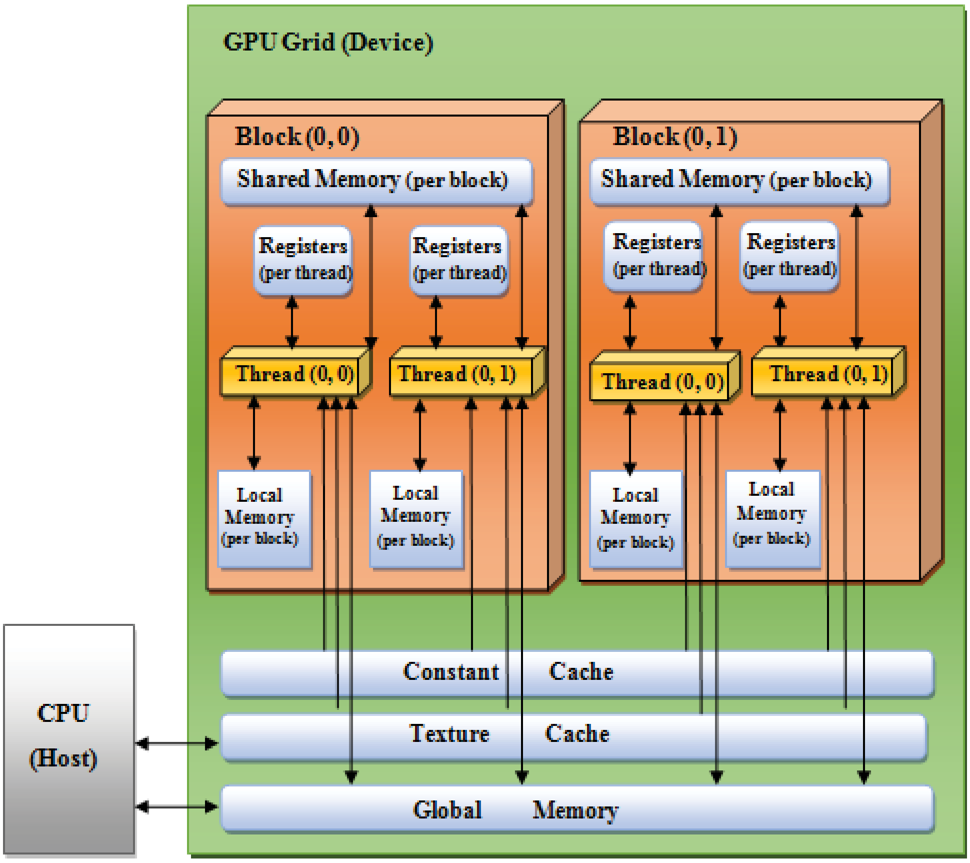


Figure 24: CUDA memory model [37]

2.3.3.3 CUDA programming

In CUDA a kernel is launched by executing the following step.

- Allocate memory in the device to store the input data.
- Copy the data from the CPU memory into the allocated memory in the GPU.
- Execute the kernel in the GPU by selecting necessary grid dimension.
- Copy the results back to the CPU.
- Free the allocated memory in the GPU.

Figure below shows a basic host program on the host to execute a function on the device.

```
int main ( ){
int *h_a, *h_b, *h_out;                                              // Host copy of inputs
int *d_a, *d_b, *d_out;                                              // Device copy of inputs
cudaMalloc ((void**)&d_a, N*sizeof(int));                    //Memory allocation for inputs in device
cudaMalloc ((void**)&d_b, N*sizeof(int));
cudaMalloc ((void**)&d_out, N*sizeof(int));
cudaMemcpy(d_a, h_a, N*sizeof(int), cudaMemcpyHostToDevice);     //Copy input to the device
cudaMemcpy(d_b, h_b, N*sizeof(int), cudaMemcpyHostToDevice);
encryption<<<blocks,threads>>>(d_state,d_key,d_cipher);           //Execute kernel on the device
cudaMemcpy(out,d_out,N*sizeof(iht),cudaMemcpyDeviceToHost);      //Copy the results to the host
cudaFree(d_a);cudaFree(d_b);cudaFree(d_out)                        //Free memory allocated on host
```

Figure 25: CUDA host program on CPU

The instructions shown in the figure above is executed in a sequential manner. The kernel execution has to wait until all the data is stored into the GPU memory, and after execution it needs to send all the output data to the CPU before it can be displaced for the user. The introduction of CUDA STREAMS has given a new possibility for CUDA programmers to be able to execute the function in a smarter way.

2.3.4 CUDA STREAMS

A stream can be defined as a sequence of operation in CUDA, executed in an order, released by the host. Operations in different streams can be interleaved and run concurrently. CUDA STREAMS allow the overlapping of communication and execution of kernel using CUDA on the GPU. The various instructions that can be processed concurrently using CUDA STREAMS are:

- CUDA Kernel execution.
- Data transfer from Host to Device
- Data transfer from Device to Host.
- Operations on CPU.

In case of data parallelism, the large data is divided into segments and processed using CUDA STREAMS as shown in figure 26 below. The figure depicts the performance improvement against a serial CUDA implementation.

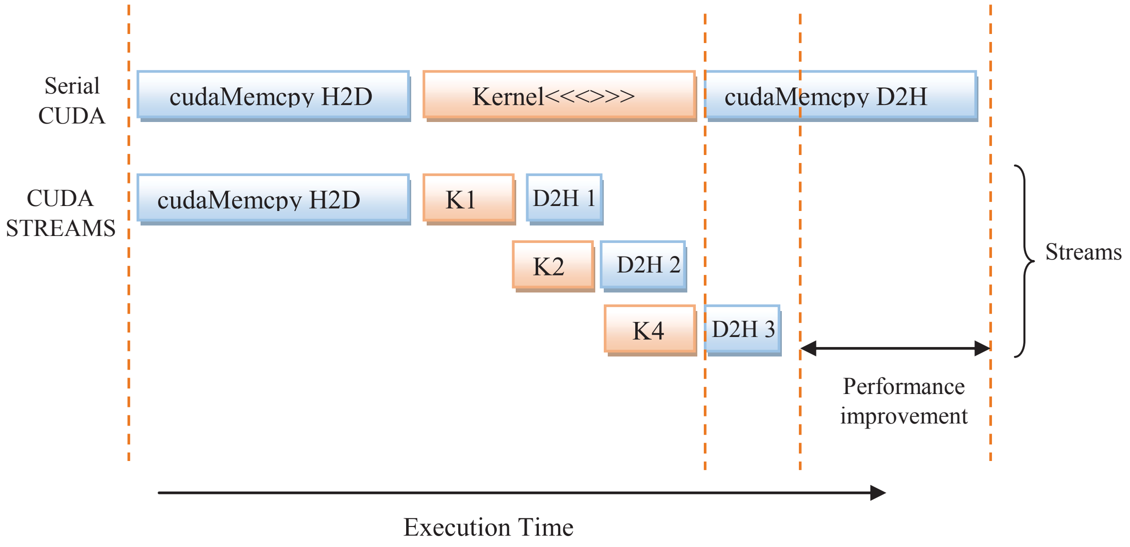


Figure 26: CUDA STREAM execution model [33]

Another efficient way of exploiting parallelism using CUDA STREAMS as implemented in this research work is shown below.

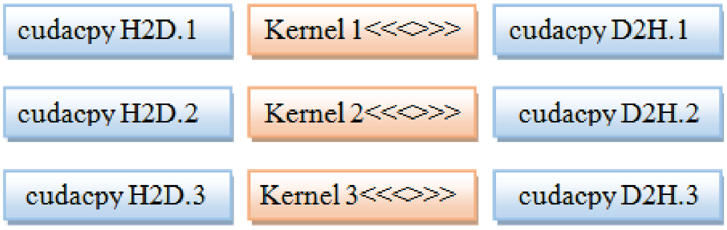


Figure 27: Optimized CUDA STREAM execution model [33]

The effect on the performance of a program using CUDA STREAMS cannot be simply defined as it depends on the task complexity and data size. When in some cases CUDA STREAMS can act as an accelerator, while there are also cases when simply using a CUDA program performs better than using streams.

2.4 RELATED WORK

The exponential increase in data generation and the growth of several security hazards have forced researchers to search for better and faster implementation of encryption algorithms to avoid data vulnerability. To tackle the problem of security threats and complexity various algorithms have been introduced and implemented. The next important issue to deal with is to reduce the encryption time of the algorithms. Optimizing the algorithm itself would do this, but to a limited extent. The introduction of multi-threading, multi-core CPUs and GPUs have been a major breakthrough in decreasing the processing time of a program by exploiting parallelism in the algorithms and processing them using multiple processing units.

Early comparisons of CPU and GPU performance in encrypting data have been done on various encryption algorithms including AES, DES, 3DES, RSA and many more. This paper is mostly concerned with the AES implementations. The AES implementations performed by previous researchers are quite motivating and intellectual. However when few of them fail to exploit the parallelism on the Graphics device efficiently; few others have ignored effective computation time including data transfer time due to which the results and inferences are incomplete. There is still insufficient information to be able to accurately predict the best utilization techniques of a Graphic Card for implementation of an algorithm. This section presents few previous research works on this area. This research paper takes the previous research work as a motivation. Considering the limitations in previous research works as a research gap, this paper aims to exploit efficient parallelism on the GPU, and on multi-core CPU to make a fair and reliable comparison. Also it aims to deduce implementation techniques on multi-core CPU and GPU, in order to utilize them for future implementations.

Parallel implementation of AES algorithms is not a new topic of interest. Previously, works on implementing AES algorithm on multi-core CPU and GPU have been done. Shao et al in research paper [4] suggests, AES algorithm implemented using a 16-core GPU @ 550MHz using openssl reduced the computation time by 1.6 times, when compared to a Pentium(R) Dual-Core E5200 @ 2.5GHz CPU . This paper however ignores the latency caused due to transfer of data between the host and device and vice versa which is an important factor of concern in GPU computing. Also the performance improvement is quite small to claim that GPUs can efficiently outperform CPU for AES encryption.

As discussed in Le et al, the AES algorithms have been implemented using data parallelism [9]. The paper boasts a boost of 7x speedup of AES parallel algorithm when compared to its sequential version for data size beyond 200 Mbytes, where as the performance boost drops down to 4 and 2 for data size below 10 Kbytes. For a significant speeding up of the algorithm, which is the ratio of GPU to CPU execution time, the data input should be large. The larger the input data the higher is the speedup value. However, the key expansion block of the encryption function is done in CPU rather than on a GPU, so this could be regarded as partial parallelism of the algorithm. This raises the question of how complete parallelism performance would be, which this paper tries to solve.

Li et al in research paper [5] concentrates on the improving the throughput of the AES algorithm. The paper compares similar implementations on various models of NVIDIA. The author suggests that the efficiency of AES algorithm can be improved by an OpenCl implementation of Electronic Codebook mode Encryption and Cipher Feedback mode Decryption, however it entails a performance penalty when compared to CUDA implementation. Also the key expansion is performed on the CPU before transferring the round keys to the device similar to another research paper [1] by Luken et al.

In a report by Maksim [7], the performance of AES-128, AES-192 and is tested using various numbers of threads per block. The number of threads allocated per block is 64, 128, 256 and 512. These were done on the three variants of AES-128, AES-192 and AES-256. This resulted in highest throughput of 400+MBps when 64 threads per block were used. The throughputs have been reported to be less than CPU for data sizes less than 1MB and exponentially growing for data sizes exceeding 1MB. Higher throughputs and lesser execution times are achieved when the input file size is larger. This research work takes this paper as a motivation and aims to present the thread block combination effect more intensely.

In another research paper [10], Zhao et al proposed a CUDA implementation of ECB mode encryption and CBC mode decryption of AES algorithm to achieve a execution speed up to 50 times on NVIDIA Tesla C2050 GPU over sequential implementation on Intel Core I7-920 which is a 4 core processor with 8 virtual cores. The comparison is based on a single threaded CPU implementation which makes the results outdated and incomplete to be fairly compared with that of parallel GPU computation. Also the data transfer time is not considered which is a major drawback of the paper.

This research aims to implement AES encryption considering the limitations in the previous research works in order to acquire more reliable and complete results.

CHAPTER 3 METHODOLOGY

This section is a crucial part of the research work as it helps to understand the cause and effect for the problem in hand. This research work uses "the experimental method" also known as the quasi experiment, in which a researcher observes the consequences while actively influencing something. In other words, it is a systematic scientific approach of research in which a researcher measures any change in other variables while controlling and manipulating one or more variable.

The research work is carried out in two sections

1. Literature Review

   A literature review allows one to gain and demonstrate skills in both information seeking and critical appraisal. It also helps to generate a hypothetical analysis of the outcome of research. The purpose of literature review in this research is fill the knowledge gap on parallel programming techniques, GPU programming, efficient algorithm build, AES encryption. Programmer needs to have clear understanding of CPU and GPU architectural disputes, CUDA programming model, and Multi-core processor architecture.

   To carry out literature review:

   - Latest research articles organized around and related directly to the thesis is studied and analysed.
   - Books and articles are studied in order to complete the knowledge gap in the area.

2. Experiment

   An experiment is a systematic procedure carried out to verify, refute and establish the validity of a hypothesis. The experiment performed in this research work is a controlled experiment which helps providing insight into cause-and-effect of demonstrating the outcome. An experiment completes a research work fully. The experiment details are described in this section.

3.1 PRE-REQUISITE FOR EXPERIEMENT

- A CUDA-capable GPU
  - NVIDIA Quadro K4000
  - Architecture: Kepler
  - 768 processing cores, 32 cores in each Streaming Processor.
- A supported version of Microsoft Windows
  - Windows 7
- A CUDA-supported Microsoft Visual Studio
  - Microsoft Visual Studio 2012
- NVIDIA CUDA toolkit
- CPU
  - Intel® Xeon® CPU E5-1650
  - No of Cores: 6
  - No of threads: 12 (2 logical cores per physical)

- Clock speed: 3.2 GHz

3.2 EXPERIMENTATION SETUP

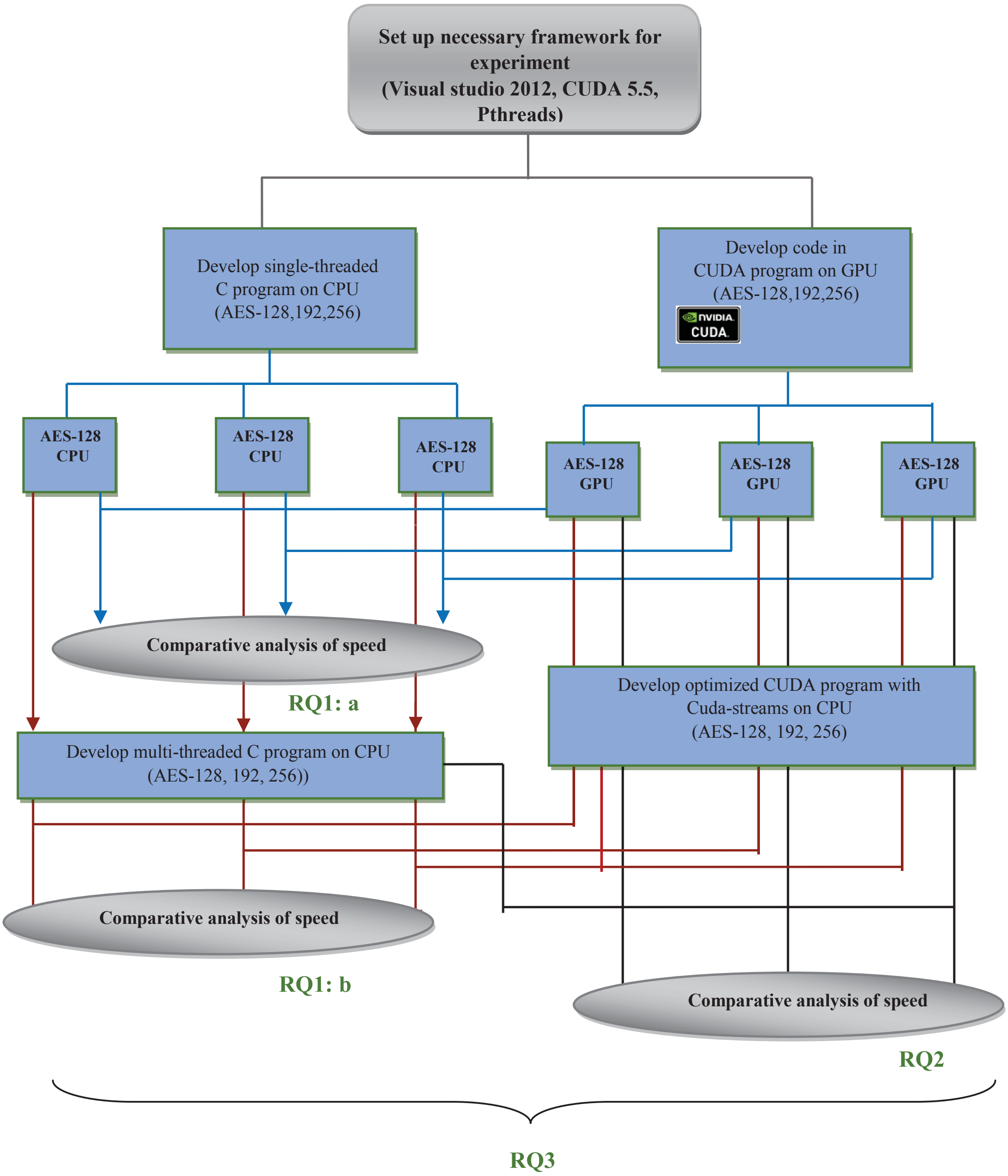

Set up necessary framework for experiment
(Visual studio 2012, CUDA 5.5, Pthreads)
Develop single-threaded C program on CPU (AES-128,192,256)
Develop code in CUDA program on GPU (AES-128,192,256)
NVIDIA CUDA
AES-128 CPU
AES-128 CPU
AES-128 CPU
AES-128 GPU
AES-128 GPU
AES-128 GPU
Comparative analysis of speed
RQ1: a
Develop optimized CUDA program with Cuda-streams on CPU (AES-128, 192, 256)
Develop multi-threaded C program on CPU (AES-128, 192, 256))
Comparative analysis of speed
RQ1: b
Comparative analysis of speed
RQ2
RQ3

Figure 28: Experiment setup

3.3 EXPERIMENT

3.3.1 Implementation on CPU

AES algorithm operates on an input size of 16 bytes using key size of 128 bits, 192 bits and 256 bits for AES-128, AES-192 and AES-256 version respectively. The algorithm is developed on Intel® Xeon® CPU E5-1650 in such a way that it takes a large set of data as input and encrypts it in chunks of 16 bytes each sequentially. The encryption key remains constant for each data chunk. The key is first expanded to generate the round keys which are then used in each round of the encryption process.

The experiment is performed with a data size of 32000 bytes and 32000*5 bytes. The data used is a long array of random integers. In each case, execution time is recorded. A single AES encryption process (k=1) involves execution time to:

1. Divide the state array (input array) into chunks of data.
2. Perform encryption on each chunk of data.
3. Store the results back into the output array.

The process is repeated for k=10, in order to increase readability of results. 20 readings of execution time are taken in each case. Its Average, Standard Deviation and Confidence Interval are calculated.

Implementation on single threaded CPU.

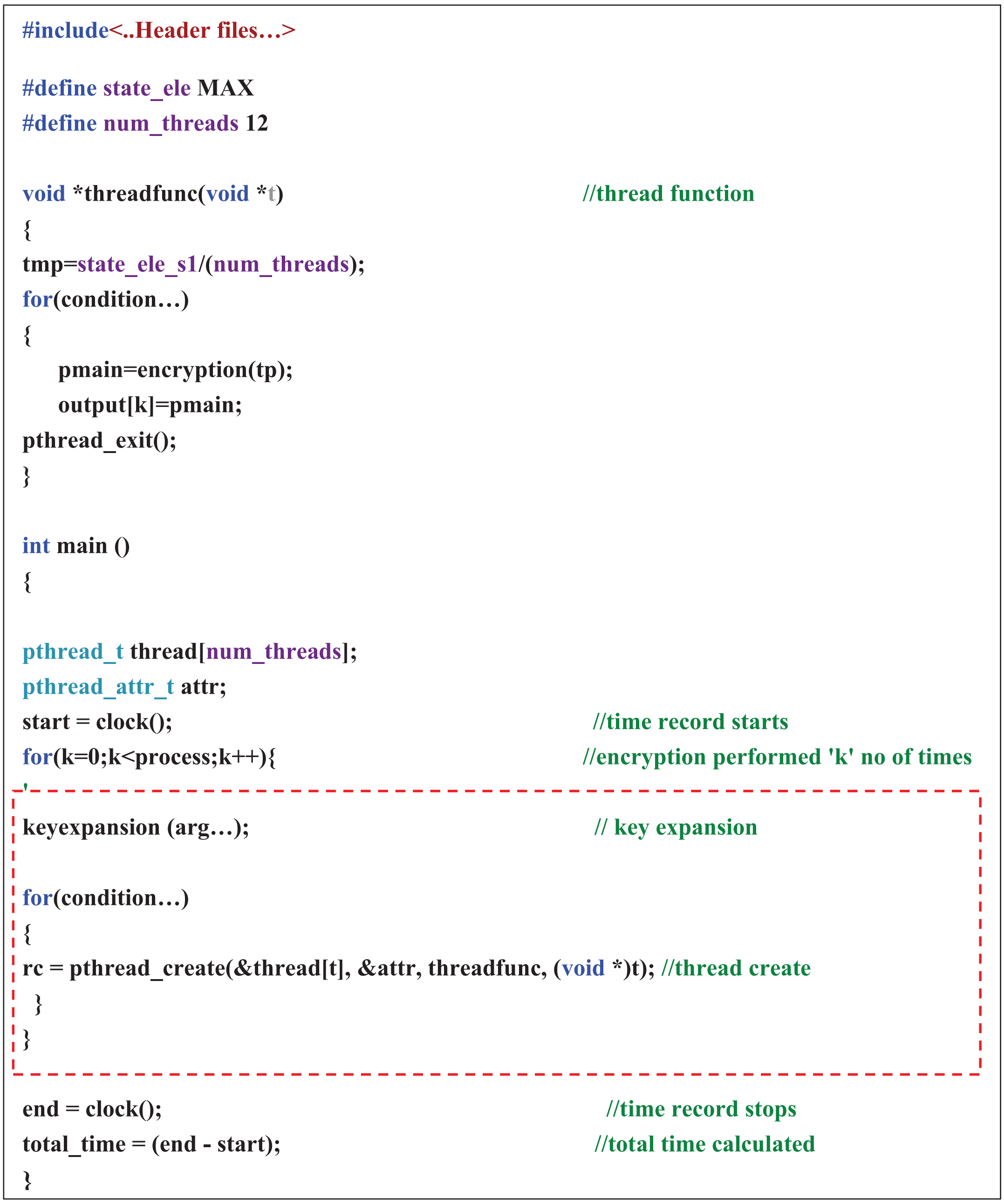
A single threaded CPU program implies that the program is executed by a single processor in the CPU. The structure of the program is shown below in figure 29. The instructions within the red box represent a process.

```
#include<..Header files…>
#define N MAX
int cipherkey[size];
int main()
{
int state[N];

for(i=0;i<N;i++){                                   //initialize input state
      state[i]=i;
  }
start = clock();                                    //time record starts
 keyexpansion (arg…);                               // key expansion

for(k=0;k<process;k++){                             //encryption performed 'k' no of times'


  for(…condition…
                                                    //data encrypted in chunks
            {
             chunk[j]=state[j];
            }
    encryption(chunk, cipherkey);                      //function call

     for(Condition…){
            state[j]=chunk[j];                      //results copied back to state array
            }
   }
   end = clock();                                     //time record stops
total_time = (end - start);                         //total time calculated
}
```

Figure 29: Single threaded CPU program structure

Implementation on multi-core CPU

A multi threaded program for AES algorithm is developed using POSIX threads. The CPU contains 6 physical cores, each containing 2 virtual cores. Effective utility of the CPU, in order to achieve highest performance for a fair comparison with other implementations, demands the use of all available cores efficiently. Hence 12 threads are utilized to process the entire data set. The program is developed such that the data is ~ equally divided among the threads. Each thread executes on its part of data independently and concurrently. The structure of the program is shown below in figure 30. The instructions within the red box represent a process.



```
#include<..Header files…>

#define state_ele MAX
#define num_threads 12

void *threadfunc(void *t)                              //thread function
{
tmp=state_ele_s1/(num_threads);
for(condition…)
{
    pmain=encryption(tp);
    output[k]=pmain;
pthread_exit();
}

int main ()
{

pthread_t thread[num_threads];
pthread_attr_t attr;
start = clock();                                        //time record starts
for(k=0;k<process;k++){                                //encryption performed 'k' no of times

keyexpansion (arg…);                                    // key expansion

for(condition…)
{
rc = pthread_create(&thread[t], &attr, threadfunc, (void *)t); //thread create
 }
}

end = clock();                                           //time record stops
total_time = (end - start);                             //total time calculated
}
```

Figure 30: Multi threaded CPU program structure

3.3.2 Implementation on GPU.

Key points:

- This implementation is a typical case of domain decomposition or data parallel programming, which used the SIMD model.
- The plain text is divided into block of 16 bytes, which are then encrypted simultaneously. The encryption key remains same for each block. All the round keys are generated by the GPU before the encryption function is called.

- This typically the case of data decomposition or data parallel programming which uses the SIMD model.

AES-128, 192 and 256 are implemented on the Quadro K4000 NVIDIA GPU using CUDA programming. A single AES encryption process (k=1) involves execution time to:

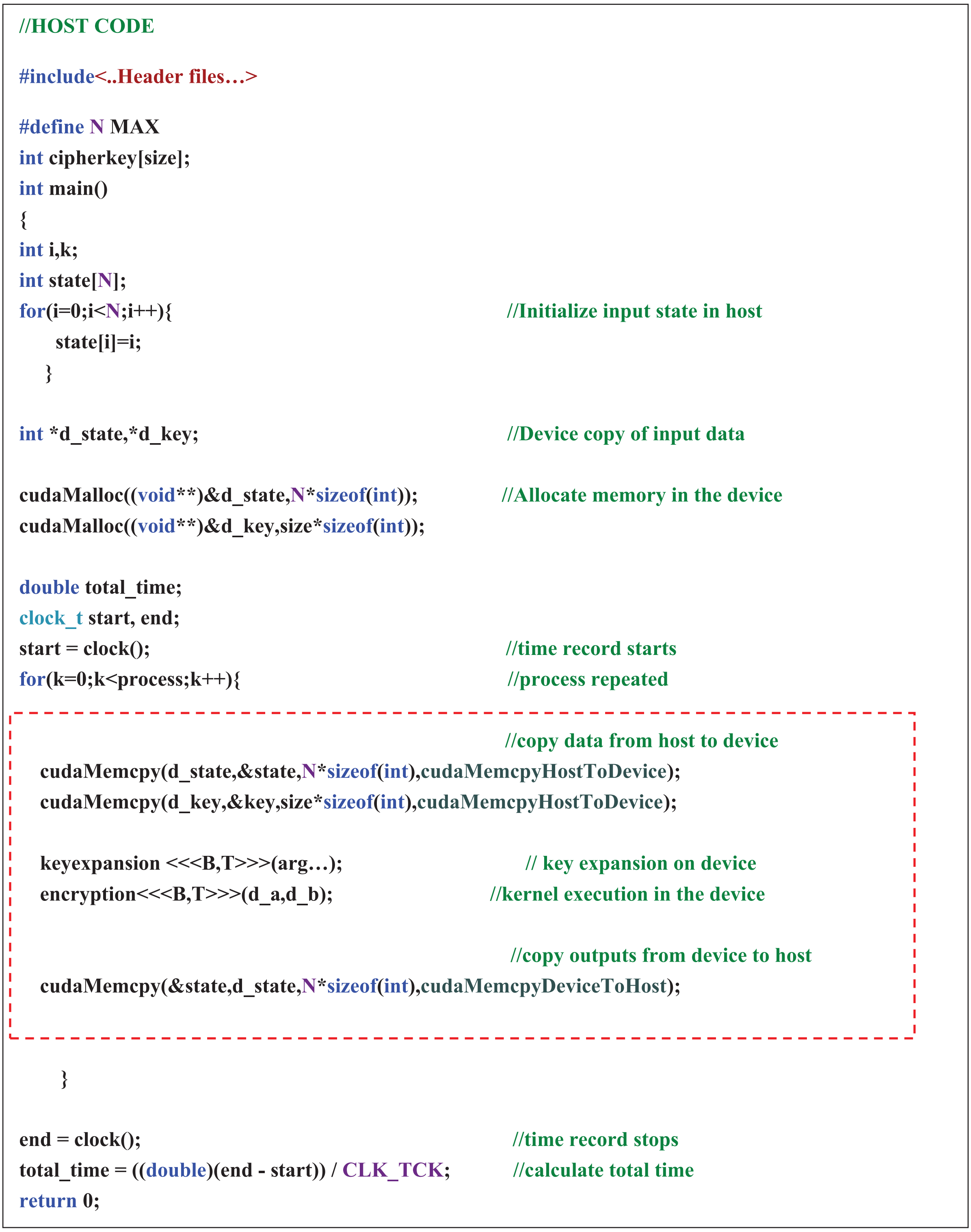
- Initialize the input date on the host (CPU).
- Copy the input data to the device.
- Execute the program with the copied input data in the device.
- Copy the output data to the host.

The program structure of the CUDA program is shown in figure 31 and figure 32. The program is structured in such a way that each thread in the GPU takes a chunk of the data and executes it parallel with other threads. The implementation is accomplished using two different data sizes 32000 bytes and 32000*5 bytes. In case of 32000 bytes, for single granularity 2000 threads are utilized so that each thread operates on data of size 16 bytes. In the case of 32000*5 bytes, the host (CPU) passes the data in chunks of 32000 bytes to the device (GPU). The GPU executes the 32000 bytes and returns the results before it gets the next chunk from the host. The process is repeated for k=10, in order to increase readability of results. 20 readings of execution time are taken in each case. Its Average, Standard Deviation and Confidence Interval are calculated.

The execution time includes:

- Time taken to copy data from host to device
- Time taken to execute the kernel (key expansion and encryption) by the device.
- Time taken to copy the results from device to host.

The structure of the CUDA program is shown below.



```
//HOST CODE

#include<..Header files…>

#define N MAX
int cipherkey[size];
int main()
{
int i,k;
int state[N];
for(i=0;i<N;i++){                                   //Initialize input state in host
      state[i]=i;
    }

int *d_state,*d_key;                                //Device copy of input data

cudaMalloc((void**)&d_state,N*sizeof(int));         //Allocate memory in the device
cudaMalloc((void**)&d_key,size*sizeof(int));

double total_time;
clock_t start, end;
start = clock();                                    //time record starts
for(k=0;k<process;k++){                             //process repeated

                                                    //copy data from host to device
   cudaMemcpy(d_state,&state,N*sizeof(int),cudaMemcpyHostToDevice);
   cudaMemcpy(d_key,&key,size*sizeof(int),cudaMemcpyHostToDevice);

   keyexpansion <<<B,T>>>(arg…);                      // key expansion on device
   encryption<<<B,T>>>(d_a,d_b);                    //kernel execution in the device


                                                    //copy outputs from device to host
   cudaMemcpy(&state,d_state,N*sizeof(int),cudaMemcpyDeviceToHost);


       }

end = clock();                                      //time record stops
total_time = ((double)(end - start)) / CLK_TCK;     //calculate total time
return 0;
```

Figure 31: CUDA programming structure (host code)

```
//DEVICE CODE

__global__ void keyexpansion (int*key,int*roundkeys…..)

__global__ void encryption (int*state, int*key,int*cipher)

{           Mixcolumn (arg...);

            Shiftrow (arg...);

            Subbyte (arg...);

            Addroundkey (arg...);

}
```

Figure 32: CUDA programming structure (device code)

The experiment is performed on different granularity levels include granularity 1, 2, 10 and 100. In granularity 1 a total of 2000 threads are utilized, where as in granularity 2, 10 and 100, number of threads utilized are 1000, 200 and 20 respectively. Each granularity level is implemented for different thread block combinations <B, T>, where B denotes the number of blocks utilized and T denotes the number of threads utilized in each block. For granularity 1 the thread block combinations experimented on includes <20, 100>, <100, 20>, <50, 40>, <40, 50>, <1000, 2>, <2, 1000> and <2000, 1>. For granularity 2 the thread block combinations includes <10, 100>, <100, 10>, <50, 20>, <20, 50>, <40, 25>, <25, 40> and <1000, 1>. For granularity 10 the thread block combinations includes <20, 10>, <10, 20>, <8, 25>, <25, 8>, <2, 100>, <100, 2> and <200, 1>. And for granularity 100 the thread block combinations includes <2, 10>, <10, 2>, <5, 4>, <4, 5>, <20, 1>, <1, 20> and <2, 10>. 20 readings are taken for each case and the Average, Standard Deviation and Confidence Interval are calculated.

Optimized CUDA using CUDA STREAMS

The AES 128, 192 and 256 algorithms are then implemented on the GPU using an optimized CUDA version using CUDA STREAMS. The algorithm is implemented for data size 32000 bytes, 32000*2 bytes and 32000* 5 bytes. This is implemented using the best thread block distribution obtained from the previous results. For data size 32000 bytes and 32000*5 bytes, two streams are used with each stream processing 16000 bytes and 32000 bytes of data respectively. For data size of 32000*5 bytes, five streams are used with each stream processing 32000 bytes of data. A single AES encryption process (k=1) involves execution time to:

- Memory copy from host to device streams
- Kernel execution by all streams.
- Memory copy from all streams of the device to the host.

The process is repeated for k=10, in order to increase readability of results. 20 readings of execution time are taken in each case. Its Average, Standard Deviation and Confidence

Interval are calculated. The structure of the CUDASTREAM program using 2 steams is shown in figure 33.

```
//HOST CODE
#include<..Header files…>
#define N MAX
void main()
{
cudaStream_t stream0,stream1;
cudaStreamCreate()
start = clock();                                        //time record starts
for(k=0;k<process;k++)                                  //process repeated
{
                                                        //copy data from host to device
  cudaMemcpyAsync(d_a0,&h_a0, N *sizeof(int),cudaMemcpyHostToDevice,stream0);
  cudaMemcpyAsync(d_a1,&h_a1, N *sizeof(int),cudaMemcpyHostToDevice,stream1);
  keyexpansion<<<1,16>>>(d_key,…)                      //key expansion in the device
  encryption<<<50,20,0,stream0>>>(d_a0,d_b);            //kernel execution in the device
  encryption<<<50,20,0,stream1>>>(d_a1,d_b);
                                                        //copy data from host to device
  cudaMemcpyAsync(&h_a0,d_a0, N *sizeof(int),cudaMemcpyDeviceToHost,stream0);
  cudaMemcpyAsync(&h_a1,d_a1, N *sizeof(int),cudaMemcpyDeviceToHost,stream1);
}
end = clock();                                          //time record stops
total_time = ((double)(end - start)) / CLK_TCK;         //calculate total time
}
```

Figure 33: CUDA STREAMS programming structure (host code)

CHAPTER 4 RESULTS

With the advancement in the technology and the role of communication, huge amount of data needs to be transferred while maintaining high level of privacy. Large amounts of data are continuously encrypted every second. Speed of encryption has become a major concern in the modern era of data protection. The degradation in the possibility of advancement in the hardware of a system demands for re-programming the software to achieve higher performance. Parallel computing is the preferred solution adopted by most developers to design an algorithm that can be processed faster in parallel computers. Understanding the different ways of parallelising a solution would help to choose the best solution in any application. This research work examines the performance of encryption algorithm on different levels of parallelism on a CPU and GPU. This section illustrates the acquired results in this research.

4.1 IMPLEMENTATION ON SINGLE CORE CPU

A single threaded C program is developed for AES-128, AES-192 and AES-256 encryption algorithms. Data is processed sequentially in chunks of 16 bytes each. 20 readings are taken in each case for better accuracy. Readings are taken for a single process run (k=1) and for 10 runs (k=10). The purpose of executing the process 10 times is to acquire better readability of data making it easy for comparison. The readings for k=10 are ~ 10 times of readings for k =1, which is accurate. Results for k=10, is shown in APPENDIX A. The implementation is carried out in two cases for each AES algorithm as described below. Table 1 – table 3 shows the execution time single process (k=1) of AES 128/192/526 encryption algorithms using Single threaded C program on the CPU.

4.1.1 AES 128 on Single threaded C

AES 128 algorithm operates on an input state of 16 bytes or 128 bits and a cipher key of size 16 bytes. It contains 10 transformation rounds as described in chapter 2. Each round contains a set of transformation functions.

Case 1:	*Case 2:*
Input data size= 32000 bytes	Input data size=32000*5
Key size= 16 bytes.	Key size =16 bytes

The table 1 below shows the average execution time, standard deviation and confidence interval of the values acquired during a single process. The unit of measurements is in “seconds”.

Single threaded C for AES 128		
	Data Size= 32000 bytes	Data Size= 32000*5 bytes
Average	0.0325	0.1495
Standard Deviation	0.00473	0.010247
Confidence Interval	0.030427 - 0.034573	0.145009 - 0.153991

Table 1: Execution time for single threaded C for AES-128

It can be seen that AES 128 takes ~ 0.032 seconds to execute a single process run for a data size of 32000 bytes long. It takes ~ 0.15 seconds to execute the data of size 32000*5 bytes which is ~ 5 times higher than the former case.

4.1.2 AES 192 on Single threaded C

AES 192 algorithm operates on an input state of 16 bytes or 128 bits and a cipher key of size 24 bytes. It contains 12 transformation rounds as described in chapter 2. Each round contains a set of transformation functions.

Case 1:

Input data size= 32000 bytes
Key size= 24 bytes.

Case 2:

Input data size=32000*5
Key size =24 bytes

Table 2 shows the execution time of AES 192 encryption algorithm using Single threaded C program on the CPU. It shows the average execution time, standard deviation and confidence interval of the values acquired during a single process. The unit of measurements is in “seconds”.

Single threaded C for AES 192		
	Data Size= 32000 bytes	Data Size= 32000*5 bytes
Average	0.037	0.1753
Standard Deviation	0.005231	0.017448
Confidence Interval	0.034707 - 0.039293	0.167653 - 0.182947

Table 2: Execution time for Single threaded C for AES-192

It can be seen that AES 192 takes ~ 0.04 seconds to execute a single process run for a data size of 32000 bytes long. It takes ~ 0.2 seconds to execute the data of size 32000*5 bytes which is ~ 5 times slower than the former case.

4.1.3 AES 256 on Single threaded C

AES 256 algorithm operates on an input state of 16bytes or 128 bits and a cipher key of size 32 bytes. It contains 14 transformation rounds as described in chapter 2. Each round contains a set of transformation functions.

Case 1:	***Case 2:***
Input data size= 32000 bytes	Input data size=32000*5
Key size= 32 bytes.	Key size =32 bytes

Table 3 shows the execution time of AES 256 encryption algorithm using Single threaded C program on the CPU. It shows the average execution time, standard deviation and confidence interval of the values acquired during the process. The unit of measurements is in "seconds".

Single threaded C for AES 256		
	Data Size= 32000 bytes	Data Size= 32000*5 bytes
Average	0.0521	0.2067
Standard Deviation	0.007711	0.014942
Confidence Interval	0.04872 - 0.05548	0.200151 – 0.213249

Table 3: Execution time for Single threaded C for AES-256

It can be seen that AES 256 takes ~ 0.05 seconds to execute a single process run for a data size of 32000 bytes long. It takes ~ 0.2 seconds to execute the data of size 32000*5 bytes which is ~ 4 times slower than the former case.

4.2 IMPLEMENTATION ON GPU USING CUDA

The encryption algorithms AES-128, AES-192 and AES-256 are implemented on NVIDIA GPU (Quadro K 4000). The implementation is carried out with two different data sizes. In each case various combinations of thread and block distributions are tested to see the effect on the algorithm execution and to identify the best grid distribution that can provide least execution time. The algorithm with each thread-block distribution is tested with different granularity levels (Granularity 1, 2, 10 and 100). The tables 4 – table 27 shows the execution time of AES encryption on a GPU using CUDA programming. 20 readings are taken for each combination to maintain accuracy. Readings are taken for a single process run (k=1) and for 10 runs (k=10). The purpose of executing the process 10 times is to acquire better readability of data making it easier for comparison. The tables below show the average execution time, standard deviation and confidence interval of the values acquired during the process. The unit of measurement is in "seconds".

4.2.1 AES 128 on GPU

Case 1:

Input data size = 32000 bytes
Key size= 16 bytes

- Granularity 1:

Total number of threads utilized = 2000.

	AES 128 on GPU using CUDA Data Size = 32000 Granularity 1						
Grid Dimension	**<2, 1000>**	**<20, 100>**	**<40,50>**	**<50,40>**	**<100,20>**	**<1000,2>**	**<2000,1>**
Average	0.00225	0.002	0.00175	0.00275	0.00325	0.0155	0.048
Standard Deviation	0.002394	0.002481	0.002415	0.002519	0.00215	0.001519	0.070208
Confidence Interval	0.001508 - 0.002992	0.001231 - 0.002769	0.001002 - 0.002498	0.001969 - 0.003531	0.002502 - 0.003998	0.015029 - 0.015971	0.026243 - 0.069757

Table 4: Execution time for CUDA AES-128 with granularity 1 for 32000 bytes data

Table 4 shows the execution time of AES 128 on GPU with granularity 1. For a data size of 32000 bytes, 2000 threads are utilized where each thread handles 16 bytes of data. In other words, every thread independently runs the AES algorithm on different parts of the data. It can be observed that the fastest execution time obtained is ~0.0017 seconds which is about 18 times faster than the single threaded C program on the CPU.

- Granularity 2:

Total number of threads utilized = 1000.

Table 5 shows the execution time of AES 128 on GPU with granularity 2. For a data size of 32000 bytes 1000 threads are utilized where each thread handles 32 bytes of data. The 32 bytes of data is internally divided into chunks and executed by the single thread sequentially. In other words, every thread independently runs the AES algorithm on different parts of the data. It can be observed that the fastest execution time obtained is ~0.0015 seconds which is about 22 times faster than the single threaded C program on the CPU.

	AES 128 on GPU using CUDA Data Size = 32000 Granularity 2						
Grid Dimension	**<10,100>**	**<20,50>**	**<25, 40>>**	**<40,25>**	**<50, 20>**	**<100,10>**	**<1000,1>**
Average	0.002	0.00225	0.0015	0.00175	0.00275	0.00425	0.0275
Standard Deviation	0.002513	0.002552	0.002351	0.002314	0.002552	0.001832	0.002565
Confidence Interval	0.000899 - 0.003101	0.001132 - 0.003368	0.00047 - 0.00253	0.000736 - 0.002764	0.001632 - 0.003868	0.003447 - 0.005053	0.026376 - 0.028624

Table 5: Execution time CUDA AES-128 with granularity 2 for 32000 bytes input

- Granularity 10:

Total number of threads utilized = 200.

	AES 128 on GPU using CUDA Data Size = 32000 Granularity 10						
Grid Dimension	**<2,100>**	**<8,25>**	**<10,20>**	**<20,10>**	**<25,8>**	**<100,2>**	**<200,1>**
Average	0.0065	0.00655	0.00775	0.0085	0.009	0.01525	0.02875
Standard Deviation	0.002351	0.002438	0.002552	0.002351	0.002351	0.001118	0.002221
Confidence Interval	0.005470 - 0.007530	0.005481 - 0.007619	0.006632 - 0.008868	0.007470 - 0.009530	0.008101 - 0.009899	0.014760 - 0.015740	0.027776 - 0.029724

Table 6: Execution time CUDA AES-128 with granularity 10 for 32000 bytes input

Table 6 shows the execution time of AES 128 on GPU with granularity 10. For a data size of 32000 bytes 200 threads are utilized where each thread handles 160 bytes of data. The 160 bytes of data is internally divided into chunks and executed by the single thread sequentially. In other words, every thread independently runs the AES algorithm on different parts of the data. It can be observed that the fastest execution time obtained is ~0.0065 seconds which is about 5 times faster than the single threaded C program on the CPU.

- Granularity 100:

Total number of threads utilized = 20.

	AES 128 on GPU using CUDA Data Size = 32000 Granularity 100					
Grid Dimension	**<1,20>**	**<2,10>**	**<4,5>**	**<5, 4>**	**<10, 2>**	**<20,1>**
Average	0.05875	0.05775	0.05675	0.058	0.05875	0.06125
Standard Deviation	0.002221	0.002552	0.002447	0.002384	0.002221	0.002221
Confidence Interval	0.057776 - 0.059724	0.056632 - 0.058868	0.055678 - 0.057822	0.056955 - 0.059045	0.057776 - 0.059724	0.060276 - 0.062224

Table 7: Execution time CUDA AES-128 with granularity 100 for 32000 bytes input

Table 7 shows the execution time of AES 128 on GPU with granularity 100. For a data size of 32000 bytes 20 threads are utilized where each thread handles 1600 bytes of data. The 1600 bytes of data is internally divided into chunks and executed by the single thread sequentially. In other words, every thread independently runs the AES algorithm on different parts of the data. It can be observed that the fastest execution time obtained is ~0.056 seconds which is about twice higher than the execution time of single threaded C program on the CPU.

Case 2:

Input data size = 32000*5 bytes
Key size= 16 bytes

In this case, the total data size is 32000*5 bytes. The data is divided into chunks of 32000 bytes each and processed on the Graphic card sequentially. Each process in turn divides the 32000 byte chunks into smaller chunks and processes it in parallel

- Granularity 1

Total number of threads utilized = 2000.

	AES 128 on GPU using CUDA Data Size = 32000*5 Granularity 1						
Grid Dimension	**<2, 1000>**	**<20, 100>**	**<40,50>**	**<50,40>**	**<100,20>**	**<1000,2>**	**<2000,1>**
Average	0.01675	0.014	0.014	0.0155	0.0195	0.07325	0.1565
Standard Deviation	0.002447	0.002052	0.002615	0.002236	0.002236	0.023411	0.002351
Confidence Interval	0.015678 - 0.017822	0.013101 - 0.014899	0.012853 - 0.0151463	0.014520 - 0.016480	0.018520 - 0.020480	0.062990 - 0.083510	0.155470 - 0.155470

Table 8: Execution time CUDA AES-128 with granularity 1 for 32000*5 bytes input

Table 8 shows the execution time of AES 128 on GPU with granularity 1. The total data size is 32000*5 bytes which is processed on the graphic card in chunks of size 32000 bytes each. For a chunk of data of size 32000 bytes a total of 2000 threads are utilized where each thread handles 16 bytes of data. In other words, every thread independently runs the AES algorithm on different parts of the data chunk. It can be observed that the fastest execution time obtained here is ~0.014 seconds which is about 11 times faster than the single threaded C program on the CPU.

- Granularity 2:

Total number of threads utilized = 1000.

	AES 128 on GPU using CUDA Data Size = 32000*5 Granularity 2						
Grid Dimension	**<10,100>**	**<20,50>**	**<25, 40>>**	**<40,25>**	**<50, 20>**	**<100,10>**	**<1000,1>**
Average	0.0125	0.013	0.01345	0.013	0.01175	0.021	0.14075
Standard Deviation	0.002565	0.002513	0.002235	0.002991	0.002447	0.002052	0.001832
Confidence Interval	0.011376 - 0.013624	0.011898 - 0.014101	0.012470 - 0.014430	0.011689 - 0.014311	0.010678 - 0.012822	0.020101 - 0.021899	0.139947 - 0.141553

Table 9: Execution time CUDA AES-128 with granularity 2 for 32000*5 bytes input

Table 9 shows the execution time of AES 128 on GPU with granularity 2. The total data size is 32000*5 bytes which is processed on the graphic card in chunks of size 32000 bytes each. For a chunk of data of size 32000 bytes a total of 1000 threads are utilized where each thread handles 32 bytes of data. This data is internally divided into yet smaller chunks and executed by single thread sequentially. It can be observed that the fastest execution time obtained is ~0.0117 seconds which is about 13 times faster than the single threaded C program on the CPU.

- Granularity 10:

Total number of threads utilized = 200.

	AES 128 on GPU using CUDA Data Size = 32000*5 Granularity 10						
Grid Dimension	**<2,100>**	**<8,25>**	**<10,20>**	**<20,10>**	**<25,8>**	**<100,2>**	**<200,1>**
Average	0.038	0.0375	0.036	0.03935	0.0415	0.078	0.14725
Standard Deviation	0.002513	0.002565	0.002052	0.001565	0.002856203	0.002513	0.002552
Confidence Interval	0.036899 - 0.039101	0.036376 - 0.038624	0.035101 - 0.036899	0.038664 - 0.040036	0.040248 - 0.042751	0.076899 - 0.079101	0.146132 - 0.148368

Table 10: Execution time CUDA AES-128 with granularity 10 for 32000*5 bytes input

The total data size is 32000*5 bytes which is processed on the graphic card in chunks of size 32000 bytes each. For a chunk of data of size 32000 bytes a total of 200 threads are utilized where each thread handles 160 bytes of data. This data is internally divided into yet smaller chunks and executed by single thread sequentially. It can be observed in table 10 that the fastest execution time obtained is ~0.036 seconds which is about 4 times faster than the single threaded C program on the CPU.

- Granularity 100:

Total number of threads utilized = 20.

Table 11 shows the execution time of AES 128 on GPU with granularity 100 for a data size is 32000*5 bytes which is processed on the graphic card in chunks of size 32000 bytes each. For a chunk of data of size 32000 bytes a total of 20 threads are utilized where each thread handles 1600 bytes of data. This data is internally divided into yet smaller chunks and executed by single thread sequentially. It can be observed that the fastest execution time obtained is ~0.285 seconds which is about twice higher than the execution time of single threaded C program on the CPU.

	AES 128 on GPU using CUDA Data Size = 32000*5 Granularity 100					
Grid Dimension	**<1,20>**	**<2,10>**	**<4,5>**	**<5, 4>**	**<10, 2>**	**<20,1>**
Average	0.295	0.288	0.285	0.288	0.28525	0.3095
Standard Deviation	0	0.002991	5.69532E-17	0.002513	0.001118	0.00484
Confidence Interval	0.295000 - 0.295000	0.286689 - 0.289311	0.285000 - 0.285000	0.286899 - 0.289101	0.284760 - 0.285740	0.307379 - 0.311621

Table 11: Execution time CUDA AES-128 with granularity 100 for 32000*5 bytes input

4.2.2 AES 192 on GPU

Case 1:

Input data size = 32000 bytes
Key size= 24 bytes

- Granularity 1:

Total number of threads utilized = 2000.

	AES 192 on GPU using CUDA Data Size = 32000 Granularity 1						
Grid Dimension	**<2,1000>**	**<20,100>**	**<40,50>**	**<50,40>**	**<100,20>**	**<1000,2>**	**<2000,1>**
Average	0.004250	0.004000	0.003500	0.004750	0.005250	0.021000	0.041000
Standard Deviation	0.001832	0.002052	0.002350	0.001118	0.001118	0.002052	0.002052
Confidence Interval	0.003447 - 0.005053	0.003101 - 0.004899	0.002469 - 0.004530	0.004260 - 0.005239	0.004760 - 0.005739	0.020101 - 0.021899	0.040101 - 0.041899

Table 12: Execution time CUDA AES-192 with granularity 1 for 32000 bytes input

Table 12 shows the execution time of AES 192 on GPU with granularity 1. For a data size of 32000 bytes 2000 threads are utilized where each thread handles 16 bytes of data. In other words, every thread independently runs the AES algorithm on different parts of the data. It can be observed that the fastest execution time obtained is ~0.003 seconds which is about 10 times faster than the single threaded C program on the CPU.

- Granularity 2:

Total number of threads utilized = 1000.

Table 13 shows the execution time of AES 192 on GPU with granularity 2. For a data size of 32000 bytes 1000 threads are utilized where each thread handles 32 bytes of data. The 32 bytes of data is internally divided into chunks and executed by the single thread sequentially. In other words, every thread independently runs the AES algorithm on different parts of the data. It can be observed that the fastest execution time obtained is ~0.0017 seconds which is about 21 times faster than the single threaded C program on the CPU.

	AES 192 on GPU using CUDA Data Size = 32000 Granularity 2						
Grid Dimension	**<10,100>**	**<20,50>**	**<25, 40>**	**<40, 25>**	**<50, 20>**	**<100, 10>**	**<1000,1>**
Average	0.002500	0.002000	0.001750	0.004000	0.003500	0.005000	0.035250
Standard Deviation	0.002565	0.002513	0.002447	0.002052	0.002350	8.898E-19	0.001118
Confidence Interval	0.001376 - 0.003624	0.000898 - 0.003101	0.000678 - 0.002822	0.003101 - 0.004899	0.002469 - 0.004530	0.005000 - 0.005000	0.034760 - 0.03574

Table 13: Execution time CUDA AES-192 with granularity 2 for 32000 bytes input

- Granularity 10:

Total number of threads utilized = 200.

	AES 192 on GPU using CUDA Data Size = 32000 Granularity 10						
Grid Dimension	**<2, 100>**	**<8, 25>**	**<10, 20>**	**<20, 10>**	**<25, 8>**	**<100, 2>**	**<200,1>**
Average	0.009750	0.009000	0.009500	0.010000	0.010250	0.020750	0.036500
Standard Deviation	0.001118	0.002051	0.001538	1.779E-18	0.001118	0.002447	0.002351
Confidence Interval	0.009260 - 0.010240	0.008100 - 0.009899	0.008825 - 0.010174	0.010000 - 0.010000	0.009760 - 0.010740	0.019678 - 0.021822	0.035470 - 0.037530

Table 14: Execution time CUDA AES-192 with granularity 10 for 32000 bytes input

Table 14 shows the execution time of AES 128 on GPU with granularity 10. For a data size of 32000 bytes 200 threads are utilized where each thread handles 160 bytes of data. The 160 bytes of data is internally divided into chunks and executed by the single thread sequentially. In other words, every thread independently runs the AES algorithm on different parts of the data. It can be observed that the fastest execution time obtained is ~0.009 seconds which is about 4 times faster than the single threaded C program on the CPU.

- Granularity 100:

Total number of threads utilized = 20.

	AES 192 on GPU using CUDA Data Size = 32000 Granularity 100					
Grid Dimension	**<1, 20>**	**<2, 10>**	**<4, 5>**	**<5, 4>**	**<10, 2>**	**<20, 1>**
Average	0.071500	0.070250	0.069500	0.070000	0.070500	0.076250
Standard Deviation	0.002351	0.001118	0.001539	2.8476E-17	0.001538	0.002221
Confidence Interval	0.070470 - 0.072530	0.069760 - 0.070739	0.068826 - 0.070174	0.070000 - 0.070000	0.069825 - 0.071174	0.075276 - 0.077224

Table 15: Execution time CUDA AES-192 with granularity 100 for 32000 bytes input

Table 15 shows the execution time of AES 192 on GPU with granularity 100. For a data size of 32000 bytes 20 threads are utilized where each thread handles 1600 bytes of data. The 1600 bytes of data is internally divided into chunks and executed by the single thread sequentially. In other words, every thread independently runs the AES algorithm on different parts of the data. It can be observed that the fastest execution time obtained is ~0.069 seconds which is about 2 times higher than the execution time of single threaded C program on the CPU.

Case 2:

Input data size = 32000*5 bytes
Key size= 24 bytes

In this case, the total data size is 32000*5 bytes. The data is divided into chunks of 32000 bytes each and processed on the Graphic card sequentially. Each process in turn divides the 32000 byte chunks into smaller chunks and processes it in parallel

- Granularity 1

Total number of threads utilized = 2000.

	AES 192 on GPU using CUDA Data Size = 32000*5 Granularity 1						
Grid Dimension	**<2,1000>**	**<20,100>**	**<40,50>**	**<50,40>**	**<100,20>**	**<1000,2>**	**<2000,1>**
Average	0.02075	0.0165	0.01725	0.01925	0.02475	0.10625	0.2035
Standard Deviation	0.001832	0.002351	0.002552	0.001832	0.001118	0.002221	0.002351
Confidence Interval	0.019947 - 0.021553	0.01547 - 0.01753	0.016132 - 0.028368	0.018447 - 0.020053	0.02426 - 0.02524	0.105276 - 0.107224	0.20247 - 0.204530

Table 16: Execution time CUDA AES-192 with granularity 1 for 32000*5 bytes input

Table 16 shows the execution time of AES 192 on GPU with granularity 1. The total data size is 32000*5 bytes which is processed on the graphic card in chunks of size 32000 bytes each. For a chunk of data of size 32000 bytes a total of 2000 threads are utilized where each thread handles 16 bytes of data. In other words, every thread independently runs the AES algorithm on different parts of the data chunk. It can be observed that the fastest execution time obtained here is ~0.0165 seconds which is about 11 times faster than the single threaded C program on the CPU.

- Granularity 2:

Total number of threads utilized = 1000.

	AES 192 on GPU using CUDA Data Size = 32000*5 Granularity 2						
Grid Dimension	**<10,100>**	**<20,50>**	**<25, 40>**	**<40, 25>**	**<50, 20>**	**<100, 10>**	**<1000,1>**
Average	0.014500	0.014250	0.014550	0.014570	0.015500	0.026500	0.175250
Standard Deviation	0.001539	0.001832	0.001118	0.001539	0.001539	0.002351	0.001118
Confidence Interval	0.013826 - 0.015174	0.013447 - 0.015053	0.014260 - 0.015240	0.013826 - 0.015174	0.014826 - 0.016174	0.02547 - 0.027530	0.174760 - 0.175740

Table 17: Execution time CUDA AES-192 with granularity 2 for 32000*5 bytes input

Table 17 shows the execution time of AES 192 on GPU with granularity 2. The total data size is 32000*5 bytes which is processed on the graphic card in chunks of size 32000 bytes each. For a chunk of data of size 32000 bytes a total of 1000 threads are utilized where each thread handles 32 bytes of data. This data is internally divided into yet smaller chunks and executed by single thread sequentially. It can be observed that the fastest execution time obtained is ~0.014 seconds which is about 12 times faster than the single threaded C program on the CPU.

- Granularity 10:

Total number of threads utilized = 200.

	AES 192 on GPU using CUDA Data Size = 32000*5 Granularity 10						
Grid Dimension	**<2, 100>**	**<8, 25>**	**<10, 20>**	**<20, 10>**	**<25, 8>**	**<100, 2>**	**<200,1>**
Average	0.044250	0.044250	0.044750	0.047500	0.049250	0.095250	0.181250
Standard Deviation	0.002447	0.001832	0.002552	0.002565	0.001832	0.001970	0.002221
Confidence Interval	0.043178 - 0.045322	0.043447 - 0.045053	0.043632 - 0.045868	0.046376 - 0.048624	0.048447 - 0.050053	0.094387 - 0.096113	0.180276 - 0.182224

Table 18: Execution time CUDA AES-192 with granularity 10 for 32000*5 bytes input

Table 18 shows the execution time of AES 192 on GPU with granularity 10. The total data size is 32000*5 bytes which is processed on the graphic card in chunks of size 32000 bytes each. For a chunk of data of size 32000 bytes a total of 200 threads are utilized where each thread handles 160 bytes of data. This data is internally divided into yet smaller chunks and executed by single thread sequentially. It can be observed that the fastest execution time obtained is ~0.044 seconds which is about 4 times faster than the single threaded C program on the CPU.

- Granularity 100:

Total number of threads utilized = 20.

	AES 192 on GPU using CUDA Data Size = 32000*5 Granularity 100					
Grid Dimension	**<1, 20>**	**<2, 10>**	**<4, 5>**	**<5, 4>**	**<10, 2>**	**<20, 1>**
Average	0.35575	0.3495	0.348	0.348	0.351	0.37525
Standard Deviation	0.001832	0.001539	0.002513	0.002513	0.002052	0.00197
Confidence Interval	0.354947 - 0.356553	0.348826 - 0.350174	0.346899 - 0.349101	0.346899 - 0.349101	0.350101 - 0.351899	0.374387 - 0.376113

Table 19: Execution time CUDA AES-192 with granularity 100 for 32000*5 bytes input

Table 19 shows the execution time of AES 192 on GPU with granularity 100. The total data size is 32000*5 bytes which is processed on the graphic card in chunks of size 32000 bytes each. For a chunk of data of size 32000 bytes a total of 20 threads are utilized where each thread handles 1600 bytes of data. This data is internally divided into yet smaller chunks and executed by single thread sequentially. It can be observed that the fastest execution time obtained is ~0.348 seconds which is about twice higher than the execution time of single threaded C program on the CPU.

4.2.3 AES 256 on GPU

Case 1:

Input data size = 32000 bytes
Key size= 32 bytes

- Granularity 1:

Total number of threads utilized = 2000.

	AES 256 on GPU using CUDA Data Size = 32000 Granularity 1						
Grid Dimension	**<2,1000>**	**<20,100>**	**<40,50>**	**<50,40>**	**<100,20>**	**<1000,2>**	**<2000,1>**
Average	0.00525	0.0055	0.004	0.00425	0.00575	0.024	0.04775
Standard Deviation	0.001118	0.001539	0.002052	0.001832	0.001832	0.002052	0.002552
Confidence Interval	0.00476 - 0.005740	0.004826 - 0.006174	0.003101 - 0.004899	0.003447 - 0.005053	0.004947 - 0.006553	0.023101 - 0.024899	0.046632 - 0.048868

Table 20: Execution time CUDA AES-256 with granularity 1 for 32000 bytes input

Table 20 shows the execution time of AES 256 on GPU with granularity 1. For a data size of 32000 bytes 2000 threads are utilized where each thread handles 16 bytes of data. In other words, every thread independently runs the AES algorithm on different parts of the data. It can be observed that the fastest execution time obtained is ~0.004 seconds which is about 13 times faster than the single threaded C program on the CPU.

- Granularity 2:

Total number of threads utilized = 1000.

	AES 256 on GPU using CUDA Data Size = 32000 Granularity 2						
Grid Dimension	**<10,100>**	**<20,50>**	**<25, 40>**	**<40, 25>**	**<50, 20>**	**<100, 10>**	**<1000,1>**
Average	0.00375	0.00375	0.0035	0.00475	0.004	0.0065	0.04025
Standard Deviation	0.002221	0.002221	0.002351	0.001118	0.002052	0.002351	0.001118
Confidence Interval	0.002776 - 0.004724	0.002776 - 0.004724	0.00247 - 0.004530	0.00426 - 0.005240	0.003101 - 0.004899	0.00547 - 0.007530	0.03976 - 0.040740

Table 21: Execution time CUDA AES-256 with granularity 2 for 32000 bytes input

Table 21 shows the execution time of AES 256 on GPU with granularity 2. For a data size of 32000 bytes 1000 threads are utilized where each thread handles 32 bytes of data. The 32 bytes of data is internally divided into chunks and executed by the single thread sequentially. In other words, every thread independently runs the AES algorithm on different parts of the data. It can be observed that the fastest execution time obtained is ~0.0035 seconds which is about 12 times faster than the single threaded C program on the CPU.

- Granularity 10:

Total number of threads utilized = 200.

	AES 256 on GPU using CUDA Data Size = 32000 Granularity 10						
Grid Dimension	**<2, 100>**	**<8, 25>**	**<10, 20>**	**<20, 10>**	**<25, 8>**	**<100, 2>**	**<200,1>**
Average	0.01175	0.0105	0.0113	0.01125	0.01175	0.02125	0.04175
Standard Deviation	0.002447	0.001539	0.002203	0.002221	0.002447	0.002221	0.002447
Confidence Interval	0.010678 - 0.012822	0.009826 - 0.011174	0.010335 - 0.012265	0.010276 - 0.012224	0.010678 - 0.012822	0.020276 - 0.022224	0.040678 - 0.042822

Table 22: Execution time CUDA AES-256 with granularity 10 for 32000 bytes input

Table 22 shows the execution time of AES 256 on GPU with granularity 10. For a data size of 32000 bytes 200 threads are utilized where each thread handles 160 bytes of data. The 160 bytes of data is internally divided into chunks and executed by the single thread sequentially. In other words, every thread independently runs the AES algorithm on different parts of the data. It can be observed that the fastest execution time obtained is ~0.0105 seconds which is about 4 times faster than the single threaded C program on the CPU.

- Granularity 100:

Total number of threads utilized = 20.

	AES 256 on GPU using CUDA Data Size = 32000 Granularity 100					
Grid Dimension	**<1, 20>**	**<2, 10>**	**<4, 5>**	**<5, 4>**	**<10, 2>**	**<20, 1>**
Average	0.08175	0.081250	0.0808	0.08175	0.083000	0.0875
Standard Deviation	0.002447	0.002221	0.001824	0.002447	0.002513	0.002565
Confidence Interval	0.080678 - 0.082822	0.080276 - 0.082224	0.080001 - 0.081599	0.080678 - 0.082822	0.081899 - 0.084101	0.086376 - 0.088624

Table 23: Execution time CUDA AES-256 with granularity 100 for 32000 bytes input

Table 23 shows the execution time of AES 192 on GPU with granularity 100. For a data size of 32000 bytes 20 threads are utilized where each thread handles 1600 bytes of data. The 1600 bytes of data is internally divided into chunks and executed by the single thread sequentially. In other words, every thread independently runs the AES algorithm on different parts of the data. It can be observed that the fastest execution time obtained is ~0.08 seconds which is approximately 2 times higher than the execution time of single threaded C program on the CPU.

Case 2:

Input data size = 32000*5 bytes
Key size= 32 bytes

In this case, the total data size is 32000*5 bytes. The data is divided into chunks of 32000 bytes each and processed on the Graphic card sequentially. Each process in turn divides the 32000 byte chunks into smaller chunks and processes it in parallel

- Granularity 1

Total number of threads utilized = 2000.

	AES 256 on GPU using CUDA Data Size = 32000*5 Granularity 1						
Grid Dimension	**<2,1000>**	**<20,100>**	**<40,50>**	**<50,40>**	**<100,20>**	**<1000,2>**	**<2000,1>**
Average	0.02325	0.019	0.01875	0.021	0.02775	0.12125	0.23275
Standard Deviation	0.002936	0.002052	0.002221	0.002616	0.002552	0.007232	0.002552
Confidence Interval	0.021963 - 0.024537	0.018101 - 0.019899	0.017776 - 0.019724	0.019854 - 0.022146	0.026632 - 0.028868	0.118080 - 0.124420	0.231632 - 0.233868

Table 24: Execution time CUDA AES-256 with granularity 1 for 32000*5 bytes input

Table 24 shows the execution time of AES 256 on GPU with granularity 1. The total data size is 32000*5 bytes which is processed on the graphic card in chunks of size 32000 bytes each. For a chunk of data of size 32000 bytes a total of 2000 threads are utilized where each thread handles 16 bytes of data. In other words, every thread independently runs the AES algorithm on different parts of the data chunk. It can be observed that the fastest execution time obtained here is ~0.018 seconds which is about 11 times faster than the single threaded C program on the CPU.

- Granularity 2:

Total number of threads utilized = 1000.

	AES 256 on GPU using CUDA Data Size = 32000*5 Granularity 2						
Grid Dimension	**<10,100>**	**<20,50>**	**<25, 40>**	**<40, 25>**	**<50, 20>**	**<100, 10>**	**<1000,1>**
Average	0.01775	0.01675	0.017	0.01775	0.017	0.02975	0.202
Standard Deviation	0.002552	0.002447	0.002513	0.002552	0.002513	0.001118	0.002513
Confidence Interval	0.016632 - 0.018868	0.015678 - 0.017822	0.015899 - 0.018101	0.016632 - 0.018868	0.015899 - 0.018101	0.02926 - 0.03024	0.200899 - 0.203101

Table 25: Execution time CUDA AES-256 with granularity 2 for 32000*5 bytes input

Table 25 shows the execution time of AES 256 on GPU with granularity 2. The total data size is 32000*5 bytes which is processed on the graphic card in chunks of size 32000 bytes each. For a chunk of data of size 32000 bytes a total of 1000 threads are utilized where each thread handles 32 bytes of data. This data is internally divided into yet smaller chunks and executed by single thread sequentially. It can be observed that the fastest execution time obtained is ~0.0167 seconds which is about 12 times faster than the single threaded C program on the CPU.

- Granularity 10:

Total number of threads utilized = 200.

	AES 256 on GPU using CUDA Data Size = 32000*5 Granularity 10						
Grid Dimension	**<2, 100>**	**<8, 25>**	**<10, 20>**	**<20, 10>**	**<25, 8>**	**<100, 2>**	**<200,1>**
Average	0.05025	0.05	0.05	0.05375	0.05725	0.10875	0.20725
Standard Deviation	0.00197	7.12E-18	0.001622	0.002221	0.002552	0.002751	0.002552
Confidence Interval	0.049387 - 0.051113	0.050000 - 0.050000	0.049289 - 0.050711	0.052776 - 0.054724	0.056132 - 0.058368	0.107545 - 0.109955	0.206132 - 0.208368

Table 26: Execution time CUDA AES-256 with granularity 10 for 32000*5 bytes input

Table 26 shows the execution time of AES 256 on GPU with granularity 10. The total data size is 32000*5 bytes which is processed on the graphic card in chunks of size 32000 bytes each. For a chunk of data of size 32000 bytes a total of 200 threads are utilized where each thread handles 160 bytes of data. This data is internally divided into yet smaller chunks and executed by single thread sequentially. It can be observed that the fastest execution time obtained is ~0.05 seconds which is about 4 times faster than the single threaded C program on the CPU.

- Granularity 100:

Total number of threads utilized = 20.

	AES 256 on GPU using CUDA Data Size = 32000*5 Granularity 100					
Grid Dimension	**<1, 20>**	**<2, 10>**	**<4, 5>**	**<5, 4>**	**<10, 2>**	**<20, 1>**
Average	0.40925	0.40375	0.4025	0.4075	0.4065	0.43325
Standard Deviation	0.001832	0.002221	0.002565	0.002565	0.002351	0.002447
Confidence Interval	0.408447 - 0.410053	0.402776 - 0.404724	0.401376 - 0.403624	0.406376 - 0.408624	0.40547 - 0.40753	0.432178 - 0.434322

Table 27: Execution time CUDA AES-256 with granularity 100 for 32000*5 bytes input

Table 27 shows the execution time of AES 256 on GPU with granularity 100. The total data size is 32000*5 bytes which is processed on the graphic card in chunks of size 32000 bytes each. For a chunk of data of size 32000 bytes a total of 20 threads are utilized where each thread handles 1600 bytes of data. This data is internally divided into yet smaller chunks and executed by single thread sequentially. It can be observed that the fastest execution time obtained is ~0.402 seconds which is about 2 times higher than the execution time of single threaded C program on the CPU.

4.3 COMPARISON OF DIFFERENT GRANULARITY LEVELS ON GPU

This section presents the variation in execution time for different granularity levels for AES-128, AES-192 and AES-256.

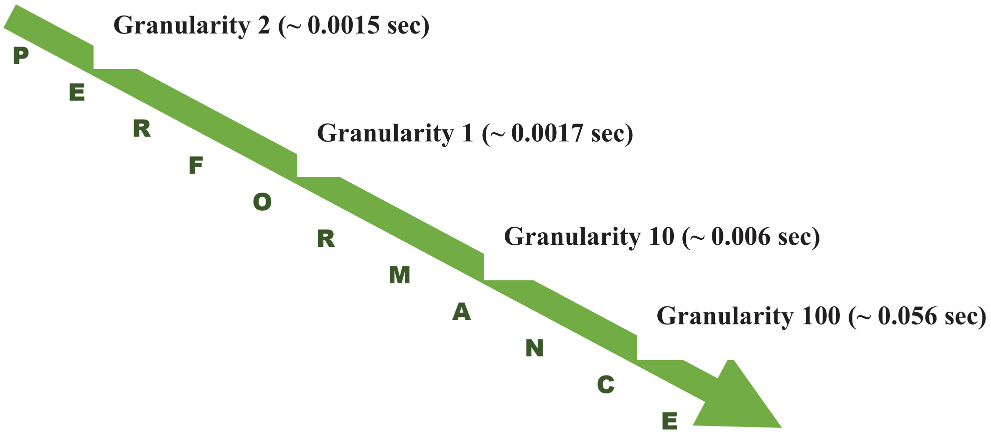

Granularity 2 (~ 0.0015 sec)
Granularity 1 (~ 0.0017 sec)
Granularity 10 (~ 0.006 sec)
Granularity 100 (~ 0.056 sec)
P E R F O R M A N C E

Figure 34: Granularity effect on AES-128 on 32000 bytes data

Figure 34 shows the execution time in ascending order for different granularities in AES-128 with a data size of 32000 bytes. It can be deduced from the figure that for granularity 2 which utilizes 1000 threads for encryption process the algorithm takes minimum execution time. Granularity 1 gives almost same results as granularity 1 with a small variation of factor 1.16. AES-128 with granularity 10 is ~ 4 times slower than granularity 2 and granularity 1. Granularity 100 gives the worst performance in this case with an execution time which is ~ 38 times slower than for granularity 2.

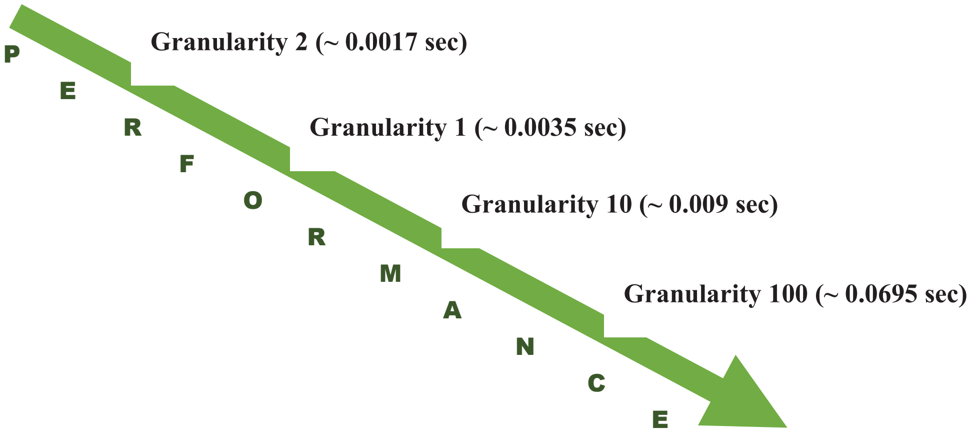

Granularity 2 (~ 0.0017 sec)
Granularity 1 (~ 0.0035 sec)
Granularity 10 (~ 0.009 sec)
Granularity 100 (~ 0.0695 sec)
PERFORMANCE

Figure 35: Granularity effect on AES-192 on 32000 bytes data

Figure 35 shows the execution time in ascending order for different granularities in AES 192 with a data size of 32000 bytes. It can be deduced from the figure that for granularity 2 which takes minimum execution time of ~0.0017 sec. It is twice faster than execution with granularity 1, ~ 5 times faster than granularity 10 and ~40 times faster than granularity 100.

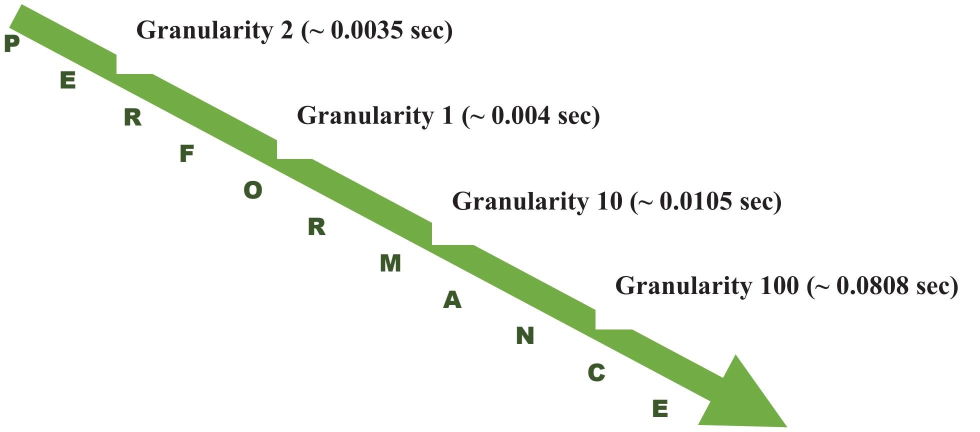

Granularity 2 (~ 0.0035 sec)
Granularity 1 (~ 0.004 sec)
Granularity 10 (~ 0.0105 sec)
Granularity 100 (~ 0.0808 sec)
PERFORMANCE

Figure 36: Granularity effect on AES-256 on 32000 bytes data

Figure 36 shows the execution time in ascending order for different granularities in AES 256 with a data size of 32000 bytes. It can be deduced from the figure that for granularity 2 which takes minimum execution. Execution time for granularity 1 is almost similar to that of granularity 1 with a negligible variation. Granularity 2 is 3 times faster than that in granularity 10 and ~23 times faster than granularity 100.

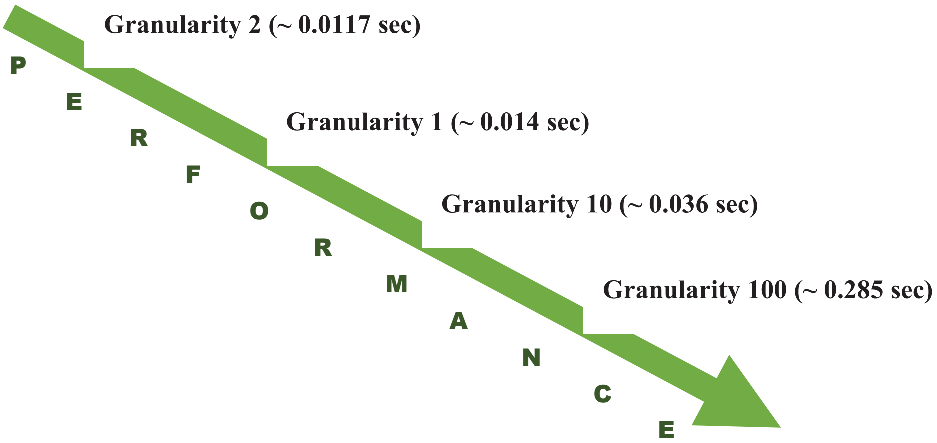

Granularity 2 (~ 0.0117 sec)
Granularity 1 (~ 0.014 sec)
Granularity 10 (~ 0.036 sec)
Granularity 100 (~ 0.285 sec)
PERFORMANCE

Figure 37: Granularity effect on AES-128 on 32000*5 bytes data

Figure 37 shows the execution time in ascending order for different granularities in AES 128 with a data size of 32000*5 bytes. It can be deduced from the figure that for granularity 2 which takes minimum execution time of ~0.0117 sec. It is approximately near to the execution time for granularity 1 with negligible variation. Granularity 2 gives execution time which is ~3 times faster than granularity 10 and ~24 times faster than granularity 100.

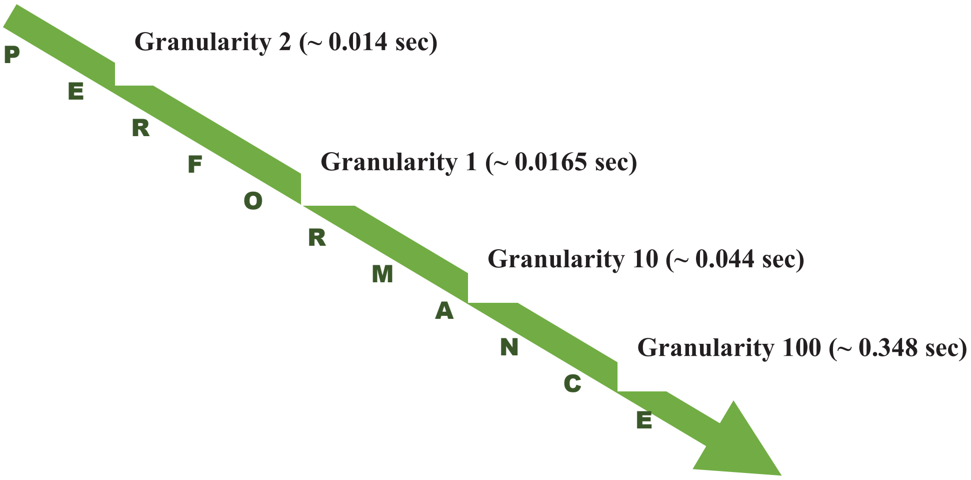

Granularity 2 (~ 0.014 sec)
Granularity 1 (~ 0.0165 sec)
Granularity 10 (~ 0.044 sec)
Granularity 100 (~ 0.348 sec)
PERFORMANCE

Figure 38: Granularity effect on AES-192 on 32000*5 bytes input

Figure 38 shows the execution time in ascending order for different granularities in AES 192 with a data size of 32000*5 bytes. It can be deduced from the figure that for granularity 2 which takes minimum execution time. It is approximately near to the execution time for granularity 1 with negligible variation. Granularity 2 gives execution time which is ~3 times faster than granularity 10 and ~25 times faster than granularity 100.

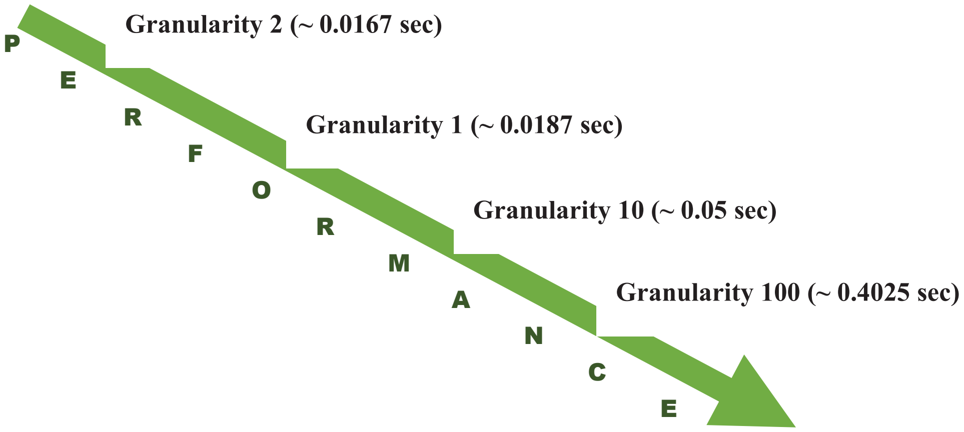

Granularity 2 (~ 0.0167 sec)
Granularity 1 (~ 0.0187 sec)
Granularity 10 (~ 0.05 sec)
Granularity 100 (~ 0.4025 sec)
PERFORMANCE

Figure 39: Granularity effect on AES-256 on 32000*5 bytes data

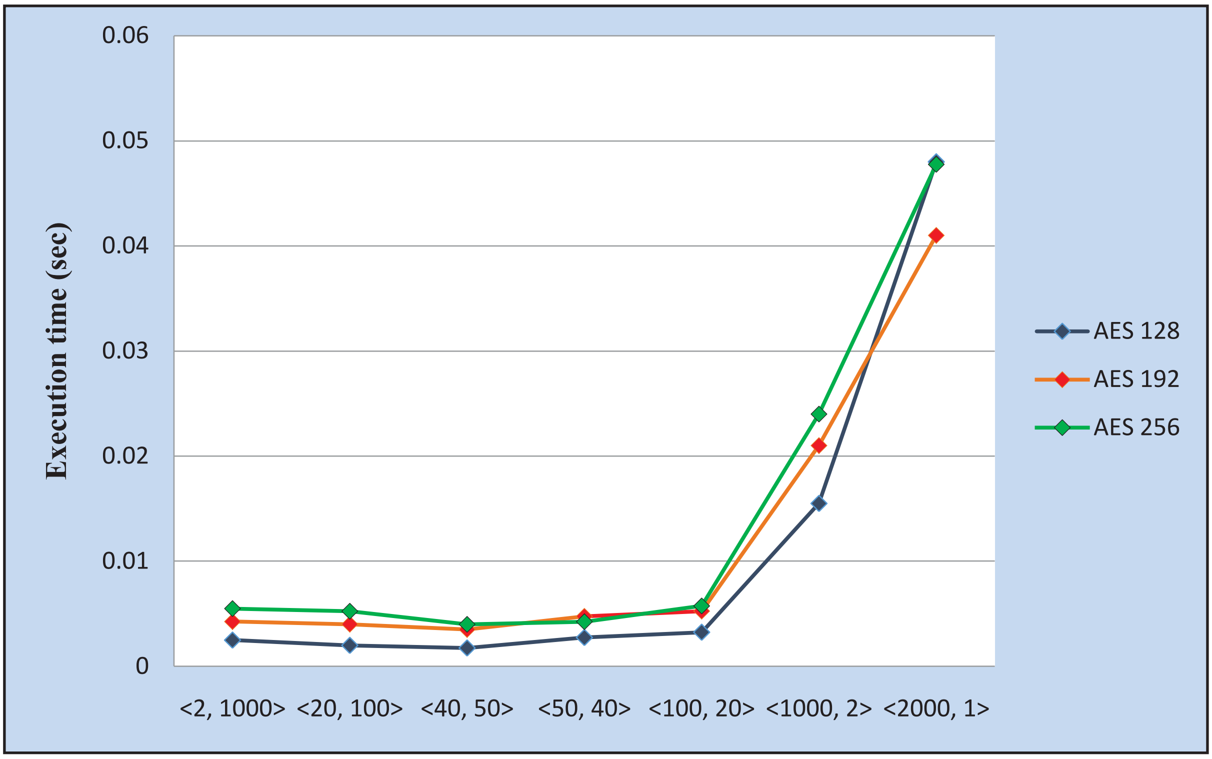
Figure 39 shows the execution time in ascending order for different granularities in AES 256 with a data size of 32000*5 bytes. It can be deduced from the figure that for granularity 2 which takes minimum execution time of ~0.0167 sec. It is approximately near to the execution time for granularity 1 with negligible variation. Granularity 2 gives execution time which is ~3 times faster than granularity 10 and ~24 times faster than granularity 100.

4.4 COMPARISON OF DIFFERENT GRID DIMENSIONS ON GPU

Grid dimension represents the distribution of threads and blocks in a GPU. It is indicated with a notation <B, T> where 'B' indicates the number of blocks utilized and 'T' indicates number of threads utilized in each block. For a combination of <B, T> a total of 'B×T' threads are utilized for the process.

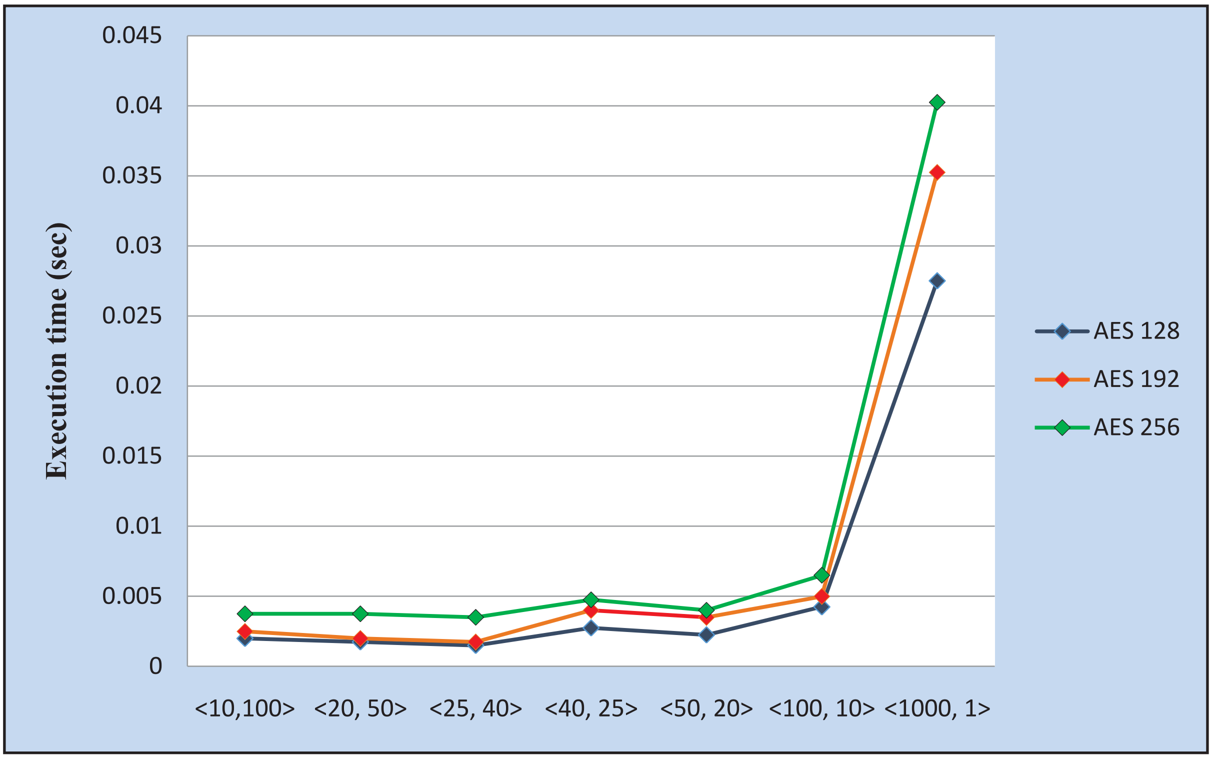
This section depicts the variation in performance in terms of speed in AES algorithm for different grid dimensions.

Graphs 1 – 8 show the variation in execution time of AES-128, AES-192 and AES-256 for a data of size 32000 bytes for granularity 1 where a total of 2000 threads are utilized. The vertical axis represents the execution time obtained for different block-thread combination indicated through the horizontal axis. It can be observed in the graph 1 that for AES-128, AES-192 and AES-256 the execution time for grid dimension <40, 50> is minimum giving the fastest execution followed by <20, 100>. The execution time is highest for grid dimension <2000, 2> followed by <1000, 2>.



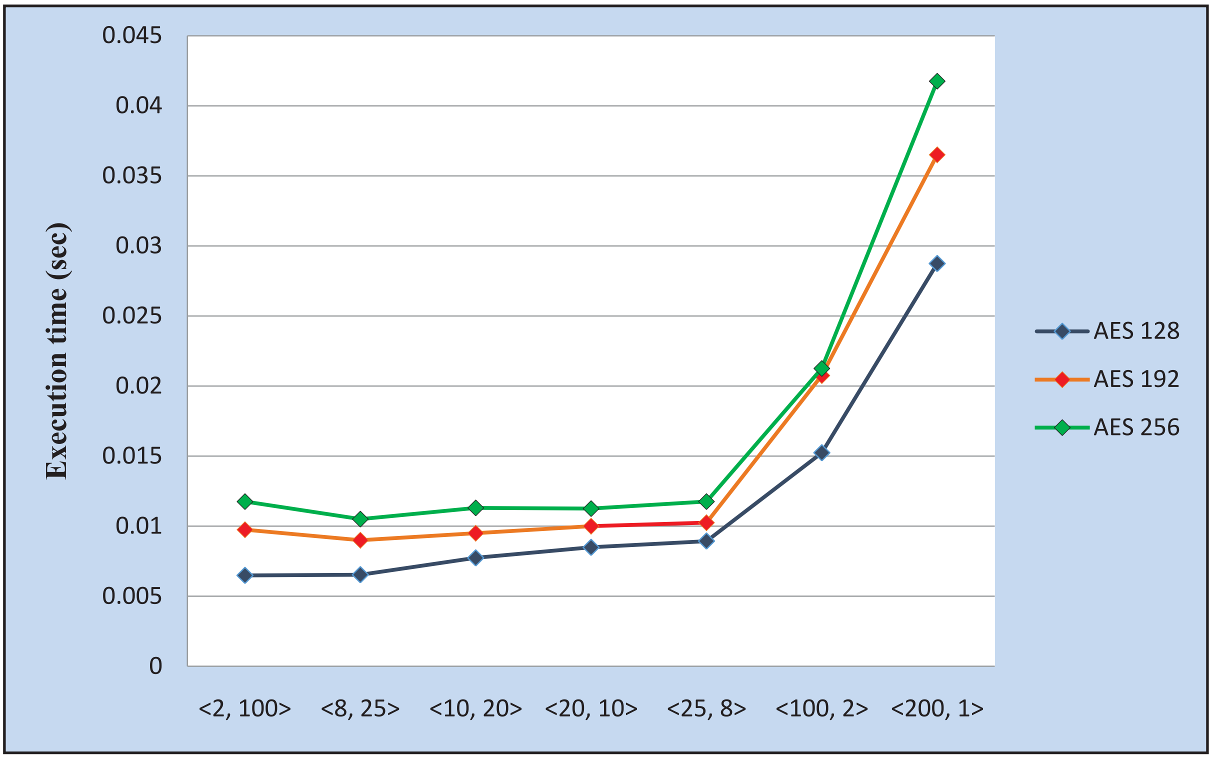
Graph 1: Performance comparison of varied grid dimensions with granularity 1 for 32000 bytes data

Graph 2 shows the variation in execution time of AES-128, AES-192 and AES-256 for a data of size 32000 bytes for granularity 2 where a total of 1000 threads are utilized. It can be depicted from the graph that AES-128 gives a faster execution for grid dimensions <25, 40> followed by grid dimensions <20, 50> and <10, 100>. Similar behaviour can be observed for AES-192 and AES-256. The execution time is highest for grid dimension <1000, 1> followed by <100, 10>.

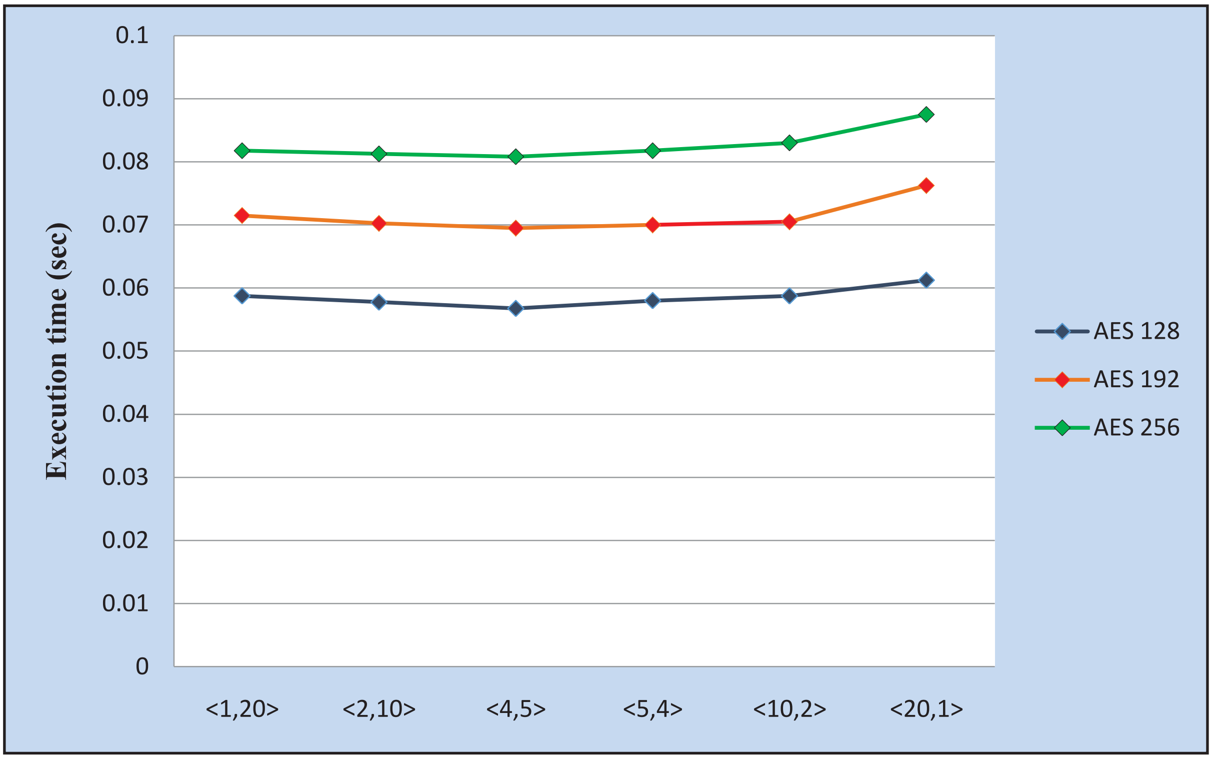


Graph 2: Performance comparison of varied grid dimensions with granularity 2 for 32000 bytes input

Graph 3 shows the variation in execution time of AES-128, AES-192 and AES-256 for a data of size 32000 bytes for granularity 10 where a total of 200 threads are utilized. It can be depicted from the graph that AES-128, AES-192 and AES-256 gives the fastest execution for grid dimension <8, 25> followed by grid dimensions <2, 100> and <10, 20>. The slowest performance is obtained by grid dimension <200, 1> followed by <100, 2>.



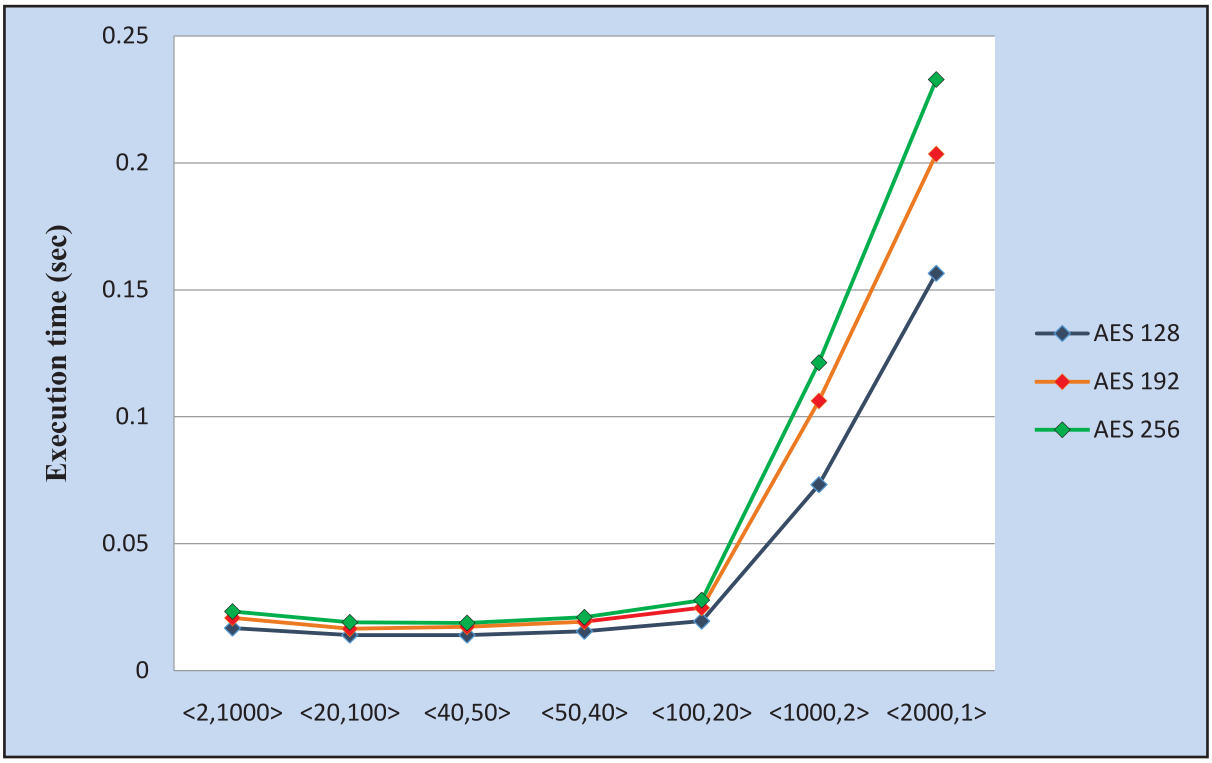
Graph 3: Performance comparison of varied grid dimensions with granularity 10 for 32000 bytes data



Graph 4: Performance comparison of varied grid dimensions with granularity 100 for 32000 bytes data

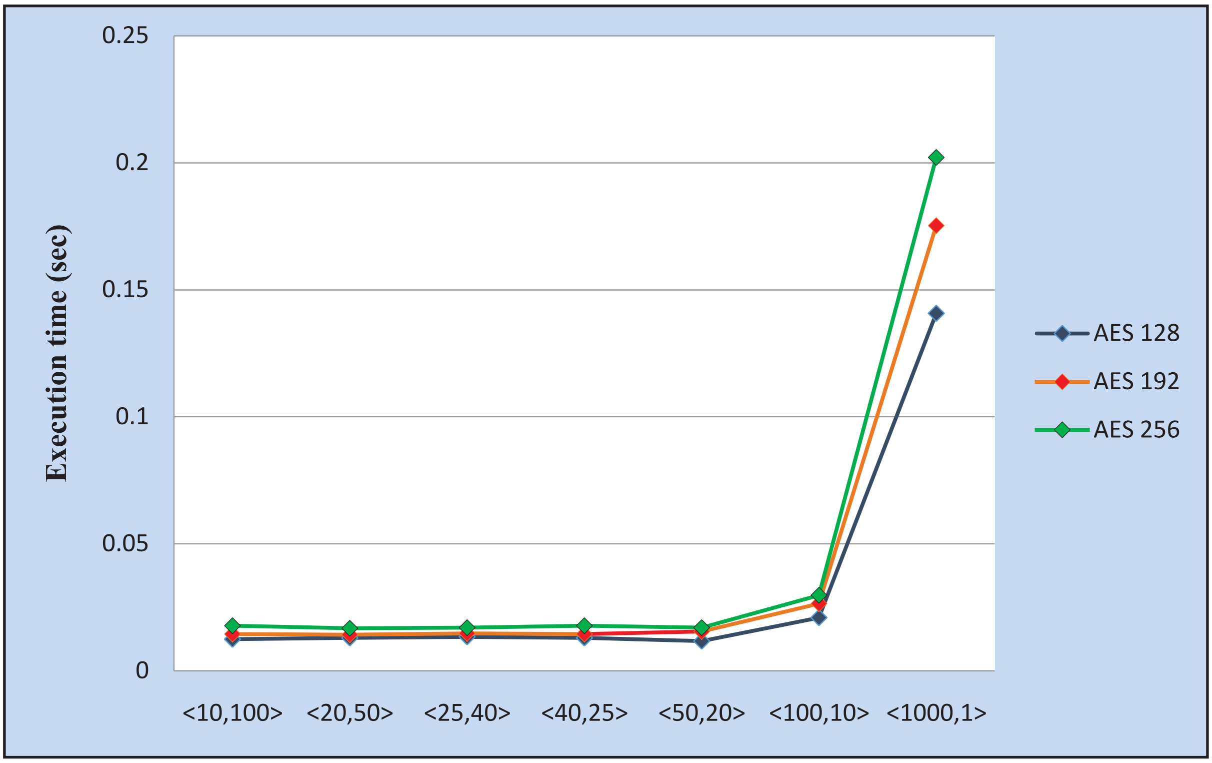
Graph 4 shows the variation in execution time of AES-128, AES-192 and AES-256 for a data of size 32000 bytes for granularity 100 where a total of 20 threads are utilized. It can be depicted from the graph that the fastest execution is obtained for grid dimensions <4, 5> followed by <2, 10>. On the other hand, the highest execution time is taken by grid dimension <20, 1>.

Graph 5 shows the variation in execution time of AES-128, AES-192 and AES-256 for a data of size 32000*5 bytes for granularity 1 where a total of 2000 threads are utilized. It can be depicted from the graph that AES-128, AES-192 and AES-256 give faster execution for grid dimension <40, 50> followed by <20, 100>. The highest execution time is taken by grid dimension <2000, 1> followed by <1000, 2>.

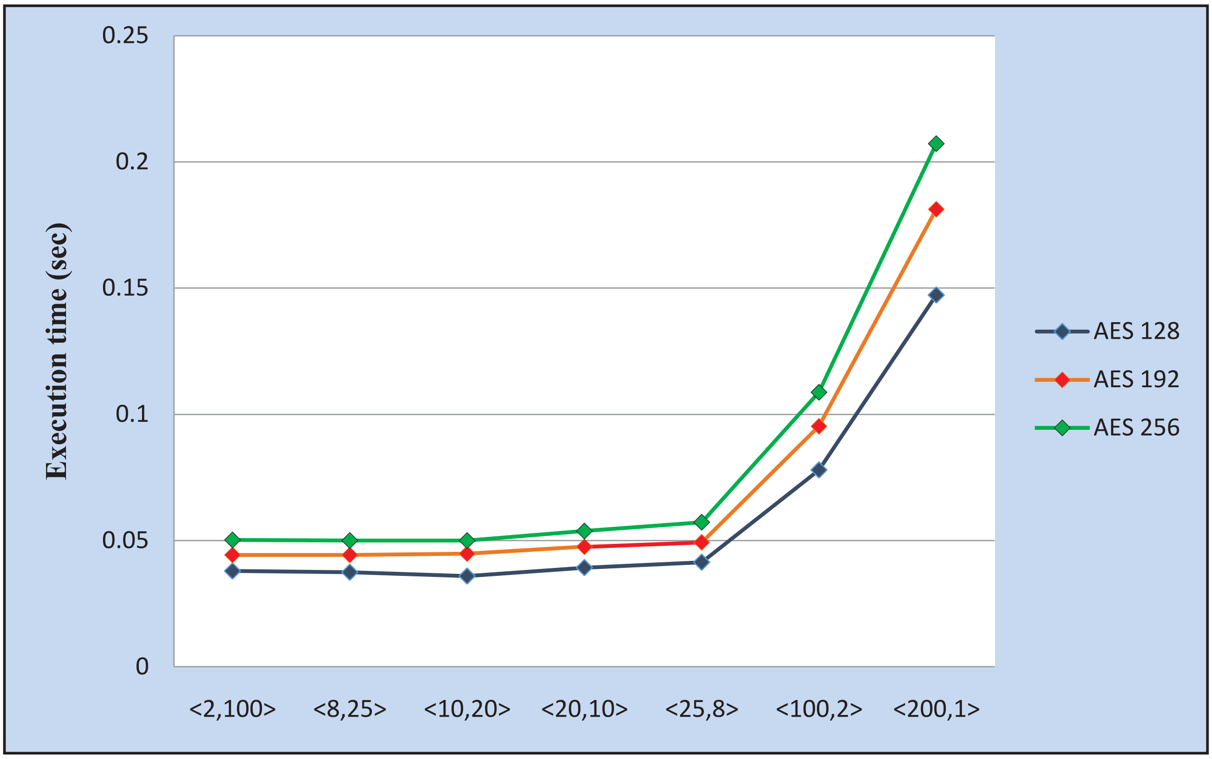


Graph 5: Performance comparison of varied grid dimensions with granularity 1 for 32000*5 bytes data

Graph 6 shows the variation in execution time of AES-128, AES-192 and AES-256 for a data of size 32000*5 bytes for granularity 2 where a total of 1000 threads are utilized. It can be depicted from the graph that fastest execution is given by grid dimensions <25, 40>, <20, 50> and <10, 100>. The highest execution time is taken by grid dimension <1000, 1> followed by <100, 10>.

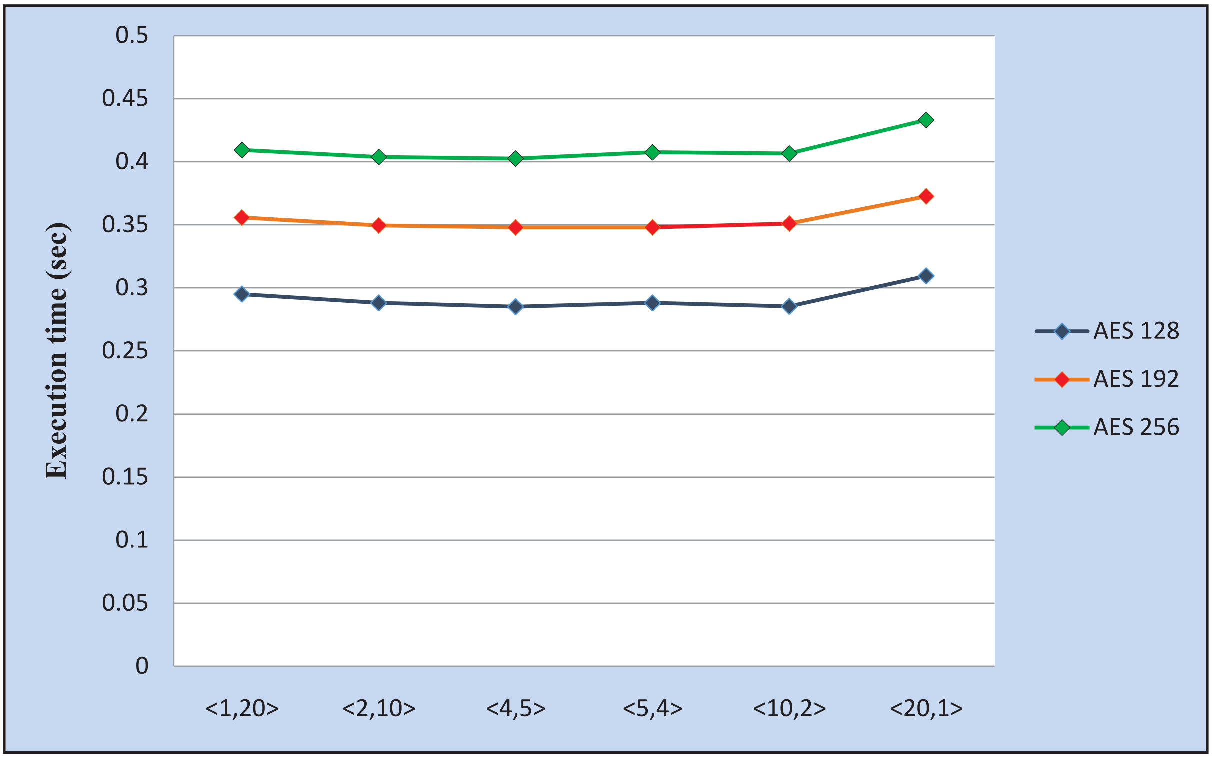


Graph 6: Performance comparison of varied grid dimensions with granularity 2 for 32000*5 bytes data



Graph 7: Performance comparison of varied grid dimensions with granularity 10 for 32000*5 bytes data

Graph 7 shows the variation in execution time of AES-128, AES-192 and AES-256 for a data of size 32000*5 bytes for granularity 10 where a total of 200 threads are utilized. It is depicted by the graph that grid dimension <8, 25> and <10, 20> gives the fastest execution followed by <2, 100>. The highest execution time is given by grid dimension <2000, 1> followed by <100, 2>.



Graph 8: Performance comparison of varied grid dimensions with granularity 100 for 32000*5 bytes data

Graph 8 shows the variation in execution time of AES-128, AES-192 and AES-256 for a data of size 32000*5 bytes for granularity 100 where a total of 20 threads are utilized. It can be depicted from the graph that the fastest execution is obtained by grid dimension <4, 5> followed by grid dimension <2, 10>. The highest execution time is given by grid dimension <20, 1> followed by <10, 2>.

4.5 IMPLEMENTATION ON MULTI-CORE CPU

Multi-core CPU has multiple processing cores on a single chip. Intel® Xeon® CPU E5-1650 used in this research has 6 independent processing cores with 2 virtual cores each. POSIX thread programming provides the benefit of efficiently utilizing the available cores on a system towards solving a time consuming complex problem. In this research a total of 12 threads are utilized for implementing AES-128, AES-192 and AES-256 encryption algorithms for data sizes 32000 bytes and 32000* bytes independently. Readings are taken for a single process run (k=1) and for 10 runs (k=10). The purpose of executing the process 10 times is to acquire better readability of data making it easy for comparison. This section depicts the results obtained from the implementation followed by its comparisons with previous implementations. Table 28, 29 and 30 shows the average execution time, standard deviation and confidence interval for multi threaded Pthreads programming for AES 128, AES 192 and AES 256 respectively.

Multi threaded C using POSIX THREAD for AES 128 **No of threads=12**		
	Data Size=32000	Data Size=32000*5
Average	0.0085	0.03475
Standard Deviation	0.002856	0.003024
Confidence Interval	0.007248 - 0.009752	0.033425 - 0.036075

Table 28: Execution time in multi-threaded C using Pthreads for AES-128

Multi threaded C using POSIX THREAD for AES 192 **No of threads=12**		
	Data Size=32000	Data Size=32000*5
Average	0.0095	0.0375
Standard Deviation	0.00394	0.004136
Confidence Interval	0.007773 - 0.011227	0.035687 - 0.039313

Table 29: Execution time in multi-threaded C using Pthreads for AES-192

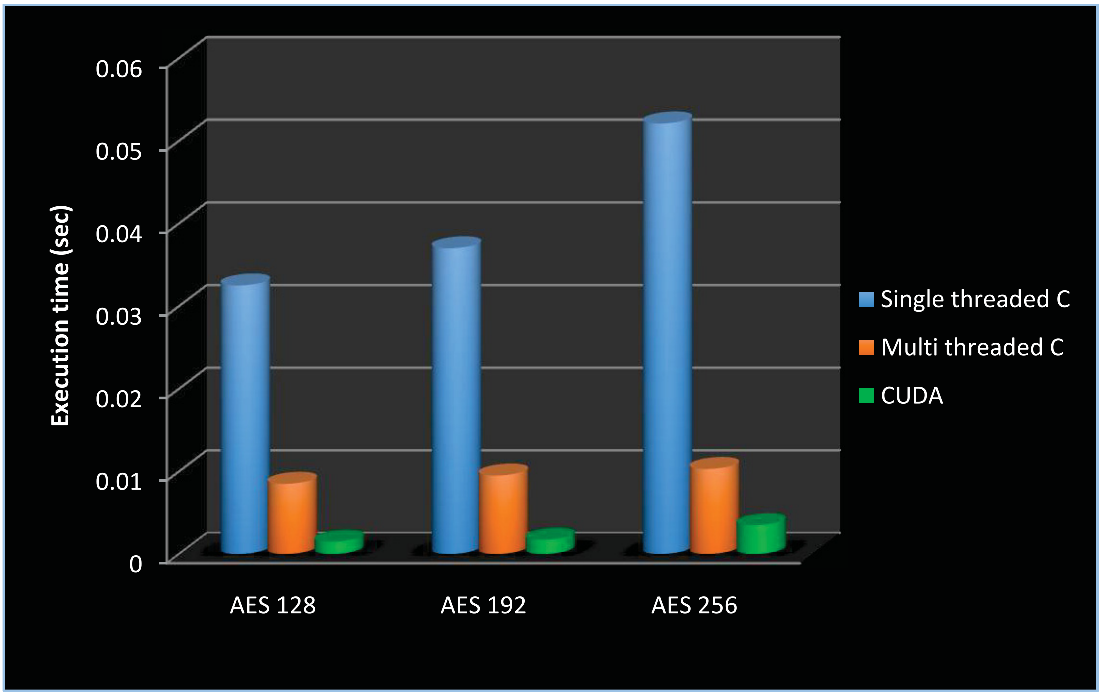
Multi threaded C using POSIX THREAD for AES 128 **No of threads=12**		
	Data Size=32000	Data Size=32000*5
Average	0.0103	0.04175
Standard Deviation	0.003278	0.006544
Confidence Interval	0.008863 - 0.011737	0.038882 - 0.044618

Table 30: Execution time in multi-threaded C using Pthreads for AES-256

4.6 PEROFMANCE COMPARISION OF GPU AND GPU

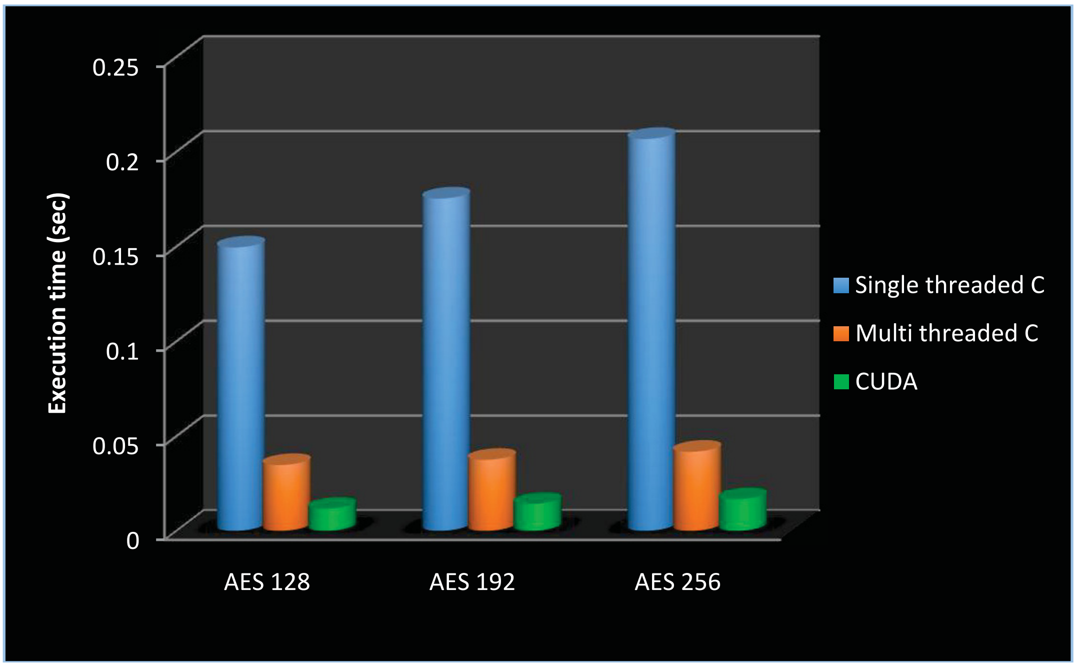
Graphs 9 and 12 shows the variation in execution time for the implementation of AES algorithms on a single threaded CPU using C, multi threaded CPU using POSIX thread and on GPU using CUDA programming.

Graph 9 shows the execution time variation in data size of 32000 bytes. It is observed for AES 128 in this case Pthread execution is ~4 times faster than single threaded C program, whereas CUDA is ~6 times faster than the Pthreads on CPU. For AES 192 algorithm, Pthread program is ~ 4 times faster than C and is ~ 5 times slower than CUDA. For AES 256 Pthread program gives almost 5 times faster execution than the single threaded C and is about 3 times slower than the CUDA program.



Graph 9: Performance comparison of C, Pthreads and CUDA for 32000 bytes data

Graph 10 shows the execution time variation in data size of 32000*5 bytes. It is observed that for AES 128, execution time of Pthreads program is ~4 times faster than single threaded C program, whereas CUDA on GPU is ~3 times faster than the Pthreads on CPU. For AES 192 algorithm, Pthread program is ~5 times faster than C and is ~ 3 times slower than CUDA. For AES 256 Pthread program gives execution ~ 5 times lower than the single threaded C whereas CUDA is ~2.5 times faster than Pthread program.



Graph 10: Performance comparison of C, Pthreads and CUDA for 32000*5 bytes data

For a summarized view the execution time of AES algorithm in ascending order is shown below.

- AES 128, for 32000 bytes input data:
$CUDA(G2) < CUDA(G1) < CUDA(G10) < pthreads < Single\ threaded\ C < CUDA(G100)$

- AES 128, for 32000*5 bytes input data:
$CUDA(G2) < CUDA(G1) < pthreads < CUDA(G10) < Single\ threaded\ C < CUDA(G100)$

- AES 192, for 32000 bytes input data:
$CUDA(G2) < CUDA(G1) < CUDA(G10) < pthreads < Single\ threaded\ C < CUDA(G100)$

- AES 192, for 32000*5 bytes input data:
$CUDA(G2) < CUDA(G1) < pthreads < CUDA(G10) < Single\ threaded\ C < CUDA(G100)$

- AES 256, for 32000 bytes input data:
$CUDA(G2) < CUDA(G1) < CUDA(G10) \sim pthreads < Single\ threaded\ C < CUDA(G100)$

- AES 128, for 32000*5 bytes input data:
$CUDA(G2) < CUDA(G1) < pthreads < CUDA(G10) < Single\ threaded\ C < CUDA(G100)$

**G1, G2, G10 and G100 denote Granularity 1, 2, 10 and 100 respectively.*

4.7 AES ALGORITHM USING CUDA STREAMS

The AES algorithm is implemented in an optimized CUDA version using CUDA STREAMS. CUDA STREAMS allows overlapping the communication and execution of kernel in a program. This section presents the outputs obtained from the implementation of AES-128, AES-192 and AES-256 using CUDA STREAMS for varied data sizes. 20 readings are taken for each combination to maintain accuracy. Readings are taken for a single process run (k=1) and for 10 runs (k=10). The purpose of executing the process 10 times is to acquire better readability of data making it easy for comparison. The tables below shows the average execution time, standard deviation and confidence interval of the values acquired during the process. Readings for k=10 is shown in APPENDIX A. The unit of measurements is in "seconds".

4.7.1 AES 128 using CUDA STREAMS

Case 1:

Data size: 32000 bytes
Number of streams used: 2
Data in each stream: 16000 bytes

	k=1	**k=10**
Average	0.002	0.02785
Standard Deviation	0.001101	0.002477
Confidence Interval	0.000898 - 0.003101	0.026765 - 0.028935

Table 31: AES-128 Execution using CUDA STREAMS for 2 streams with 16000 bytes each

The CUDA STREAMS program for AES 128 for a data size of 32000 bytes where each streams takes 16000 bytes as input gives an execution time ~0.002 sec which is ~ equal to time taken by CUDA without streams. There is no acceleration in execution of the algorithm using CUDA STREAMS.

Case 2:

Data size: 32000*2 bytes
Number of streams used: 2
Data in each stream: 32000 bytes

	k=1	**k=10**
Average	0.00275	0.0295
Standard Deviation	0.002552	0.020765
Confidence Interval	0.001632 - 0.003868	0.018487 - 0.030513

Table 32: AES-128 Execution using CUDA STREAMS for 2 streams with 32000 bytes each

For a data size of 32000*2 bytes using two streams, each stream taking 32000 bytes as input the execution time is ~ 0.0027 seconds, which is ~ equal to the time taken to execute 32000 bytes input in the first case with a small variation of factor 1.3.

Case 3:

Data size: 32000*5 bytes
Number of streams used: 5
Data in each stream: 32000 bytes

	k=1	k=10
Average	0.0105	0.10525
Standard Deviation	0.001539	0.003024
Confidence Interval	0.009826 - 0.011174	0.103925 - 0.106575

Table 33: AES-128 Execution using CUDA STREAMS for 5 streams with 32000 bytes each

CUDA STREAMS for 5 streams, with each stream taking 32000 bytes as input, (total input size =32000*5) takes ~ 0.010 seconds which is ~ 1.11 times faster than execution using CUDA with single stream.

4.7.2 AES 192 using CUDA STREAMS

Case 1:

Data size: 32000 bytes
Number of streams used: 2
Data in each stream: 16000 bytes

	k=1	k=10
Average	0.0035	0.0335
Standard Deviation	0.002350	0.002856
Confidence Interval	0.002469 - 0.004530	0.032248 - 0.034752

Table 34: AES-192 Execution using CUDA STREAMS for 2 streams with 16000 bytes each

The CUDA STREAMS program for AES 192 for a data size of 32000 bytes where each streams takes 16000 bytes as input gives an execution time which is ~ equal to the time taken by CUDA program without using streams. There is no acceleration in the execution speed using CUDA STREAMS.

Case 2:

Data size: 32000*2 bytes
Number of streams used: 2
Data in each stream: 32000 bytes

	k=1	k=10
Average	0.007	0.06325
Standard Deviation	0.002513	0.002447
Confidence Interval	0.005899 - 0.008101	0.062178 - 0.064322

Table 35: AES 192 Execution using CUDA STREAMS for 2 streams with 32000 bytes each

For a data size of 32000*2 bytes using two streams, each stream taking 32000 bytes as input the execution time is ~ 0.007 seconds, which is twice the time taken in case 1.

Case 3:

Data size: 32000*5 bytes
Number of streams used: 5
Data in each stream: 32000 bytes

	k=1	k=10
Average	0.0135	0.12925
Standard Deviation	0.002351	0.002447
Confidence Interval	0.012470 - 0.014530	0.128178 - 0.130322

Table 36: AES-192 Execution using CUDA STREAMS for 5 streams with 32000 bytes each

CUDA STREAMS for 5 streams, with each stream taking 32000 bytes as input, (total input size =32000*5) takes ~ 0.013 seconds which is ~ 1.05 times faster than CUDA without streams.

4.7.3 AES 256 using CUDA STREAMS

Case 1:

Data size: 32000 bytes
Number of streams used: 2
Data in each stream: 16000 bytes

	k=1	k=10
Average	0.004	0.0375
Standard Deviation	0.002051	0.003035
Confidence Interval	0.003100 - 0.004899	0.03617 - 0.03883

Table 37: AES-256 Execution using CUDA STREAMS for 2 streams with 16000 bytes each

The CUDA STREAMS program for AES 256 for a data size of 32000 bytes where each streams takes 16000 bytes as input gives an execution time which is ~ equal to the time taken by CUDA without streams.

Case 2:

Data size: 32000*2 bytes
Number of streams used: 2
Data in each stream: 32000 bytes

	k=1	k=10
Average	0.00775	0.07225
Standard Deviation	0.002552	0.002552
Confidence Interval	0.006632 - 0.008868	0.071132 - 0.073368

Table 38: AES-256 Execution using CUDA STREAMS for 2 streams with 32000 bytes each

For a data size of 32000*2 bytes using two streams, each stream taking 32000 bytes as input the execution time is ~ 0.008 seconds, which is ~2 times higher than execution time in case 1.

Case 3:

Data size: 32000*5 bytes
Number of streams used: 5
Data in each stream: 32000 bytes

	k=1	**k=10**
Average	0.01625	0.15
Standard Deviation	0.002221	0.001231
Confidence Interval	0.015276 - 0.017224	0.148769 - 0.151231

Table 39: AES-256 Execution using CUDA STREAMS for 5 streams with 32000 bytes each

CUDA STREAMS for 5 streams, with each stream taking 32000 bytes as input, (total input size =32000*5) takes ~ 0.015 seconds which is ~ 1.03 times of the time taken by CUDA program without using streams.

The results show that CUDA STREAMS is inefficient to accelerate the execution of AES algorithm compared to CUDA without using streams. However there is a small increment in speed for higher data size per stream.

CHAPTER 5 DISCUSSIONS

5.1 VALIDITY THREATS

There are two types of validity threats that need to be considered while conducting a research namely, internal validity and external validity.

5.1.1 Internal Validity

Internal Validity refers to the ability of the research paper to be able to establish relation between cause and effect [38]. Deviation from this validity can affect the correctness of the results and its inferences. In order to avoid this validity threat the following steps have been taken:

- The algorithm design has been thoroughly understood before formulating the code.
- 20 readings have been taken in every case in order to maintain accuracy of the results for a fair and accurate comparison. Standard deviation and confidence interval has been calculated to assure consistency in the results. The readings are taken upto 6 digits after decimal to assure accuracy.
- The tests have been performed for k=1 and k=10, where k denotes the number of process runs. The purpose is to increase the readability for accurate and easy comparison. The tests has given consistent results.
- Possible optimizations have been taken while formulating the code in order to present a valid comparison. Global memory usage is done where ever applicable so as to reduce latency of accessing memory. Each work item can access those data directly from the global memory. These data includes, the SBOX, multiplication tables and cipher key which is common for the entire data chunk.
- The compiler used has been set to full optimization to get fastest execution possible.
- The correctness of the code is tested in a step by step manner at every stage to assure valid output.
- The correctness of output is tested function by function comparing it with standard examples.
- The consistency in the generated output is been tested for every implementation in order to maintain correct cipher-text.
- Code optimisation level is maintained constant for all the implementations to make sure that the amount of work done in a single process by a core remains constant in all the devices for a fair comparison.

- The performance variation pattern has been critically observed for different combinations for the three algorithms to assure consistency and accuracy.

5.1.2 External Validity

External Validity deals with the generalizability of the results of the research work outside the study [38]. The functions in the algorithm are formed referring to the defined standard. The functions therefore are universal and acceptable, and would hold good for execution in real time environment. Although more optimizations can be performed in order to get better performances, this research work used the considerable optimizations. The obtained performance is independent of the type of input, assuring the correctness of the results obtained.

This research work uses data parallelism technique on the AES algorithm. Hence, it is the data size that effects the change in the performance of devices, and not the algorithm complexity. The inference acquired through this research is expected to hold good for any application on which data parallelism is applied.

The execution time of the device depends upon the kind of GPU used. This research work uses a NVIDIA Quadro K4000 which has Kepler architecture. The algorithm performance may show variation in the comparison factor between the two devices, yet the general observation holds good for all NVIDIA GPUs using CUDA

5.2 DISCUSSIONS

Optimisation comparability of CPU and GPU:

AES algorithm as explained in chapter 2 consists of two main functions, namely the key expansion and encryption. The cipher key is constant for all the chunks of input data and hence is stored in the global memory. The C program on the CPU executes the key expansion function only once and stores the results in the global memory which is then used for encryption by all the input data chunks. In case of CUDA version on the GPU, the key expansion function is an integral part of the encryption function and hence each thread individually calculates the round keys for its own data. This may question the comparability of the performance of the two devices. The argument here is that, the key expansion function takes negligible time to execute by a core on the GPU. The key expansion function has been tested on the GPU and the results show that for 10 runs of key expansion function the execution time is ~ less than 0.001 sec. Hence, a single key expansion function by all cores simultaneously, will have no impact on the overall performance of the device. The generated round keys are stored in the local memory of each core which is supposed to be the fastest memory leading to fast execution of data. However for data size of more than 32000*10 bytes such that each core handles more than 10 data chunks, the key expansion function might decrease the performance of the device.

Thread/block combinations of the GPU:

The performance of the algorithm on the devices is seen to be higher in two cases of thread/block distribution. One is when there is a balanced distribution of blocks and threads in each block. For example in case of granularity 1, 2, 10 and 100 on an average the grid dimension <40, 50>, <25, 40>, <8, 25> and <4, 5> respectively gives relatively higher performance compared to other combinations. Another case which gave ~ equal or slightly lower performance is when the threads of each block are effectively utilized however considering the balance in distribution. For example in case of granularity 1, 2, 10 and 100 on a average the grid dimension <20, 100>, <10, 100>, <20, 10> and <2, 10> gives the next high performance. It can also be deduced that for extreme grid dimensions the execution time is higher than other combinations. It is the situation when threads in each block are not efficiently utilized. For example in case of granularity 1, 2, 10 and 100, grid dimensions <2000, 1>, <1000, 1>, <200, 1>, and <20, 1> respectively give the least performance. From the above observation it is understood that in order to generate maximum results there needs to be a trade of between the number of blocks and threads used. The grid needs to be organized such that the maximum number of blocks with maximum number of threads in each block can give a considerably higher performance. The ratio of number of threads to that of blocks shown below, should be greater than 1, and as small and possible.

$$number\ of\ threads/number\ of\ blocks$$

Higher ratio tends to higher execution time, as can be seen in graph 1 to graph 8.

Effect of granularity:

From the deduced results it is observed that for all the three versions of AES algorithms, granularity 2 provides the fastest execution giving the least execution time compared to granularity 1, 10 and 100, followed by granularity 1 and granularity 10. Whereas granularity 100 gives the least performance. From this observation it is inferred that to achieve high GPU performance there needs to be proper trade off between the amount of task done and utilization of threads in the device. In case of granularity 10 and 100, the amount of task by each thread is high but the threads in the device are not efficiently utilized. The execution time of granularity 100 is higher than the single threaded C program and for higher data size the execution time of granularity 10, and granularity 100 moves higher giving unacceptable performance reduction. The reason is that for the case of granularity 100, the program on whole consists of heavy serial computation and hence performs better on the CPU. The conclusion is that, “higher granularities serve better, provided the program contains sufficient parallelism”.

Thread utilization by Pthreads program:

The CPU used in this experiment is Intel….CPU, which has 6 physical cores and 12 virtual cores. Assuming that each core would take 2 cores to execute the algorithm, utilizing 12 threads would give the best performance by the CPU. For a data of size 32000 bytes, 12 threads has been utilized such that 10 threads handle 167 sets of input each and 2 threads handles 165 threads each, making it a total of 2000 sets. For data size of 32000*5 bytes, 10 threads handle 833 sets of data chunk each and 2 threads handle 835 data chunks each. The variation in data division is negligible and has no impact on the reliability of results.

Effectiveness of CUDA STREAM in accelerating performance:

CUDA STREAMS are effective when each stream has considerable data to process. The effect of latency can be compensated when the work to be processed is considerably high. In this research work it can be observed for data size of 32000 bytes, two streams with 16000 bytes each gives a performance with is lower than the performance of the CUDA program with single stream. The reason is the lack of effective utilization of threads in each stream. Instead, for each stream taking a data of 32000 bytes performs execution in the same amount of time. CUDA STREAM program accelerated the performance giving a slightly lower execution time in case of data size of 32000*5 bytes in which 5 streams are utilized such that each stream handles 32000 bytes of data. An intermediate value has been chosen in order to analysis the pattern of performance variation, an intermediate value has been chosen, increasing data size to 64000 bytes. Two streams have been utilized such that each stream handles 32000 bytes of data. This implementation shows small acceleration in performance. It can be inferred that CUDA STREAMS are effective only for higher data sizes and effective utilization of threads in each stream.

CHAPTER 6 CONCLUSION AND FUTURE WORK

6.1 ANSWER TO RESEARCH QUESTIONS

RQ1.

a) Does the GPU outperform the single core CPU in the implementation of AES algorithm?

Answer:

The GPU outperforms the single core CPU in the implementation of AES algorithm. For a data size of 32000 bytes, it accelerates the execution of AES 128, AES 192 and AES 256 by a factor of 22, 21 and 12 respectively. For a data size of 32000*5 bytes in which data is divided into chunks of 32000 bytes each to be processed by the GPU in sequence, it accelerates the execution of AES 128, AES 192 and AES 256 by a factor of 13, 12 and 12 times respectively.

b) Does the GPU outperform the multi core CPU in the implementation of AES algorithm?

Answer:

The GPU outperforms the multi core CPU in the implementation of AES algorithm. For a data size of 32000 bytes, the GPU performs 6, 5 and 3 times faster than multi core CPU for the implementation of AES 128, AES 192 and AES 256 respectively. For data of size 32000*5 bytes, where the data is sequentially processed in chunks of 32000 bytes each, the GPU is 3, 3 and 2.5 times faster than CPU for the implementation of AES 128, AES 192 and AES 256 respectively.

RQ2. Does the use of CUDA STREAMS have a positive impact on the performance of the AES algorithm?

Answer:

The use of CUDA STREAMS does not have a positive impact on the performance of AES algorithm for a data size of 32000 bytes. However it shows an increase in the performance for data sizes of 32000*5 bytes. Hence, the CUDA STREAMS can have a positive impact on the AES algorithm for higher data size with considerable data in each stream.

RQ3. Which implementation gives the fastest AES execution?

Answer:

For AES 128, AES 192 and AES 256, the CUDA program on the GPU with granularity 2 in which each thread handles 32 bytes of input data which is internally executed in chunks of 16 bytes, with a balanced thread/block distribution gives the fastest execution. Granularity 2 implies that each thread handles 32 bytes of input data which is internally executed in chunks of 16 bytes. The task is distributed between the threads and blocks in such a way that the GPU utilizes efficient number of blocks with efficient number of threads in each block, effectively employing GPU for parallelism.

** The results are clearly depicted in Chapter 4 and discussed in chapter 5*

6.2 FUTURE WORK

This thesis work is limited, due to many constraints out of which time is a crucial player. To expand the findings related to exploitation of parallelism using GPU, the following are some of the avenues for future work.

- This research work can be performed on other cryptographic ciphers such as blowfish, serpent, and Salsa20 to find out if the results hold good for those implementations.
- It would be of value to exploit task parallelism to accelerate the AES algorithm.
- Similar algorithm model can be implemented on different GPUs in order to compare and deduce absolute results.
- It would be interesting to see how the use of a heterogeneous system i.e., the combination of CPU and GPU towards the execution of AES algorithm can accelerate its performance.

BIBLIOGRAPHY

[1] B. P. Luken, M. Ouyang, and A. H. Desoky, "AES and DES Encryption with GPU," in *ISCA PDCCS*, 2009, pp. 67–70.

[2] L. Swierczewski, "3DES ECB Optimized for Massively Parallel CUDA GPU Architecture," *ArXiv13054376 Cs*, May 2013.

[3] H.-P. Yeh, Y.-S. Chang, C.-F. Lin, and S.-M. Yuan, "Accelerating 3-DES Performance Using GPU," in *International Conference on Cyber-Enabled Distributed Computing and Knowledge Discovery (CyberC)*, 2011, pp. 250–256.

[4] F. Shao, Z. Chang, and Y. Zhang, "AES Encryption Algorithm Based on the High Performance Computing of GPU," in *Second International Conference on Communication Software and Networks, 2010. ICCSN '10*, 2010, pp. 588–590.

[5] X. Wang, X. Li, M. Zou, and J. Zhou, "AES finalists implementation for GPU and multi-core CPU based on OpenCL," in *IEEE International Conference on Anti-Counterfeiting, Security and Identification (ASID)*, 2011, pp. 38–42.

[6] H. Zhang, D. Zhang, and X. Bi, "Comparison and Analysis of GPGPU and Parallel Computing on Multi-Core CPU."

[7] M. Bobrov, "Cryptographic algorithm acceleration using CUDA enabled GPUs in typical system configurations," *Theses*, Aug. 2010.

[8] N.-P. Tran, M. Lee, and D. H. Choi, "Heterogeneous parallel computing for data encryption application," in *6th International Conference on Computer Sciences and Convergence Information Technology (ICCIT)*, 2011, pp. 562–566.

[9] D. Le, J. Chang, X. Gou, A. Zhang, and C. Lu, "Parallel AES algorithm for fast Data Encryption on GPU," in *2nd International Conference on Computer Engineering and Technology (ICCET)*, 2010, vol. 6, pp. V6–1–6.

[10] Q. Li, C. Zhong, K. Zhao, X. Mei, and X. Chu, "Implementation and Analysis of AES Encryption on GPU," *in 2012 IEEE 14th International Conference on High Performance Computing and Communication 2012 IEEE 9th International Conference on Embedded Software and Systems (HPCC-ICESS)*, 2012, pp. 843–848.

[11] A. M. Chiuta, "AES Encryption and Decryption Using Direct3D 10 API," *ArXiv12010398 Cs*, Jan. 2012.

[12] T. R. Daniel and S. Mircea, "AES on GPU using CUDA," in *European Conference for the Applied Mathematics & Informatics. World Scientific and Engineering Academy and Society Press*, 2010.

[13] R. Inam, "An Introduction to GPGPU Programming-CUDA Architecture," *Mälardalen Univ. Mälardalen Real-Time Res. Cent.*, 2011.

[14] C. Cullinan, C. Wyant, T. Frattesi, and X. Huang, "Computing performance benchmarks among cpu, gpu, and fpga," *Internet Www. Wpi EduPubsE-Proj.-Proj.-030212-123508unrestrictedBenchmarking Final*, 2013.

[15] H. Jo, S.-T. Hong, J.-W. Chang, and D. H. Choi, "Data Encryption on GPU for High-Performance Database Systems," *Procedia Comput. Sci.*, vol. 19, pp. 147–154, 2013.

[16] V. W. Lee, C. Kim, J. Chhugani, M. Deisher, D. Kim, A. D. Nguyen, N. Satish, M. Smelyanskiy, S. Chennupaty, P. Hammarlund, R. Singhal, and P. Dubey, "Debunking the 100X GPU vs. CPU Myth: An Evaluation of Throughput Computing on CPU and GPU," in *Proceedings of the 37th Annual International Symposium on Computer Architecture*, New York, NY, USA, 2010, pp. 451–460.

[17] O. Gervasi, D. Russo, and F. Vella, "The AES Implantation Based on OpenCL for Multi/many Core Architecture," in *International Conference on Computational Science and Its Applications (ICCSA)*, 2010, pp. 129–134.

[18] N. Nishikawa, K. Iwai, H. Tanaka, and T. Kurokawa, "Throughput and Power Efficiency Evaluations of Block Ciphers on Kepler and GCN GPUs," in *First International Symposium on Computing and Networking (CANDAR)*, 2013, pp. 366–372.

[19] C. Gregg and K. Hazelwood, "Where is the data? Why you cannot debate CPU vs. GPU performance without the answer," in *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2011, pp. 134–144.

[20] M. Taher, "Accelerating scientific applications using GPU's," *in Design and Test Workshop (IDT)*, *2009 4th International*, 2009, pp. 1–6.

[21] V.Venugopal and D. M. Shila, "High throughput implementations of cryptography algorithms on GPU and FPGA," *in Instrumentation and Measurement Technology Conference (I2MTC), 2013 IEEE International*, 2013, pp. 723–727.

[22] D. L. Cook, J. Ioannidis, A. D. Keromytis, and J. Luck, "CryptoGraphics: Secret key cryptography using graphics cards," *in Topics in Cryptology–CT-RSA 2005, Springer,* 2005, pp. 334–350.

[23] K. Iwai, T. Kurokawa, and N. Nisikawa, "AES Encryption Implementation on CUDA GPU and Its Analysis," in *2010 First International Conference on Networking and Computing (ICNC)*, 2010, pp. 209–214.

[24] P. Maistri, F. Masson, and R. Leveugle, "Implementation of the Advanced Encryption Standard on GPUs with the NVIDIA CUDA framework," *in 2011 IEEE Symposium on Industrial Electronics and Applications (ISIEA),* 2011, pp. 213–217.

[25] "FIPS 197, Advanced Encryption Standard (AES) [Online]. Available: http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf

[26] V. Xouris, "Parallel hashing, compression and encryption with opencl under os x," *Master's thesis, School of Informatics-University of Edinburgh,* 2010.

[27] "GPU Gems 3 - Chapter 36. AES Encryption and Decryption on the GPU." [Online]. Available: http://http.developer.nvidia.com/GPUGems3/gpugems3_ch36.html.

[28] C. McClanahan, "History and Evolution of GPU Architecture," *Pap. Surv. Httpmcclanahoochie Comblogwpcontentuploads201103gpu-Hist-Pap. Pdf*, 2010.

[29] A. Barnes, R. Fernando, K. Mettananda, and R. Ragel, "Improving the throughput of the AES algorithm with multicore processors," in *2012 7th IEEE International Conference on Industrial and Information Systems (ICIIS)*, 2012, pp. 1–6.

[30] "On Fair Comparison between CPU and GPU." [Online]. Available: http://www.eecs.berkeley.edu/~sangjin/2013/02/12/CPU-GPU-comparison.html.

[31] J. Ortega, H. Trefftz, and C. Trefftz, "Parallelizing AES on multicores and GPUs," in *Proceedings of the IEEE International Conference on Electro/Information Technology (EIT)*, 2011, pp. 15–17.

[32] C. So-In, S. Poolsanguan, C. Poonriboon, K. Rujirakul, and C. Phudphut, "Performance Evaluation of Parallel AES Implementations over CUDA GPU Framework," *Int J Digit. Content Technol. Its Appl.*, vol. 7, no. 5, pp. 501–511, 2013.

[33] "Slide 1 - StreamsAndConcurrencyWebinar".[Online] Available: http://on-demand.gputechconf.com/gtc-express/2011/presentations/StreamsAndConcurrencyWebinar.pdf

[34] "POSIX Threads Programming."[Online].Available:https://computing.llnl.gov/tutorials/pthreads/.

[35] J. Gómez-Luna, J. M. González-Linares, J. I. Benavides, and N. Guil, "Performance models for CUDA streams on NVIDIA GeForce series," Technical Report, University of Málaga, 2011. http://www. ac. uma. es/∼vip/publications/UMA-DAC-11-02. pdf.

[36] M. Domeika, "Development and Optimization Techniques for Multi-Core Processors," *Dr. Dobb's*. [Online]. Available: http://www.drdobbs.com/development-and-optimization-techniques/212600040.

[37] D. B. Kirk and W. W. Hwu, *Programming Massively Parallel Processors, Second Edition: A Hands-on Approach*, 2 edition. Amsterdam; Boston; Waltham, Mass.: Morgan Kaufmann, 2012.

[38] Steven Taylor and Gordon J.G. Asmundson, "Ineternal and External Validity In Clinical Research", [Online].Available: http://www.sagepub.com/upm-data/19352_Chapter_3.pdf.

[39] Aes algorithm flash: Rijndael (2004). [Online].Available: http://www.cs.bc.edu/~straubin/cs381-05/blockciphers/rijndael_ingles2004.swf.

[40] "Huong Nguyen: Computer systems.[Online]. Available: http://cnx.org/contents/611fa6c7-a16d-460a-a221-ae57ff2379be@1.

[41]“iXBT Labs Review - NVIDIA CUDA,” *iXBT Labs*. [Online]. Available: http://ixbtlabs.com/articles3/video/cuda-1-p1.html.

[42]“Ron Maltiel: Semiconductor Experts, Witnesses, Consultants and Patent Litigation Support" (March 2012). [Online]. Available: http://ixbtlabs.com/articles3/video/cuda-1-p1.html.

APPENDIX A

This section delivers the execution time for AES 128, AES 192 and AES 256 encryption algorithm, using single threaded C, multi-threaded C (Pthreads), and CUDA. 20 runs have been taken for every implementation, for a single process (k=1), and for 10 processes (k=10). As mentioned earlier, the process is run 10 times in order to achieve readable data for easy comparison. The execution time for k=10 is approximately 10 times that of k=1. This section also shows the average execution time, standard deviation and confidence interval for every implementation on the CPU and the GPU for k=10, as a continuation to Chapter 4.

TableA.1 to Table A.6 shows the execution time for AES implementation on single threaded CPU.
TableA.7 to TableA.78 shows the execution time with different granularities and grid dimension on GPU using CUDA.
TableA.79 to TableA.84 shows the execution time obtained on multi threaded CPU using Pthreads.
TableA.85 to TableA.93 shows the execution time obtained on the GPU for different data sizes using CUDA STREAMS.

Single threaded C for AES 128				
	Data Size=32000		Data Size=160000	
S.No	**k=1**	**k=10**	**k=1**	**k=10**
1	0.025	0.28	0.155	1.385
2	0.03	0.285	0.14	1.37
3	0.03	0.275	0.155	1.355
4	0.035	0.275	0.14	1.395
5	0.035	0.27	0.15	1.375
6	0.03	0.285	0.145	1.42
7	0.025	0.33	0.145	1.43
8	0.035	0.3	0.165	1.4
9	0.04	0.315	0.18	1.385
10	0.03	0.27	0.145	1.36
11	0.035	0.335	0.145	1.4
12	0.03	0.31	0.155	1.615
13	0.035	0.345	0.155	1.405
14	0.035	0.29	0.145	1.365
15	0.03	0.29	0.14	1.375
16	0.025	0.36	0.14	1.39
17	0.04	0.27	0.16	1.385
18	0.035	0.27	0.14	1.38
19	0.04	0.325	0.145	1.64
20	0.03	0.35	0.145	1.375

Table A.1: Execution time of AES 128 using Single threaded C on single core CPU

Single threaded C for AES 128		
	k=10	
	Data Size= 32000 bytes	Data Size= 32000*5 bytes
Average	0.3015	1.41025
Standard Deviation	0.029961	0.076734
Confidence Interval	0.288369 - 0.314631	1.376621 - 1.443879

Table A.2: Calculations for AES 128 using Single threaded C on single core CPU

Single threaded C for AES 192				
	Data Size=32000		Data Size=160000	
S.No	**k=1**	**k=10**	**k=1**	**k=10**
1	0.03	0.335	0.18	1.635
2	0.035	0.335	0.16	1.65
3	0.035	0.385	0.17	1.62
4	0.035	0.4	0.195	1.71
5	0.04	0.34	0.175	1.61
6	0.04	0.32	0.2	1.66
7	0.035	0.32	0.205	1.655
8	0.035	0.355	0.16	1.62
9	0.045	0.32	0.22	1.755
10	0.035	0.335	0.18	1.695
11	0.03	0.33	0.17	1.665
12	0.05	0.35	0.165	1.71
13	0.04	0.325	0.165	1.615
14	0.035	0.32	0.16	1.635
15	0.045	0.365	0.185	1.635
16	0.03	0.335	0.165	1.83
17	0.035	0.355	0.165	1.64
18	0.035	0.335	0.165	1.76
19	0.035	0.335	0.156	1.625
20	0.04	0.37	0.165	1.615

Table A.3: Execution time of AES 192 using Single threaded C on single core CPU.

Single threaded C for AES 192		
	k=10	
	Data Size= 32000 bytes	Data Size= 32000*5 bytes
Average	0.34325	1.667
Standard Deviation	0.022436	0.059192
Confidence Interval	0.333417 - 0.353083	1.641058 - 1.692942

Table A.4: Calculations for AES 192 using Single threaded C on single core CPU

Single threaded C for AES 256				
	Data Size=32000		Data Size=160000	
S.No	**k=1**	**k=10**	**k=1**	**k=10**
1	0.046	0.452	0.218	2.001
2	0.046	0.421	0.219	2.116
3	0.046	0.436	0.211	2.043
4	0.046	0.358	0.194	1.903
5	0.046	0.436	0.22	1.825
6	0.055	0.374	0.214	1.854
7	0.062	0.452	0.187	1.809
8	0.047	0.374	0.219	1.918
9	0.046	0.433	0.218	1.809
10	0.046	0.358	0.218	1.794
11	0.062	0.436	0.234	1.95
12	0.046	0.421	0.187	1.95
13	0.062	0.421	0.202	1.825
14	0.062	0.421	0.218	1.872
15	0.046	0.374	0.194	1.805
16	0.046	0.436	0.187	1.929
17	0.062	0.386	0.202	1.947
18	0.046	0.428	0.218	1.931
19	0.062	0.387	0.187	1.923
20	0.062	0.387	0.187	1.923

Table A.5: Execution time of AES 256 using Single threaded C on single core CPU

Single threaded C for AES 256		
	k=10	
	Data Size= 32000 bytes	Data Size= 32000*5 bytes
Average	0.40955	1.90635
Standard Deviation	0.0312	0.085185
Confidence Interval	0.395876 – 0.423224	1.869017 - 1.943683

Table A.6: Calculations for AES 256 using Single threaded C on single core CPU

CUDA for AES 128 Data Size=32000 Granularity 1 (Total threads used=2000) k=1							
Grid Dimension							
S.No	<20,100>	<100,20>	<50,40>	<40,50>	<1000,2>	<2,1000>	<2000,1>
1	0.005	0	0.005	0.005	0.02	0.002	0.035
2	0.005	0.005	0.005	0	0.015	0.003	0.035
3	0.005	0.005	0	0.005	0.015	0.005	0.03
4	0	0	0	0	0.015	0	0.035
5	0	0.005	0.005	0	0.02	0	0.35
6	0	0	0.005	0	0.015	0	0.03
7	0	0.005	0.005	0	0.015	0	0.03
8	0.005	0.005	0.005	0	0.015	0	0.03
9	0.005	0.005	0	0	0.015	0	0.035
10	0	0	0	0	0.015	0.005	0.03
11	0	0	0	0.005	0.015	0	0.03
12	0	0.005	0.005	0.005	0.015	0	0.03
13	0.005	0.005	0.005	0	0.015	0.005	0.035
14	0	0.005	0	0	0.015	0	0.03
15	0	0	0.005	0	0.015	0.005	0.035
16	0.005	0.005	0	0	0.015	0	0.03
17	0.005	0.005	0.005	0.005	0.015	0.005	0.03
18	0	0	0	0.005	0.015	0.005	0.03
19	0	0.005	0	0	0.015	0.005	0.035
20	0	0.005	0.005	0.005	0.015	0.005	0.035

Table A.7.: Execution time of AES 128 using CUDA on GPU with granularity 1 (Data size 32000, k=1)

CUDA for AES 128 Data Size=32000 Granularity 1 (Total threads used=2000) k=10							
Grid Dimension							
S.No	<20,100>	<100,20>	<50,40>	<40,50>	<1000,2>	<2,1000>	<2000,1>
1	0.03	0.035	0.025	0.025	0.16	0.025	0.31
2	0.03	0.035	0.03	0.025	0.165	0.025	0.31
3	0.035	0.04	0.03	0.025	0.16	0.025	0.31
4	0.03	0.035	0.03	0.025	0.16	0.025	0.31
5	0.035	0.04	0.03	0.025	0.16	0.025	0.31
6	0.03	0.04	0.03	0.025	0.16	0.02	0.315
7	0.03	0.035	0.025	0.02	0.16	0.025	0.31
8	0.03	0.035	0.025	0.02	0.16	0.03	0.31
9	0.03	0.04	0.03	0.025	0.16	0.025	0.31
10	0.03	0.04	0.03	0.025	0.16	0.025	0.31
11	0.03	0.04	0.03	0.02	0.16	0.025	0.31
12	0.03	0.035	0.03	0.025	0.155	0.025	0.31
13	0.035	0.035	0.03	0.025	0.16	0.03	0.305
14	0.03	0.04	0.03	0.025	0.16	0.025	0.31
15	0.03	0.04	0.025	0.025	0.16	0.025	0.305
16	0.035	0.035	0.03	0.025	0.16	0.025	0.31
17	0.035	0.035	0.03	0.025	0.16	0.025	0.31
18	0.035	0.04	0.025	0.025	0.165	0.03	0.31
19	0.03	0.04	0.03	0.025	0.16	0.03	0.31
20	0.03	0.035	0.03	0.02	0.16	0.025	0.315

Table A.8: Execution time of AES 128 using CUDA on GPU with granularity 1 (Data size 32000, k=10)

	AES 128 on GPU using CUDA Data Size = 32000 Granularity 1						
	k=10						
Grid Dimension	**<20,100>**	**<100,20>**	**<50,40>**	**<40,50>**	**<1000,2>**	**<2,1000>**	**<2000,1>**
Average	0.0315	0.0375	0.02875	0.024	0.16025	0.02575	0.31
Standard Deviation	0.002351	0.002565	0.002221	0.002052	0.00197	0.002447	0.002294
Confidence Interval	0.03047 - 0.03253	0.036376 - 0.038624	0.027776 - 0.029724	0.023101 - 0.024899	0.159387 - 0.161113	0.024678 - 0.026822	0.308995 - 0.311005

Table A.9: Calculations for AES 128 using CUDA on GPU with granularity 1 (Data size 32000, k=10)

CUDA for AES 128 Data Size=32000 Granularity 2 (Total threads used=1000) k=1							
Grid Dimension							
S.No	<10,100>	<100,10>	<50,20>	<20,50>	<40,25>	<25,40>	<1000,1>
1	0	0.005	0.005	0	0.002	0.005	0.025
2	0	0.005	0.005	0.005	0.003	0	0.03
3	0.005	0	0.005	0.005	0	0	0.025
4	0.005	0	0	0.005	0.005	0	0.03
5	0	0.005	0.005	0.005	0.005	0	0.03
6	0	0.005	0.005	0	0	0	0.03
7	0	0.005	0.005	0.005	0	0.005	0.025
8	0.005	0	0	0.005	0	0	0.03
9	0.005	0.005	0	0.005	0	0.005	0.025
10	0.005	0.005	0	0	0	0	0.025
11	0	0.005	0.005	0	0	0.005	0.025
12	0	0.005	0	0	0	0	0.03
13	0	0.005	0.005	0	0.005	0	0.03
14	0	0.005	0	0	0	0	0.03
15	0.005	0.005	0.005	0	0.005	0	0.025
16	0	0.005	0.005	0.005	0.005	0.005	0.025
17	0	0.005	0.005	0	0	0	0.03
18	0	0.005	0	0	0	0.005	0.03
19	0.005	0.005	0	0	0	0	0.025
20	0.005	0.005	0	0.005	0.005	0	0.025

Table A.10: Execution time of AES 128 using CUDA on GPU with granularity 2 (Data size 32000, k=1).

CUDA for AES 128 Data Size=32000 Granularity 2 (Total threads used=1000) k=10							
Grid Dimension							
S.No	<10,100>	<100,10>	<50,20>	<20,50>	<40,25>	<25,40>	<1000,1>
1	0.025	0.04	0.02	0.025	0.025	0.02	0.28
2	0.025	0.04	0.025	0.02	0.025	0.02	0.275
3	0.025	0.04	0.025	0.025	0.025	0.02	0.28
4	0.025	0.035	0.02	0.025	0.025	0.025	0.28
5	0.025	0.04	0.025	0.025	0.02	0.02	0.275
6	0.025	0.04	0.025	0.02	0.025	0.02	0.085
7	0.02	0.04	0.025	0.02	0.02	0.025	0.28
8	0.025	0.04	0.025	0.025	0.025	0.025	0.28
9	0.02	0.04	0.025	0.025	0.02	0.02	0.285
10	0.025	0.045	0.02	0.02	0.02	0.025	0.28
11	0.025	0.045	0.025	0.02	0.02	0.02	0.275
12	0.025	0.04	0.02	0.02	0.025	0.02	0.28
13	0.025	0.04	0.02	0.02	0.025	0.025	0.28
14	0.025	0.045	0.02	0.025	0.025	0.02	0.28
15	0.02	0.04	0.025	0.02	0.02	0.02	0.28
16	0.025	0.045	0.025	0.025	0.02	0.025	0.28
17	0.02	0.045	0.025	0.025	0.025	0.025	0.28
18	0.025	0.04	0.02	0.02	0.025	0.02	0.28
19	0.025	0.045	0.025	0.025	0.02	0.02	0.28
20	0.02	0.04	0.025	0.025	0.025	0.02	0.28

Table A.11: Execution time of AES 128 using CUDA on GPU with granularity 2 (Data size 32000, k=10)

	AES 128 on GPU using CUDA Data Size = 32000 Granularity 2						
	k=10						
Grid Dimension	**<10,100>**	**<100,10>**	**<50,20>**	**<20,50>**	**<40,25>**	**<25,40>**	**<1000,1>**
Average	0.02375	0.04125	0.02325	0.02275	0.023	0.02175	0.26975
Standard Deviation	0.002221	0.002751	0.002447	0.002552	0.002513	0.002447	0.043543
Confidence Interval	0.022776 - 0.024724	0.040045 - 0.042445	0.022178 - 0.024322	0.021632 - 0.023868	0.021899 - 0.024101	0.022822 - 0.022822	0.250667 - 0.288833

Table A.12: Calculations for AES 128 using CUDA on GPU with granularity 2 (Data size 32000, k=10)

CUDA for AES 128 Data Size=32000 Granularity 10 (Total threads used=200) k=1							
Grid Dimension							
S.No	<20,10>	<10,20>	<8,25>	<25,8>	<2,100>	<100,2>	<200,1>
1	0.01	0.01	0.005	0.005	0.005	0.015	0.025
2	0.01	0.005	0.01	0.01	0.01	0.015	0.03
3	0.01	0.01	0.01	0.01	0.005	0.015	0.03
4	0.01	0.005	0.01	0.01	0.005	0.015	0.03
5	0.01	0.01	0.005	0.01	0.005	0.015	0.03
6	0.01	0.005	0.005	0.005	0.005	0.015	0.03
7	0.01	0.005	0.005	0.01	0.01	0.015	0.03
8	0.01	0.01	0.005	0.01	0.005	0.015	0.025
9	0.005	0.01	0.005	0.005	0.005	0.015	0.03
10	0.005	0.005	0.005	0.01	0.005	0.015	0.03
11	0.01	0.01	0.005	0.01	0.005	0.015	0.03
12	0.005	0.005	0.005	0.005	0.005	0.015	0.03
13	0.005	0.01	0.005	0.01	0.005	0.015	0.025
14	0.005	0.005	0.01	0.01	0.005	0.015	0.03
15	0.01	0.01	0.005	0.01	0.01	0.015	0.03
16	0.01	0.01	0.005	0.01	0.005	0.02	0.03
17	0.005	0.005	0.005	0.01	0.005	0.015	0.03
18	0.01	0.01	0.01	0.01	0.01	0.015	0.03
19	0.01	0.005	0.005	0.01	0.01	0.015	0.025
20	0.01	0.01	0.011	0.01	0.01	0.015	0.025

Table A.13: Execution time of AES 128 using CUDA on GPU with granularity 10 (Data size 32000, k=1)

CUDA for AES 128 Data Size=32000 Granularity 10 (Total threads used=200) k=10							
Grid Dimension							
S.No	<20,10>	<10,20>	<8,25>	<25,8>	<2,100>	<100,2>	<200,1>
1	0.08	0.075	0.075	0.08	0.07	0.15	0.29
2	0.08	0.075	0.075	0.08	0.07	0.15	0.29
3	0.08	0.075	0.07	0.08	0.07	0.15	0.29
4	0.08	0.07	0.075	0.08	0.07	0.155	0.29
5	0.08	0.075	0.075	0.08	0.07	0.155	0.29
6	0.08	0.07	0.07	0.08	0.07	0.155	0.29
7	0.08	0.075	0.075	0.08	0.07	0.155	0.29
8	0.08	0.07	0.075	0.08	0.07	0.155	0.29
9	0.08	0.075	0.075	0.08	0.07	0.155	0.29
10	0.08	0.07	0.075	0.08	0.075	0.155	0.29
11	0.08	0.07	0.075	0.08	0.075	0.155	0.29
12	0.08	0.075	0.075	0.08	0.075	0.155	0.29
13	0.08	0.07	0.075	0.08	0.075	0.155	0.29
14	0.08	0.075	0.08	0.08	0.075	0.155	0.29
15	0.08	0.07	0.075	0.08	0.075	0.155	0.29
16	0.08	0.07	0.07	0.08	0.075	0.155	0.295
17	0.08	0.075	0.075	0.08	0.075	0.155	0.295
18	0.08	0.07	0.08	0.075	0.075	0.155	0.295
19	0.08	0.075	0.075	0.08	0.075	0.16	0.295
20	0.075	0.075	0.075	0.08	0.075	0.16	0.295

Table A.14: Execution time of AES 128 using CUDA on GPU with granularity 10 (Data size 32000, k=10)

	AES 128 on GPU using CUDA Data Size = 32000 Granularity 10						
	k=10						
Grid Dimension	**<20,10>**	**<10,20>**	**<8,25>**	**<25,8>**	**<2,100>**	**<100,2>**	**<200,1>**
Average	0.07975	0.07275	0.07475	0.07975	0.07275	0.15475	0.29125
Standard Deviation	0.001118	0.002552	0.002552	0.001118	0.002552	0.002552	0.002221
Confidence Interval	0.07926 - 0.08024	0.071632 - 0.073868	0.073632 - 0.075868	0.07926 - 0.08024	0.071632 - 0.073868	0.153632 - 0.155868	0.290276 - 0.290276

Table A.15: Calculations for AES 128 using CUDA on GPU with granularity 10 (Data size 32000, k=10)

CUDA for AES 128 Data Size=32000 Granularity 100 (Total threads used=20) k=1							
Grid Dimension							
S.No	<2,10>	<10,2>	<5,4>	<4,5>	<20,1>	<1,20>	<2,10>
1	0.055	0.06	0.058	0.055	0.06	0.06	0.055
2	0.055	0.06	0.057	0.055	0.06	0.06	0.055
3	0.055	0.06	0.06	0.055	0.06	0.06	0.055
4	0.055	0.06	0.06	0.055	0.06	0.06	0.055
5	0.055	0.06	0.06	0.055	0.06	0.06	0.055
6	0.055	0.06	0.06	0.055	0.06	0.06	0.055
7	0.055	0.06	0.06	0.055	0.06	0.06	0.055
8	0.055	0.06	0.06	0.055	0.06	0.06	0.055
9	0.055	0.06	0.06	0.055	0.06	0.06	0.055
10	0.06	0.06	0.06	0.055	0.06	0.06	0.06
11	0.06	0.06	0.06	0.055	0.06	0.06	0.06
12	0.06	0.06	0.06	0.055	0.06	0.06	0.06
13	0.06	0.06	0.06	0.055	0.06	0.06	0.06
14	0.06	0.06	0.055	0.06	0.06	0.06	0.06
15	0.06	0.06	0.055	0.06	0.06	0.06	0.06
16	0.06	0.055	0.055	0.06	0.065	0.055	0.06
17	0.06	0.055	0.055	0.06	0.065	0.055	0.06
18	0.06	0.055	0.055	0.06	0.065	0.055	0.06
19	0.06	0.055	0.055	0.06	0.065	0.055	0.06
20	0.06	0.055	0.055	0.06	0.065	0.055	0.06

Table A.16: Execution time of AES 128 using CUDA on GPU with granularity 100 (Data size 32000, k=1)

CUDA for AES 128 Data Size=32000 Granularity 100 (Total threads used=20) k=10							
Grid Dimension							
S.No	<2,10>	<10,2>	<5,4>	<4,5>	<20,1>	<1,20>	<2,10>
1	0.575	0.575	0.57	0.565	0.618	0.59	0.575
2	0.575	0.575	0.57	0.565	0.615	0.59	0.575
3	0.575	0.575	0.57	0.565	0.615	0.59	0.575
4	0.575	0.575	0.57	0.565	0.615	0.59	0.575
5	0.575	0.575	0.57	0.565	0.615	0.59	0.575
6	0.575	0.575	0.57	0.565	0.615	0.59	0.575
7	0.575	0.575	0.57	0.565	0.615	0.59	0.575
8	0.575	0.575	0.57	0.565	0.615	0.59	0.575
9	0.575	0.575	0.57	0.565	0.615	0.59	0.575
10	0.575	0.575	0.57	0.57	0.615	0.59	0.575
11	0.575	0.575	0.57	0.57	0.615	0.59	0.575
12	0.575	0.575	0.57	0.57	0.615	0.585	0.575
13	0.575	0.575	0.57	0.57	0.615	0.585	0.575
14	0.575	0.58	0.57	0.57	0.62	0.585	0.575
15	0.575	0.58	0.57	0.57	0.62	0.585	0.575
16	0.575	0.58	0.57	0.57	0.62	0.585	0.575
17	0.58	0.58	0.57	0.57	0.62	0.585	0.58
18	0.58	0.58	0.57	0.57	0.62	0.585	0.58
19	0.57	0.58	0.565	0.57	0.62	0.585	0.57
20	0.57	0.58	0.575	0.565	0.62	0.585	0.57

Table A.17: Execution time of AES 128 using CUDA on GPU with granularity 100 (Data size 32000, k=10).

	AES 128 on GPU using CUDA Data Size = 32000 Granularity 100					
	k=10					
Grid Dimension	**<2,10>**	**<10,2>**	**<5,4>**	**<4,5>**	**<20,1>**	**<1,20>**
Average	0.575	0.5675	0.57	0.57675	0.6169	0.58775
Standard Deviation	0.002294	0.002565	0.001622	0.002447	0.002426	0.002552
Confidence Interval	0.573995 - 0.576005	0.566376 - 0.568624	0.569289 - 0.570711	0.575678 - 0.577822	0.615837 - 0.617963	0.586632 - 0.588868

Table A.18: Calculations for AES 128 using CUDA on GPU with granularity 100 (Data size 32000, k=10)

CUDA for AES 128 Data Size=32000*5 Granularity 1 (Total threads used=2000) k=1							
	Grid Dimension						
S.No	<20,100>	<100,20>	<50,40>	<40,50>	<1000,2>	<2,1000>	<2000,1>
1	0.015	0.015	0.02	0.015	0.005	0.02	0.155
2	0.015	0.015	0.02	0.015	0.005	0.02	0.155
3	0.015	0.02	0.02	0.015	0.08	0.02	0.155
4	0.015	0.02	0.015	0.015	0.08	0.02	0.155
5	0.015	0.02	0.015	0.015	0.08	0.02	0.155
6	0.015	0.02	0.015	0.015	0.08	0.015	0.155
7	0.015	0.02	0.015	0.015	0.08	0.015	0.155
8	0.015	0.02	0.015	0.015	0.08	0.015	0.155
9	0.01	0.02	0.015	0.01	0.08	0.015	0.155
10	0.01	0.02	0.015	0.01	0.08	0.015	0.155
11	0.01	0.02	0.015	0.01	0.08	0.015	0.16
12	0.01	0.02	0.015	0.01	0.08	0.015	0.16
13	0.015	0.02	0.015	0.015	0.08	0.015	0.16
14	0.015	0.02	0.015	0.015	0.08	0.015	0.16
15	0.015	0.02	0.015	0.015	0.08	0.015	0.16
16	0.015	0.02	0.015	0.015	0.08	0.015	0.16
17	0.015	0.025	0.015	0.015	0.08	0.015	0.16
18	0.015	0.015	0.015	0.015	0.085	0.015	0.155
19	0.015	0.02	0.015	0.01	0.085	0.02	0.155
20	0.015	0.02	0.01	0.02	0.085	0.02	0.155

Table A.19: Execution time of AES 128 using CUDA on GPU with granularity 1 (Data size 32000*5, k=1)

CUDA for AES 128 Data Size=32000*5 Granularity 1 (Total threads used=2000) k=10							
	Grid Dimension						
S.No	<20,100>	<100,20>	<50,40>	<40,50>	<1000,2>	<2,1000>	<2000,1>
1	0.135	0.195	0.155	0.135	0.81	0.16	1.565
2	0.135	0.195	0.155	0.135	0.81	0.16	1.565
3	0.135	0.195	0.155	0.135	0.81	0.16	1.565
4	0.135	0.195	0.155	0.135	0.81	0.16	1.565
5	0.135	0.195	0.15	0.135	0.81	0.16	1.565
6	0.135	0.195	0.15	0.135	0.81	0.165	1.565
7	0.135	0.195	0.15	0.135	0.81	0.165	1.565
8	0.135	0.19	0.15	0.135	0.81	0.165	1.565
9	0.135	0.19	0.15	0.135	0.81	0.165	1.565
10	0.135	0.19	0.15	0.135	0.81	0.165	1.565
11	0.13	0.19	0.15	0.14	0.81	0.165	1.565
12	0.13	0.19	0.15	0.14	0.805	0.165	1.565
13	0.13	0.19	0.15	0.14	0.805	0.165	1.56
14	0.13	0.19	0.15	0.14	0.805	0.165	1.56
15	0.13	0.19	0.145	0.14	0.805	0.165	1.56
16	0.13	0.205	0.145	0.14	0.815	0.165	1.56
17	0.14	0.2	0.145	0.14	0.815	0.165	1.57
18	0.14	0.2	0.145	0.14	0.815	0.17	1.57
19	0.14	0.2	0.16	0.125	0.82	0.17	1.57
20	0.145	0.2	0.16	0.125	0.82	0.155	1.57

Table A.20: Execution time of AES 128 using CUDA on GPU with granularity 1 (Data size 32000*5, k=10)

	AES 128 on GPU using CUDA Data Size = 32000*5 Granularity 1						
	k=10						
Grid Dimension	**<20,100>**	**<100,20>**	**<50,40>**	**<40,50>**	**<1000,2>**	**<2,1000>**	**<2000,1>**
Average	0.13475	0.1945	0.151	0.136	0.81075	0.16375	1.565
Standard Deviation	0.004128	0.00456	0.004472	0.004472	0.004375	0.003582	0.003244
Confidence Interval	0.132941 - 0.136559	0.192502 - 0.196498	0.149040 - 0.152960	0.134040 - 0.137960	0.808832 - 0.812668	0.162180 - 0.165320	1.563578 - 1.566422

Table A.21: Calculations for AES 128 using CUDA on GPU with granularity 1 (Data size 32000*5)

CUDA for AES 128 Data Size=32000*5 Granularity 2 (Total threads used=1000) k=1							
Grid Dimension							
S.No	<10,100>	<100,10>	<50,20>	<20,50>	<40,25>	<25,40>	<1000,1>
1	0.015	0.020	0.010	0.010	0.010	0.012	0.140
2	0.010	0.020	0.010	0.015	0.015	0.012	0.140
3	0.015	0.020	0.015	0.015	0.015	0.015	0.145
4	0.015	0.025	0.010	0.015	0.010	0.015	0.145
5	0.010	0.020	0.010	0.015	0.015	0.010	0.140
6	0.015	0.020	0.010	0.010	0.020	0.015	0.140
7	0.010	0.020	0.015	0.010	0.015	0.015	0.140
8	0.010	0.020	0.015	0.015	0.010	0.015	0.140
9	0.010	0.025	0.010	0.010	0.010	0.015	0.140
10	0.010	0.025	0.010	0.015	0.015	0.010	0.140
11	0.015	0.020	0.010	0.010	0.010	0.015	0.140
12	0.015	0.025	0.010	0.010	0.015	0.015	0.140
13	0.015	0.020	0.010	0.015	0.010	0.010	0.140
14	0.015	0.020	0.010	0.015	0.015	0.015	0.140
15	0.010	0.020	0.015	0.010	0.015	0.015	0.145
16	0.015	0.020	0.015	0.010	0.010	0.015	0.140
17	0.015	0.020	0.015	0.015	0.015	0.010	0.140
18	0.010	0.020	0.010	0.015	0.015	0.010	0.140
19	0.010	0.020	0.010	0.015	0.010	0.015	0.140
20	0.010	0.020	0.015	0.015	0.010	0.015	0.140

Table A.22: Execution time of AES 128 using CUDA on GPU with granularity 2 (Data size 32000*5, k=1)

CUDA for AES 128 Data Size=32000*5 Granularity 2 (Total threads used=1000) k=10							
Grid Dimension							
S.No	<10,100>	<100,10>	<50,20>	<20,50>	<40,25>	<25,40>	<1000,1>
1	0.117	0.215	0.130	0.125	0.125	0.120	1.395
2	0.122	0.215	0.120	0.125	0.120	0.130	1.405
3	0.125	0.215	0.125	0.125	0.130	0.130	1.390
4	0.125	0.125	0.120	0.130	0.120	0.125	1.390
5	0.125	0.210	0.125	0.120	0.130	0.125	1.390
6	0.125	0.210	0.125	0.125	0.125	0.125	1.395
7	0.125	0.225	0.135	0.130	0.120	0.130	1.390
8	0.130	0.210	0.135	0.125	0.115	0.120	1.390
9	0.120	0.210	0.135	0.130	0.125	0.130	1.400
10	0.120	0.210	0.120	0.120	0.125	0.125	1.400
11	0.115	0.215	0.135	0.120	0.130	0.135	1.400
12	0.120	0.210	0.125	0.125	0.130	0.125	1.390
13	0.120	0.215	0.135	0.130	0.120	0.120	1.390
14	0.125	0.215	0.135	0.130	0.115	0.130	1.405
15	0.125	0.210	0.130	0.125	0.125	0.120	1.405
16	0.120	0.220	0.135	0.120	0.120	0.135	1.410
17	0.125	0.220	0.135	0.120	0.125	0.125	1.405
18	0.130	0.215	0.135	0.125	0.125	0.130	1.395
19	0.130	0.215	0.130	0.120	0.120	0.140	1.405
20	0.130	0.205	0.135	0.120	0.130	0.130	1.405

Table A.23: Execution time of AES 128 using CUDA on GPU with granularity 2 (Data size 32000*5, k=10)

	AES 128 on GPU using CUDA Data Size = 32000*5 Granularity 2						
	k=10						
Grid Dimension	**<10,100>**	**<100,10>**	**<50,20>**	**<20,50>**	**<40,25>**	**<25,40>**	**<1000,1>**
Average	0.1237	0.20925	0.13	0.1245	0.12375	0.1275	1.39775
Standard Deviation	0.004378	0.020344	0.005849	0.00394	0.004833	0.005501	0.006973
Confidence Interval	0.121781 - 0.125619	0.200334 - 0.218166	0.127437 - 0.132563	0.122773 - 0.126227	0.121632 - 0.125868	0.125089 - 0.129911	1.394694 - 0.144806

Table A.24: Calculations for AES 128 using CUDA on GPU with granularity 2 (Data size 32000*5, k=10)

CUDA for AES 128 Data Size=32000*5 Granularity 10 (Total threads used=200) k=1							
Grid Dimension							
S.No	<20,10>	<10,20>	<8,25>	<25,8>	<2,100>	<100,2>	<200,1>
1	0.039	0.035	0.040	0.040	0.040	0.075	0.150
2	0.038	0.035	0.040	0.040	0.035	0.080	0.145
3	0.040	0.040	0.035	0.040	0.040	0.080	0.145
4	0.040	0.035	0.040	0.040	0.040	0.080	0.145
5	0.040	0.035	0.035	0.045	0.035	0.075	0.150
6	0.040	0.035	0.035	0.040	0.040	0.080	0.145
7	0.040	0.035	0.040	0.045	0.035	0.080	0.145
8	0.040	0.035	0.035	0.040	0.035	0.080	0.150
9	0.035	0.040	0.040	0.040	0.035	0.080	0.150
10	0.035	0.035	0.035	0.040	0.040	0.080	0.145
11	0.040	0.035	0.040	0.040	0.035	0.075	0.145
12	0.040	0.035	0.035	0.040	0.035	0.075	0.150
13	0.040	0.04	0.040	0.045	0.040	0.075	0.145
14	0.040	0.035	0.035	0.040	0.040	0.080	0.150
15	0.040	0.035	0.040	0.040	0.035	0.080	0.150
16	0.040	0.040	0.040	0.040	0.040	0.075	0.145
17	0.040	0.035	0.035	0.040	0.040	0.080	0.145
18	0.040	0.035	0.035	0.045	0.035	0.075	0.145
19	0.040	0.035	0.035	0.050	0.040	0.075	0.150
20	0.040	0.035	0.035	0.040	0.040	0.080	0.150

Table A.25: Execution time of AES 128 using CUDA on GPU with granularity 10 (Data size 32000*5, k=1)

CUDA for AES 128 Data Size=32000*5 Granularity 10 (Total threads used=200) k=10							
Grid Dimension							
S.No	<20,10>	<10,20>	<8,25>	<25,8>	<2,100>	<100,2>	<200,1>
1	0.365	0.385	0.355	0.400	0.360	0.765	1.470
2	0.365	0.385	0.360	0.400	0.365	0.775	1.460
3	0.365	0.380	0.355	0.405	0.360	0.765	1.470
4	0.365	0.380	0.355	0.405	0.365	0.770	1.465
5	0.365	0.385	0.365	0.395	0.365	0.76	1.465
6	0.370	0.380	0.365	0.400	0.365	0.775	1.470
7	0.370	0.385	0.360	0.405	0.360	0.770	1.470
8	0.370	0.385	0.365	0.395	0.360	0.755	1.465
9	0.370	0.380	0.360	0.400	0.365	0.755	1.465
10	0.370	0.390	0.360	0.400	0.365	0.770	1.465
11	0.360	0.390	0.355	0.395	0.365	0.770	1.470
12	0.360	0.395	0.355	0.405	0.365	0.755	1.470
13	0.360	0.395	0.365	0.410	0.360	0.770	1.470
14	0.360	0.390	0.360	0.400	0.355	0.755	1.465
15	0.360	0.395	0.355	0.405	0.370	0.770	1.465
16	0.360	0.390	0.365	0.400	0.365	0.770	1.465
17	0.360	0.390	0.365	0.400	0.365	0.755	1.470
18	0.360	0.395	0.360	0.405	0.365	0.770	1.465
19	0.360	0.395	0.360	0.400	0.370	0.770	1.465
20	0.360	0.395	0.365	0.405	0.365	0.770	1.465

Table A.26: Execution time of AES 128 using CUDA on GPU with granularity 10 (Data size 32000*5, k=10)

	AES 128 on GPU using CUDA Data Size = 32000*5 Granularity 10						
	k=10						
Grid Dimension	**<20,10>**	**<10,20>**	**<8,25>**	**<25,8>**	**<2,100>**	**<100,2>**	**<200,1>**
Average	0.36375	0.38775	0.35975	0.4015	0.36375	0.77075	1.46675
Standard Deviation	0.004253	0.005495	0.004128	0.004007	0.003582	0.004064	0.002936
Confidence Interval	0.361886 - 0.356614	0.385342 - 0.390158	0.357941 - 0.361559	0.399744 - 0.403256	0.362180 - 0.365320	0.768969 - 0.772531	1.465463 - 1.468037

Table A.27: Calculations for AES 128 using CUDA on GPU with granularity 10 (Data size 32000*5)

CUDA for AES 128 Data Size=32000*5 Granularity 100 (Total threads used=20) k=1						
Grid Dimension						
S.No	<2,10>	<10,2>	<5,4>	<4,5>	<20,1>	<1,20>
1	0.285	0.29	0.285	0.285	0.305	0.295
2	0.285	0.285	0.285	0.285	0.305	0.295
3	0.285	0.285	0.285	0.285	0.305	0.295
4	0.285	0.285	0.285	0.285	0.305	0.295
5	0.285	0.285	0.285	0.285	0.305	0.295
6	0.285	0.285	0.29	0.285	0.305	0.295
7	0.29	0.285	0.29	0.285	0.305	0.295
8	0.29	0.285	0.29	0.285	0.305	0.295
9	0.29	0.285	0.29	0.285	0.305	0.295
10	0.29	0.285	0.29	0.285	0.305	0.295
11	0.29	0.285	0.29	0.285	0.31	0.295
12	0.29	0.285	0.29	0.285	0.31	0.295
13	0.29	0.285	0.29	0.285	0.315	0.295
14	0.29	0.285	0.285	0.285	0.315	0.295
15	0.285	0.285	0.285	0.285	0.315	0.295
16	0.285	0.285	0.29	0.285	0.315	0.295
17	0.285	0.285	0.29	0.285	0.315	0.295
18	0.295	0.285	0.29	0.285	0.315	0.295
19	0.285	0.285	0.29	0.285	0.315	0.295
20	0.295	0.285	0.29	0.285	0.315	0.295

Table A.28: Execution time of AES 128 using CUDA on GPU with granularity 100 (Data size 32000*5, k=1)

CUDA for AES 128 Data Size=32000*5 Granularity 100 (Total threads used=20) k=10						
	Grid Dimension					
S.No	<2,10>	<10,2>	<5,4>	<4,5>	<20,1>	<1,20>
1	2.875	2.84	2.885	2.865	3.08	2.945
2	2.875	2.84	2.9	2.865	3.08	2.945
3	2.875	2.84	2.9	2.865	3.08	2.945
4	2.875	2.84	2.89	2.855	3.09	2.945
5	2.875	2.84	2.89	2.855	3.09	2.945
6	2.875	2.84	2.89	2.86	3.09	2.945
7	2.875	2.84	2.89	2.86	3.09	2.945
8	2.875	2.84	2.89	2.86	3.095	2.945
9	2.88	2.84	2.89	2.86	3.095	2.945
10	2.88	2.84	2.89	2.86	3.085	2.95
11	2.88	2.84	2.89	2.86	3.085	2.94
12	2.88	2.845	2.89	2.86	3.085	2.94
13	2.88	2.845	2.89	2.86	3.085	2.94
14	2.88	2.845	2.89	2.86	3.085	2.94
15	2.88	2.845	2.89	2.86	3.085	2.94
16	2.88	2.845	2.89	2.86	3.085	2.94
17	2.89	2.845	2.89	2.86	3.085	2.94
18	2.888	2.845	2.89	2.86	3.085	2.94
19	2.888	2.845	2.89	2.86	3.085	2.94
20	2.88	2.845	2.89	2.86	3.085	2.94

Table A.29: Execution time of AES 128 using CUDA on GPU with granularity 100 (Data size 32000*5, k=10)

	AES 128 on GPU using CUDA Data Size = 32000*5 Granularity 100					
	k=10					
Grid Dimension	**<2,10>**	**<10,2>**	**<5,4>**	**<4,5>**	**<20,1>**	**<1,20>**
Average	2.8793	2.84225	2.89075	2.86025	3.08625	2.94275
Standard Deviation	0.004692	0.002552	0.003354	0.002552	0.004253	0.003024
Confidence Interval	2.877244 - 2.877244	2.841132 - 2.843368	2.889280 - 2.892220	2.859132 - 2.861368	3.084386 - 3.088140	2.941425 - 2.944075

Table A.30: Calculations for AES 128 using CUDA on GPU with granularity 100 (Data size 32000*5, k=10)

CUDA for AES 192
Data Size=32000
Granularity 1 (Total threads used=2000)
k=1

Grid Dimension							
S.No	<20,100>	<100,20>	<50,40>	<40,50>	<1000,2>	<2,1000>	<2000,1>
1	0.005	0.005	0	0.005	0.020	0.005	0.045
2	0.005	0.005	0.005	0	0.020	0	0.040
3	0	0.005	0.005	0	0.020	0.005	0.040
4	0.005	0.005	0.005	0.005	0.020	0.005	0.040
5	0.005	0.005	0.005	0	0.020	0.005	0.040
6	0.005	0.005	0.005	0.005	0.025	0.005	0.040
7	0.005	0.005	0.005	0.005	0.020	0.005	0.040
8	0.005	0.010	0.005	0.005	0.020	0.005	0.045
9	0.005	0.005	0.005	0.005	0.020	0.005	0.045
10	0.005	0.005	0.005	0.005	0.020	0.005	0.040
11	0	0.005	0.005	0.005	0.025	0.005	0.040
12	0.005	0.005	0.005	0	0.020	0.005	0.040
13	0.005	0.005	0.005	0	0.025	0.005	0.040
14	0.005	0.005	0.005	0.005	0.020	0.005	0.040
15	0	0.005	0.005	0.005	0.020	0.005	0.045
16	0.005	0.005	0.005	0.005	0.025	0.005	0.040
17	0.005	0.005	0.005	0	0.020	0	0.040
18	0.005	0.005	0.005	0.005	0.020	0	0.040
19	0	0.005	0.005	0.005	0.020	0.005	0.040
20	0.005	0.005	0.005	0.005	0.020	0.005	0.040

Table A.31: Execution time of AES 192 using CUDA on GPU with granularity 1 (Data size 32000, k=1)

CUDA for AES 192
Data Size=32000
Granularity 1 (Total threads used=2000)
k=10

Grid Dimension							
S.No	<20,100>	<100,20>	<50,40>	<40,50>	<1000,2>	<2,1000>	<2000,1>
1	0.035	0.049	0.04	0.03	0.21	0.035	0.045
2	0.035	0.047	0.04	0.03	0.21	0.035	0.045
3	0.035	0.045	0.04	0.03	0.21	0.035	0.045
4	0.035	0.045	0.04	0.03	0.21	0.035	0.045
5	0.035	0.045	0.04	0.03	0.21	0.04	0.045
6	0.03	0.045	0.04	0.035	0.205	0.04	0.045
7	0.03	0.045	0.04	0.035	0.205	0.04	0.045
8	0.03	0.045	0.035	0.035	0.205	0.04	0.045
9	0.03	0.045	0.035	0.035	0.205	0.04	0.045
10	0.03	0.045	0.035	0.035	0.205	0.04	0.045
11	0.03	0.05	0.035	0.035	0.205	0.04	0.045
12	0.03	0.05	0.035	0.035	0.205	0.04	0.045
13	0.03	0.05	0.035	0.035	0.205	0.04	0.045
14	0.03	0.05	0.035	0.035	0.205	0.04	0.045
15	0.03	0.05	0.035	0.035	0.205	0.04	0.045
16	0.03	0.05	0.035	0.035	0.205	0.04	0.045
17	0.03	0.05	0.035	0.035	0.205	0.04	0.045
18	0.03	0.05	0.035	0.035	0.205	0.04	0.045
19	0.03	0.05	0.035	0.035	0.205	0.04	0.045
20	0.03	0.05	0.035	0.035	0.205	0.04	0.045

Table A.32: Execution time of AES 192 using CUDA on GPU with granularity 1 (Data size 32000, k=10)

	AES 192 on GPU using CUDA Data Size = 32000 Granularity 1						
	k=10						
Grid Dimension	**<20,100>**	**<100,20>**	**<50,40>**	**<40,50>**	**<1000,2>**	**<2,1000>**	**<2000,1>**
Average	0.033750	0.047800	0.036750	0.03125	0.206250	0.039000	0.045000
Standard Deviation	0.002221	0.002441	0.002447	0.002221	0.002221	0.002052	1.4238E-17
Confidence Interval	0.032776 - 0.034724	0.046730 - 0.048870	0.035678 - 0.037822	0.030276 - 0.032224	0.205276 - 0.207224	0.038101 - 0.039899	0.045000 - 0.045000

Table A.33: Calculations for AES 192 using CUDA on GPU with granularity 1 (Data size 32000, k=10)

CUDA for AES 192 Data Size=32000 Granularity 2 (Total threads used=1000) k=1							
Grid Dimension							
S.No	<10,100>	<100,10>	<50,20>	<20,50>	<40,25>	<25,40>	<1000,1>
1	0	0.005	0.005	0	0.005	0.005	0.035
2	0.005	0.005	0.005	0	0.005	0	0.035
3	0.005	0.005	0.005	0	0.005	0.005	0.035
4	0.005	0.005	0.005	0	0	0	0.035
5	0.005	0.005	0.005	0.005	0.005	0	0.035
6	0	0.005	0.005	0	0.005	0	0.035
7	0.005	0.005	0	0.005	0.005	0	0.035
8	0	0.005	0.005	0	0.005	0	0.035
9	0.005	0.005	0	0.005	0.005	0.005	0.035
10	0.005	0.005	0	0.005	0	0	0.035
11	0	0.005	0.005	0	0	0.005	0.035
12	0	0.005	0.005	0	0.005	0	0.04
13	0	0.005	0	0.005	0.005	0	0.035
14	0.005	0.005	0.005	0	0.005	0.005	0.035
15	0.005	0.005	0	0.005	0.005	0	0.035
16	0	0.005	0.005	0	0.005	0.005	0.035
17	0	0.005	0	0.005	0.005	0	0.035
18	0.005	0.005	0.005	0	0.005	0	0.035
19	0	0.005	0.005	0	0	0.005	0.035
20	0	0.005	0.005	0.005	0.005	0	0.035

Table A.34: Execution time of AES 192 using CUDA on GPU with granularity 2 (Data size 32000, k=1)

CUDA for AES 192 Data Size=32000 Granularity 2 (Total threads used=1000) k=10							
Grid Dimension							
S.No	<10,100>	<100,10>	<50,20>	<20,50>	<40,25>	<25,40>	<1000,1>
1	0.025	0.05	0.03	0.025	0.03	0.025	0.345
2	0.025	0.05	0.025	0.025	0.025	0.03	0.345
3	0.03	0.05	0.03	0.025	0.025	0.03	0.35
4	0.025	0.05	0.03	0.025	0.025	0.03	0.35
5	0.035	0.05	0.025	0.03	0.025	0.03	0.35
6	0.03	0.05	0.03	0.025	0.025	0.025	0.35
7	0.025	0.05	0.025	0.025	0.025	0.025	0.35
8	0.025	0.05	0.025	0.025	0.03	0.025	0.35
9	0.025	0.055	0.03	0.025	0.03	0.025	0.35
10	0.025	0.05	0.025	0.025	0.025	0.025	0.35
11	0.025	0.05	0.03	0.03	0.025	0.025	0.35
12	0.025	0.055	0.03	0.025	0.025	0.025	0.345
13	0.025	0.05	0.03	0.025	0.025	0.025	0.345
14	0.03	0.055	0.03	0.025	0.025	0.025	0.345
15	0.025	0.05	0.025	0.025	0.03	0.025	0.35
16	0.03	0.05	0.025	0.025	0.025	0.025	0.35
17	0.025	0.05	0.03	0.025	0.035	0.03	0.345
18	0.03	0.05	0.03	0.025	0.025	0.025	0.35
19	0.025	0.05	0.025	0.03	0.03	0.025	0.35
20	0.025	0.05	0.025	0.03	0.025	0.025	0.35

Table A.35: Execution time of AES 192 using CUDA on GPU with granularity 2 (Data size 32000, k=10)

	AES 192 on GPU using CUDA Data Size = 32000 Granularity 2						
	k=10						
Grid Dimension	**<10,100>**	**<100,10>**	**<50,20>**	**<20,50>**	**<40,25>**	**<25,40>**	**<1000,1>**
Average	0.02675	0.05075	0.02775	0.02625	0.026	0.02675	0.3485
Standard Deviation	0.002936	0.001832	0.002552	0.002221	0.002052	0.002936	0.002350
Confidence Interval	0.025463 - 0.028037	0.049947 - 0.051553	0.026632 - 0.028868	0.025276 - 0.027224	0.025101 - 0.026899	0.025463 - 0.028037	0.347469 - 0.034953

Table A.36: Calculations for AES 192 using CUDA on GPU with granularity 2 (Data size 32000, k=10)

CUDA for AES 192 Data Size=32000 Granularity 10 (Total threads used=200) k=1							
Grid Dimension							
S.No	<20,10>	<10,20>	<8,25>	<25,8>	<2,100>	<100,2>	<200,1>
1	0.01	0.005	0.005	0.015	0.005	0.02	0.04
2	0.01	0.005	0.005	0.01	0.01	0.02	0.04
3	0.01	0.01	0.005	0.01	0.01	0.02	0.04
4	0.01	0.01	0.005	0.01	0.01	0.02	0.04
5	0.01	0.01	0.01	0.01	0.01	0.02	0.04
6	0.01	0.01	0.01	0.01	0.01	0.02	0.04
7	0.01	0.01	0.01	0.01	0.01	0.02	0.035
8	0.01	0.01	0.01	0.01	0.01	0.02	0.035
9	0.01	0.01	0.01	0.01	0.01	0.02	0.035
10	0.01	0.01	0.01	0.01	0.01	0.02	0.035
11	0.01	0.01	0.01	0.01	0.01	0.02	0.035
12	0.01	0.01	0.01	0.01	0.01	0.02	0.035
13	0.01	0.01	0.01	0.01	0.01	0.02	0.035
14	0.01	0.01	0.01	0.01	0.01	0.02	0.035
15	0.01	0.01	0.01	0.01	0.01	0.02	0.035
16	0.01	0.01	0.01	0.01	0.01	0.015	0.035
17	0.01	0.01	0.01	0.01	0.01	0.025	0.035
18	0.01	0.01	0.01	0.01	0.01	0.025	0.035
19	0.01	0.01	0.01	0.01	0.01	0.025	0.035
20	0.01	0.01	0.01	0.01	0.01	0.025	0.035

Table A.37: Execution time of AES 192 using CUDA on GPU with granularity 10 (Data size 32000, k=1)

CUDA for AES 192 Data Size=32000 Granularity 10 (Total threads used=200) k=10							
Grid Dimension							
S.No	<20,10>	<10,20>	<8,25>	<25,8>	<2,100>	<100,2>	<200,1>
1	0.09	0.085	0.09	0.1	0.085	0.19	0.36
2	0.09	0.085	0.085	0.1	0.085	0.19	0.36
3	0.09	0.085	0.085	0.1	0.085	0.19	0.36
4	0.09	0.085	0.085	0.1	0.085	0.19	0.36
5	0.09	0.085	0.085	0.1	0.085	0.19	0.36
6	0.09	0.085	0.085	0.1	0.085	0.19	0.36
7	0.09	0.085	0.085	0.1	0.09	0.19	0.36
8	0.095	0.085	0.085	0.1	0.09	0.19	0.36
9	0.095	0.085	0.085	0.095	0.09	0.19	0.36
10	0.095	0.085	0.085	0.095	0.09	0.19	0.36
11	0.095	0.09	0.085	0.095	0.09	0.19	0.36
12	0.095	0.09	0.085	0.095	0.085	0.19	0.36
13	0.095	0.09	0.085	0.095	0.09	0.19	0.36
14	0.095	0.09	0.085	0.095	0.085	0.19	0.36
15	0.095	0.09	0.085	0.095	0.085	0.19	0.36
16	0.095	0.09	0.085	0.095	0.085	0.19	0.36
17	0.095	0.09	0.085	0.095	0.09	0.185	0.36
18	0.095	0.09	0.085	0.095	0.09	0.185	0.36
19	0.095	0.09	0.08	0.095	0.085	0.185	0.355
20	0.095	0.09	0.08	0.095	0.085	0.185	0.355

Table A.38: Execution time of AES 192 using CUDA on GPU with granularity 10 (Data size 32000, k=10)

	AES 192 on GPU using CUDA Data Size = 32000 Granularity 10						
	k=10						
Grid Dimension	**<20,10>**	**<10,20>**	**<8,25>**	**<25,8>**	**<2,100>**	**<100,2>**	**<200,1>**
Average	0.093250	0.087500	0.084750	0.097000	0.087000	0.189000	0.359500
Standard Deviation	0.002447	0.002565	0.001970	0.002513	0.002513	0.002052	0.001538
Confidence Interval	0.092178 - 0.094322	0.086376 - 0.088624	0.083887 - 0.085613	0.095899 - 0.098101	0.085899 - 0.088101	0.188101 - 0.0189899	0.358825 - 0.360174

Table A.39: Calculations for AES 192 using CUDA on GPU with granularity 10 (Data size 32000, k=10)

CUDA for AES 192 Data Size=32000 Granularity 100 (Total threads used=20) k=1						
Grid Dimension						
S.No	<2,10>	<10,2>	<5,4>	<4,5>	<20,1>	<1,20>
1	0.075	0.075	0.07	0.07	0.08	0.07
2	0.07	0.075	0.07	0.07	0.08	0.07
3	0.07	0.07	0.07	0.07	0.08	0.07
4	0.07	0.07	0.07	0.07	0.08	0.07
5	0.07	0.07	0.07	0.07	0.08	0.07
6	0.07	0.07	0.07	0.07	0.075	0.07
7	0.07	0.07	0.07	0.07	0.075	0.07
8	0.07	0.07	0.07	0.07	0.075	0.07
9	0.07	0.07	0.07	0.07	0.075	0.07
10	0.07	0.07	0.07	0.07	0.075	0.07
11	0.07	0.07	0.07	0.07	0.075	0.07
12	0.07	0.07	0.07	0.07	0.075	0.07
13	0.07	0.07	0.07	0.07	0.075	0.07
14	0.07	0.07	0.07	0.07	0.075	0.07
15	0.07	0.07	0.07	0.07	0.075	0.075
16	0.07	0.07	0.07	0.07	0.075	0.075
17	0.07	0.07	0.07	0.07	0.075	0.075
18	0.07	0.07	0.07	0.07	0.075	0.075
19	0.07	0.07	0.07	0.065	0.075	0.075
20	0.07	0.07	0.07	0.065	0.075	0.075

Table A.40: Execution time of AES 192 using CUDA on GPU with granularity 100 (Data size 32000, k=1)

CUDA for AES 192 Data Size=32000 Granularity 100 (Total threads used=20) k=10						
	Grid Dimension					
S.No	<2,10>	<10,2>	<5,4>	<4,5>	<20,1>	<1,20>
1	0.7	0.705	0.695	0.695	0.75	0.71
2	0.7	0.705	0.695	0.695	0.75	0.71
3	0.7	0.705	0.695	0.695	0.75	0.71
4	0.7	0.705	0.695	0.695	0.75	0.71
5	0.7	0.705	0.695	0.695	0.75	0.71
6	0.7	0.705	0.695	0.695	0.75	0.71
7	0.7	0.705	0.695	0.695	0.75	0.71
8	0.7	0.7	0.695	0.695	0.75	0.71
9	0.695	0.7	0.695	0.695	0.75	0.71
10	0.695	0.7	0.695	0.695	0.75	0.71
11	0.695	0.7	0.695	0.695	0.75	0.71
12	0.695	0.7	0.695	0.695	0.75	0.715
13	0.695	0.7	0.695	0.695	0.75	0.715
14	0.695	0.7	0.695	0.695	0.75	0.715
15	0.695	0.7	0.695	0.695	0.745	0.715
16	0.695	0.7	0.695	0.69	0.745	0.715
17	0.72	0.7	0.7	0.69	0.745	0.72
18	0.72	0.7	0.7	0.07	0.745	0.715
19	0.72	0.75	0.7	0.07	0.755	0.72
20	0.72	0.75	0.7	0.07	0.755	0.72

Table A.41: Execution time of AES 192 using CUDA on GPU with granularity 100 (Data size 32000, k=10)

	AES 192 on GPU using CUDA Data Size = 32000 Granularity 100					
	k=10					
Grid Dimension	**<2,10>**	**<10,2>**	**<5,4>**	**<4,5>**	**<20,1>**	**<1,20>**
Average	0.702000	0.706750	0.696000	0.600750	0.749500	0.713000
Standard Deviation	0.009515	0.014980	0.002052	0.228757	0.002763	0.003770
Confidence Interval	0.697830 - 0.070617	0.700185 - 0.713315	0.695101 - 0.696899	0.500495 - 0.701005	0.748289 - 0.750711	0.711348 - 0.714652

Table A.42: Calculations for AES 192 using CUDA on GPU with granularity 100 (Data size 32000, k=10)

CUDA for AES 192
Data Size=32000*5
Granularity 1 (Total threads used=2000)
k=1

	Grid Dimension						
S.No	<20,100>	<100,20>	<50,40>	<40,50>	<1000,2>	<2,1000>	<2000,1>
1	0.02	0.02	0.015	0.02	0.11	0.025	0.2
2	0.02	0.025	0.015	0.02	0.11	0.025	0.2
3	0.02	0.025	0.015	0.02	0.11	0.025	0.2
4	0.02	0.025	0.02	0.02	0.11	0.02	0.2
5	0.02	0.025	0.02	0.02	0.11	0.02	0.2
6	0.02	0.025	0.02	0.02	0.105	0.02	0.2
7	0.015	0.025	0.02	0.02	0.105	0.02	0.205
8	0.015	0.025	0.02	0.02	0.105	0.02	0.205
9	0.015	0.025	0.02	0.02	0.105	0.02	0.205
10	0.015	0.025	0.02	0.015	0.105	0.02	0.205
11	0.015	0.025	0.02	0.015	0.105	0.02	0.205
12	0.015	0.025	0.02	0.015	0.105	0.02	0.205
13	0.015	0.025	0.02	0.015	0.105	0.02	0.205
14	0.015	0.025	0.02	0.015	0.105	0.02	0.205
15	0.015	0.025	0.02	0.015	0.105	0.02	0.205
16	0.015	0.025	0.02	0.015	0.105	0.02	0.205
17	0.015	0.025	0.02	0.015	0.105	0.02	0.205
18	0.015	0.025	0.02	0.015	0.105	0.02	0.205
19	0.015	0.025	0.02	0.015	0.105	0.02	0.205
20	0.015	0.025	0.02	0.015	0.105	0.02	0.205

Table A.43: Execution time of AES 192 using CUDA on GPU with granularity 1 (Data size 32000*5, k=1)

CUDA for AES 192
Data Size=32000*5
Granularity 1 (Total threads used=2000)
k=10

	Grid Dimension						
S.No	<20,100>	<100,20>	<50,40>	<40,50>	<1000,2>	<2,1000>	<2000,1>
1	0.16	0.225	0.185	0.165	1.055	0.205	2.025
2	0.16	0.225	0.185	0.165	1.045	0.205	2.025
3	0.16	0.24	0.185	0.165	1.045	0.205	2.025
4	0.165	0.24	0.185	0.165	1.045	0.205	2.025
5	0.165	0.24	0.185	0.165	1.045	0.205	2.025
6	0.165	0.24	0.185	0.165	1.045	0.205	2.02
7	0.165	0.24	0.175	0.165	1.045	0.205	2.02
8	0.165	0.24	0.175	0.16	1.045	0.205	2.02
9	0.165	0.24	0.195	0.175	1.045	0.205	2.02
10	0.155	0.24	0.195	0.175	1.045	0.21	2.02
11	0.155	0.24	0.195	0.175	1.04	0.21	2.02
12	0.15	0.24	0.195	0.175	1.04	0.21	2.02
13	0.15	0.245	0.19	0.17	1.04	0.21	2.02
14	0.17	0.245	0.19	0.17	1.04	0.21	2.02
15	0.17	0.235	0.19	0.17	1.04	0.195	2.02
16	0.17	0.235	0.19	0.17	1.04	0.195	2.02
17	0.17	0.235	0.19	0.17	1.04	0.215	2.02
18	0.17	0.235	0.19	0.17	1.04	0.215	2.02
19	0.17	0.235	0.19	0.17	1.04	0.215	2.02
20	0.17	0.235	0.19	0.17	1.04	0.215	2.02

Table A.44: Execution time of AES 192 using CUDA on GPU with granularity 1 (Data size 32000*5, k=10)

	AES 192 on GPU using CUDA Data Size = 32000*5 Granularity 1						
	k=10						
Grid Dimension	**<20,100>**	**<100,20>**	**<50,40>**	**<40,50>**	**<1000,2>**	**<2,1000>**	**<2000,1>**
Average	0.1635	0.2375	0.188	0.16875	1.043	0.20725	2.02125
Standard Deviation	0.006708	0.005257	0.005712	0.004253	0.00377	0.00573	0.002221
Confidence Interval	0.16056 - 0.16644	0.235196 - 0.239804	0.185496 - 0.190504	0.166886 - 0.170614	1.041348 - 1.044652	0.204739 - 0.209761	2.020276 - 2.022224

Table A.45: Calculations for AES 192 using CUDA on GPU with granularity 1 (Data size 32000*5, k=10)

CUDA for AES 192 Data Size=32000*5 Granularity 2 (Total threads used=1000) k=1							
Grid Dimension							
S.No	<10,100>	<100,10>	<50,20>	<20,50>	<40,25>	<25,40>	<1000,1>
1	0.015	0.030	0.015	0.010	0.010	0.015	0.175
2	0.015	0.030	0.015	0.015	0.015	0.015	0.175
3	0.015	0.025	0.015	0.015	0.015	0.015	0.175
4	0.010	0.025	0.015	0.010	0.015	0.015	0.175
5	0.015	0.025	0.015	0.015	0.015	0.015	0.175
6	0.015	0.025	0.015	0.015	0.015	0.015	0.175
7	0.015	0.025	0.015	0.015	0.015	0.015	0.175
8	0.015	0.025	0.015	0.015	0.015	0.015	0.175
9	0.015	0.025	0.020	0.015	0.015	0.010	0.175
10	0.015	0.025	0.015	0.015	0.015	0.015	0.175
11	0.015	0.025	0.015	0.015	0.015	0.015	0.175
12	0.015	0.030	0.015	0.015	0.015	0.015	0.175
13	0.015	0.030	0.015	0.015	0.015	0.015	0.175
14	0.010	0.025	0.015	0.015	0.015	0.015	0.175
15	0.015	0.025	0.020	0.015	0.010	0.015	0.175
16	0.015	0.030	0.015	0.015	0.015	0.015	0.180
17	0.015	0.025	0.015	0.015	0.015	0.015	0.175
18	0.015	0.025	0.015	0.010	0.015	0.015	0.175
19	0.015	0.025	0.015	0.015	0.015	0.015	0.175
20	0.015	0.030	0.015	0.015	0.015	0.015	0.175

Table A.46: Execution time of AES 192 using CUDA on GPU with granularity 2 (Data size 32000*5, k=1)

CUDA for AES 192 Data Size=32000*5 Granularity 2 (Total threads used=1000) k=10							
Grid Dimension							
S.No	<10,100>	<100,10>	<50,20>	<20,50>	<40,25>	<25,40>	<1000,1>
1	0.140	0.260	0.140	0.150	0.140	0.145	1.745
2	0.150	0.260	0.150	0.140	0.135	0.140	1.750
3	0.135	0.260	0.150	0.140	0.135	0.135	1.740
4	0.150	0.260	0.140	0.155	0.140	0.140	1.740
5	0.140	0.255	0.155	0.150	0.140	0.140	1.745
6	0.150	0.255	0.145	0.145	0.140	0.145	1.745
7	0.150	0.255	0.145	0.135	0.125	0.150	1.740
8	0.145	0.245	0.145	0.140	0.145	0.145	1.750
9	0.140	0.260	0.145	0.140	0.150	0.140	1.750
10	0.140	0.260	0.135	0.140	0.145	0.155	1.745
11	0.150	0.260	0.145	0.140	0.145	0.145	1.750
12	0.135	0.260	0.145	0.140	0.150	0.140	1.740
13	0.150	0.255	0.140	0.135	0.145	0.140	1.740
14	0.140	0.255	0.135	0.140	0.145	0.135	1.745
15	0.150	0.255	0.140	0.130	0.145	0.145	1.745
16	0.150	0.245	0.145	0.145	0.145	0.150	1.740
17	0.145	0.260	0.140	0.150	0.145	0.150	1.750
18	0.140	0.260	0.150	0.140	0.155	0.135	1.750
19	0.140	0.260	0.140	0.145	0.145	0.155	1.745
20	0.150	0.260	0.155	0.145	0.140	0.150	1.750

Table A.47: Execution time of AES 192 using CUDA on GPU with granularity 2 (Data size 32000*5, k=10)

	AES 192 on GPU using CUDA Data Size = 32000*5 Granularity 2						
	k=10						
Grid Dimension	**<10,100>**	**<100,10>**	**<50,20>**	**<20,50>**	**<40,25>**	**<25,40>**	**<1000,1>**
Average	0.144500	0.257000	0.144250	0.142250	0.142750	0.144000	1.74525
Standard Deviation	0.005596	0.004702	0.005684	0.005955	0.006382	0.006198	0.004128
Confidence Interval	0.142047 - 0.146953	0.254939 - 0.259061	0.141759 - 0.146741	0.139640 - 0.144860	0.139953 - 0.145547	0.141283 - 0.146717	1.743441 - 1.747059

Table A.48: Calculations for AES 192 using CUDA on GPU with granularity 2 (Data size 32000*5, k=10)

CUDA for AES 192
Data Size=32000*5
Granularity 10 (Total threads used=200)
k=1

	Grid Dimension						
S.No	<20,10>	<10,20>	<8,25>	<25,8>	<2,100>	<100,2>	<200,1>
1	0.045	0.05	0.04	0.045	0.04	0.1	0.185
2	0.045	0.05	0.04	0.045	0.04	0.1	0.185
3	0.045	0.04	0.04	0.045	0.04	0.09	0.185
4	0.045	0.04	0.045	0.05	0.04	0.095	0.185
5	0.045	0.04	0.045	0.05	0.05	0.095	0.185
6	0.045	0.045	0.045	0.05	0.045	0.095	0.18
7	0.045	0.045	0.045	0.05	0.045	0.095	0.18
8	0.045	0.045	0.045	0.05	0.045	0.095	0.18
9	0.045	0.045	0.045	0.05	0.045	0.095	0.18
10	0.045	0.045	0.045	0.05	0.045	0.095	0.18
11	0.05	0.045	0.045	0.05	0.045	0.095	0.18
12	0.05	0.045	0.045	0.05	0.045	0.095	0.18
13	0.05	0.045	0.045	0.05	0.045	0.095	0.18
14	0.05	0.045	0.045	0.05	0.045	0.095	0.18
15	0.05	0.045	0.045	0.05	0.045	0.095	0.18
16	0.05	0.045	0.045	0.05	0.045	0.095	0.18
17	0.05	0.045	0.045	0.05	0.045	0.095	0.18
18	0.05	0.045	0.045	0.05	0.045	0.095	0.18
19	0.05	0.045	0.045	0.05	0.045	0.095	0.18
20	0.05	0.045	0.045	0.05	0.045	0.095	0.18

Table A.49 : Execution time of AES 192 using CUDA on GPU with granularity 10 (Data size 32000*5, k=1)

CUDA for AES 192
Data Size=32000*5
Granularity 10 (Total threads used=200)
k=10

	Grid Dimension						
S.No	<20,10>	<10,20>	<8,25>	<25,8>	<2,100>	<100,2>	<200,1>
1	0.465	0.45	0.435	0.475	0.445	0.945	1.79
2	0.465	0.475	0.435	0.48	0.445	0.945	1.8
3	0.465	0.435	0.435	0.48	0.445	0.945	1.8
4	0.465	0.41	0.435	0.48	0.44	0.945	1.8
5	0.465	0.43	0.435	0.485	0.44	0.945	1.8
6	0.465	0.44	0.435	0.485	0.44	0.945	1.8
7	0.465	0.44	0.435	0.485	0.44	0.945	1.8
8	0.465	0.44	0.42	0.485	0.44	0.945	1.8
9	0.47	0.445	0.42	0.485	0.44	0.945	1.8
10	0.47	0.44	0.425	0.485	0.44	0.945	1.8
11	0.47	0.44	0.43	0.49	0.44	0.945	1.8
12	0.475	0.44	0.43	0.49	0.44	0.935	1.8
13	0.46	0.44	0.43	0.49	0.44	0.95	1.8
14	0.46	0.44	0.43	0.49	0.44	0.95	1.8
15	0.46	0.43	0.43	0.49	0.44	0.94	1.8
16	0.46	0.435	0.43	0.49	0.435	0.94	1.8
17	0.46	0.435	0.43	0.49	0.435	0.94	1.8
18	0.46	0.44	0.43	0.49	0.435	0.94	1.8
19	0.46	0.435	0.43	0.49	0.435	0.94	1.8
20	0.46	0.435	0.43	0.49	0.435	0.94	1.8

Table A.50: Execution time of AES 192 using CUDA on GPU with granularity 10 (Data size 32000*5, k=10)

	AES 192 on GPU using CUDA Data Size = 32000*5 Granularity 10						
	k=10						
Grid Dimension	**<20,10>**	**<10,20>**	**<8,25>**	**<25,8>**	**<2,100>**	**<100,2>**	**<200,1>**
Average	0.464250	0.438750	0.430500	0.486250	0.439500	0.943500	1.799500
Standard Deviation	0.004375	0.011571	0.004560	0.004552	0.003204	0.003663	0.002236
Confidence Interval	0.462332 - 0.466168	0.433679 - 0.443821	0.428502 - 0.432498	0.484255 - 0.488245	0.438096 - 0.440904	0.941894 - 0.945106	1.79852 - 1.800480

Table A.51: Calculations for AES 192 using CUDA on GPU with granularity 10 (Data size 32000*5, k=10)

CUDA for AES 192 Data Size=32000*5 Granularity 100 (Total threads used=20) k=1						
Grid Dimension						
S.No	<2,10>	<10,2>	<5,4>	<4,5>	<20,1>	<1,20>
1	0.35	0.35	0.345	0.345	0.375	0.355
2	0.35	0.35	0.345	0.345	0.375	0.355
3	0.35	0.35	0.345	0.345	0.375	0.355
4	0.35	0.35	0.345	0.345	0.375	0.355
5	0.35	0.35	0.345	0.345	0.375	0.355
6	0.35	0.35	0.345	0.345	0.375	0.355
7	0.35	0.35	0.345	0.345	0.375	0.355
8	0.35	0.35	0.345	0.345	0.375	0.355
9	0.35	0.35	0.35	0.35	0.375	0.355
10	0.35	0.35	0.35	0.35	0.375	0.355
11	0.35	0.35	0.35	0.35	0.375	0.355
12	0.35	0.35	0.35	0.35	0.375	0.355
13	0.35	0.35	0.35	0.35	0.375	0.355
14	0.35	0.35	0.35	0.35	0.375	0.355
15	0.35	0.35	0.35	0.35	0.375	0.355
16	0.35	0.35	0.35	0.35	0.375	0.355
17	0.35	0.355	0.35	0.35	0.375	0.355
18	0.35	0.355	0.35	0.35	0.37	0.36
19	0.345	0.355	0.35	0.35	0.38	0.36
20	0.345	0.355	0.35	0.35	0.38	0.36

Table A.52: Execution time of AES 192 using CUDA on GPU with granularity 100 (Data size 32000*5, k=1)

CUDA for AES 192 Data Size=32000*5 Granularity 100 (Total threads used=20) k=10						
Grid Dimension						
S.No	<2,10>	<10,2>	<5,4>	<4,5>	<20,1>	<1,20>
1	3.485	3.505	3.48	3.48	3.745	3.55
2	3.49	3.505	3.48	3.52	3.745	3.55
3	3.49	3.505	3.48	3.47	3.745	3.55
4	3.49	3.505	3.48	3.47	3.745	3.55
5	3.49	3.505	3.48	3.47	3.745	3.55
6	3.49	3.505	3.48	3.47	3.745	3.55
7	3.49	3.505	3.48	3.47	3.745	3.55
8	3.49	3.505	3.48	3.47	3.745	3.55
9	3.49	3.515	3.48	3.47	3.745	3.55
10	3.49	3.51	3.48	3.47	3.745	3.555
11	3.49	3.51	3.48	3.47	3.745	3.555
12	3.49	3.51	3.48	3.47	3.75	3.555
13	3.49	3.51	3.485	3.47	3.75	3.555
14	3.49	3.51	3.475	3.475	3.75	3.555
15	3.49	3.51	3.475	3.475	3.75	3.555
16	3.49	3.51	3.475	3.475	3.75	3.555
17	3.49	3.51	3.475	3.475	3.75	3.555
18	3.49	3.51	3.475	3.475	3.75	3.555
19	3.49	3.51	3.475	3.475	3.75	3.555
20	3.49	3.51	3.475	3.475	3.74	3.555

Table A.53: Execution time of AES 192 using CUDA on GPU with granularity 100 (Data size 32000*5, k=10)

	AES 192 on GPU using CUDA Data Size = 32000*5 Granularity 100					
	k=10					
Grid Dimension	**<2,10>**	**<10,2>**	**<5,4>**	**<4,5>**	**<20,1>**	**<1,20>**
Average	3.48975	3.50825	3.4785	3.47475	3.74675	3.55275
Standard Deviation	0.001118	0.002936	0.002856	0.011059	0.002936	0.002552
Confidence Interval	3.48926 - 3.49024	3.506963 - 3.509537	3.477248 - 3.49752	3.469903 - 3.479597	3.745463 - 3.748037	3.551632 - 3.553868

Table A.54: Execution time of AES 192 using CUDA on GPU with granularity 100 (Data size 32000*5, k=10)

CUDA for AES 256 Data Size=32000 Granularity 1 (Total threads used=2000) k=1							
			Grid Dimension				
S.No	<20,100>	<100,20>	<50,40>	<40,50>	<1000,2>	<2,1000>	<2000,1>
1	0.005	0.005	0	0.005	0.025	0.005	0.05
2	0.005	0.005	0	0.005	0.025	0.005	0.05
3	0.005	0.005	0	0.005	0.025	0.005	0.05
4	0.005	0.005	0.005	0.005	0.025	0.005	0.05
5	0.005	0.005	0.005	0.005	0.025	0.005	0.05
6	0.01	0.005	0.005	0.005	0.025	0.005	0.05
7	0.005	0.005	0.005	0.005	0.025	0.005	0.05
8	0.005	0.005	0.005	0.005	0.025	0.005	0.05
9	0.005	0.005	0.005	0.005	0.025	0.005	0.05
10	0.005	0.005	0.005	0.005	0.025	0.005	0.05
11	0.01	0.005	0.005	0.005	0.025	0.005	0.05
12	0.005	0.005	0.005	0.005	0.025	0.005	0.045
13	0.005	0.005	0.005	0.005	0.025	0.005	0.045
14	0.005	0.005	0.005	0.005	0.025	0.005	0.045
15	0.005	0.005	0.005	0.005	0.025	0.005	0.045
16	0.005	0.005	0.005	0.005	0.025	0.005	0.045
17	0.005	0.005	0.005	0	0.02	0.005	0.045
18	0.005	0.01	0.005	0	0.02	0.005	0.045
19	0.005	0.01	0.005	0	0.02	0.005	0.045
20	0.005	0.01	0.005	0	0.02	0.01	0.045

Table A.55: Execution time of AES 256 using CUDA on GPU with granularity 1 (Data size 32000, k=1)

CUDA for AES 256 Data Size=32000 Granularity 1 (Total threads used=2000) k=10							
			Grid Dimension				
S.No	<20,100>	<100,20>	<50,40>	<40,50>	<1000,2>	<2,1000>	<2000,1>
1	0.04	0.055	0.035	0.04	0.24	0.045	0.465
2	0.04	0.055	0.035	0.04	0.24	0.045	0.465
3	0.04	0.055	0.035	0.04	0.24	0.045	0.465
4	0.035	0.055	0.035	0.04	0.24	0.045	0.465
5	0.035	0.055	0.035	0.04	0.24	0.045	0.465
6	0.035	0.055	0.035	0.04	0.24	0.045	0.465
7	0.035	0.055	0.035	0.04	0.24	0.045	0.465
8	0.035	0.055	0.035	0.04	0.24	0.045	0.465
9	0.035	0.055	0.035	0.04	0.24	0.045	0.465
10	0.035	0.055	0.035	0.04	0.24	0.045	0.465
11	0.035	0.055	0.035	0.04	0.24	0.045	0.465
12	0.035	0.055	0.035	0.04	0.235	0.045	0.465
13	0.035	0.055	0.035	0.04	0.235	0.045	0.465
14	0.035	0.05	0.035	0.04	0.235	0.045	0.465
15	0.035	0.05	0.03	0.035	0.235	0.045	0.465
16	0.035	0.05	0.03	0.035	0.235	0.045	0.46
17	0.035	0.05	0.04	0.035	0.235	0.04	0.46
18	0.035	0.05	0.04	0.035	0.235	0.04	0.47
19	0.035	0.05	0.04	0.035	0.235	0.04	0.47
20	0.03	0.05	0.04	0.045	0.235	0.04	0.47

Table A.56: Execution time of AES 256 using CUDA on GPU with granularity 1 (Data size 32000, k=10)

	AES 256 on GPU using CUDA Data Size = 32000 Granularity 1						
	k=10						
Grid Dimension	**<20,100>**	**<100,20>**	**<50,40>**	**<40,50>**	**<1000,2>**	**<2,1000>**	**<2000,1>**
Average	0.0355	0.05325	0.0355	0.039	0.23775	0.044	0.46525
Standard Deviation	0.002236	0.002447	0.002763	0.002616	0.002552	0.002052	0.002552
Confidence Interval	0.03452 - 0.036480	0.052178 - 0.054332	0.034289 - 0.036711	0.037854 - 0.040146	0.236632 - 0.238868	0.043101 - 0.044899	0.464132 - 0.466368

Table A.57: Calculations for AES 256 using CUDA on GPU with granularity 1 (Data size 32000, k=10)

CUDA for AES 256 Data Size=32000 Granularity 2 (Total threads used=1000) k=1							
Grid Dimension							
S.No	<10,100>	<100,10>	<50,20>	<20,50>	<40,25>	<25,40>	<1000,1>
1	0.005	0.005	0.005	0.005	0.005	0.005	0.04
2	0	0.005	0.005	0.005	0.005	0	0.04
3	0.005	0.005	0.005	0	0.005	0.005	0.04
4	0.005	0.005	0.005	0.005	0	0	0.04
5	0.005	0.01	0.005	0	0.005	0.005	0.04
6	0.005	0.01	0.005	0.005	0.005	0.005	0.04
7	0.005	0.005	0.005	0.005	0.005	0.005	0.04
8	0.005	0.01	0.005	0.005	0.005	0	0.04
9	0	0.005	0.005	0.005	0.005	0	0.04
10	0.005	0.005	0.005	0.005	0.005	0	0.04
11	0.005	0.005	0.005	0.005	0.005	0.005	0.04
12	0	0.005	0.005	0.005	0.005	0.005	0.04
13	0.005	0.005	0.005	0.005	0.005	0.005	0.04
14	0.005	0.01	0.005	0.005	0.005	0.005	0.04
15	0.005	0.005	0	0	0.005	0.005	0.04
16	0	0.01	0	0.005	0.005	0.005	0.04
17	0.005	0.01	0	0.005	0.005	0.005	0.04
18	0.005	0.005	0	0	0.005	0.005	0.04
19	0	0.005	0.005	0.005	0.005	0.005	0.04
20	0.005	0.005	0.005	0	0.005	0	0.045

Table A.58: Execution time of AES 256 using CUDA on GPU with granularity 2 (Data size 32000, k=1)

CUDA for AES 256 Data Size=32000 Granularity 2 (Total threads used=1000) k=10							
Grid Dimension							
S.No	<10,100>	<100,10>	<50,20>	<20,50>	<40,25>	<25,40>	<1000,1>
1	0.035	0.06	0.035	0.035	0.03	0.03	0.405
2	0.03	0.06	0.03	0.035	0.035	0.035	0.405
3	0.03	0.055	0.03	0.035	0.035	0.03	0.405
4	0.03	0.055	0.035	0.025	0.03	0.03	0.405
5	0.03	0.055	0.035	0.03	0.03	0.03	0.405
6	0.035	0.055	0.03	0.03	0.025	0.03	0.405
7	0.03	0.055	0.03	0.03	0.03	0.03	0.41
8	0.03	0.06	0.03	0.03	0.03	0.03	0.405
9	0.03	0.06	0.03	0.03	0.035	0.035	0.4
10	0.03	0.055	0.03	0.035	0.035	0.035	0.4
11	0.03	0.055	0.03	0.03	0.03	0.03	0.405
12	0.03	0.055	0.03	0.03	0.03	0.03	0.405
13	0.03	0.055	0.03	0.03	0.03	0.035	0.405
14	0.03	0.055	0.03	0.035	0.03	0.035	0.4
15	0.03	0.055	0.03	0.03	0.03	0.035	0.415
16	0.025	0.055	0.03	0.035	0.03	0.035	0.405
17	0.03	0.055	0.03	0.03	0.03	0.03	0.405
18	0.035	0.06	0.03	0.03	0.03	0.035	0.4
19	0.03	0.06	0.03	0.03	0.035	0.03	0.405
20	0.03	0.055	0.03	0.03	0.03	0.035	0.405

Table A.59: Execution time of AES 256 using CUDA on GPU with granularity 2 (Data size 32000, k=10)

	AES 256 on GPU using CUDA Data Size = 32000 Granularity 2						
	k=10						
Grid Dimension	**<10,100>**	**<100,10>**	**<50,20>**	**<20,50>**	**<40,25>**	**<25,40>**	**<1000,1>**
Average	0.0305	0.0565	0.03075	0.03125	0.031000	0.03225	0.40475
Standard Deviation	0.002236	0.002351	0.001832	0.002751	0.002616	0.002552	0.003432
Confidence Interval	0.02952 - 0.031480	0.05547 - 0.057530	0.029947 - 0.031553	0.030045 - 0.032455	0.029854 - 0.032146	0.031132 - 0.033368	0.403246 - 0.406254

Table A.60: Calculations for AES 256 using CUDA on GPU with granularity 2 (Data size 32000, k=10)

CUDA for AES 256 Data Size=32000 Granularity 10 (Total threads used=200) k=1							
			Grid Dimension				
S.No	<20,10>	<10,20>	<8,25>	<25,8>	<2,100>	<100,2>	<200,1>
1	0.01	0.01	0.01	0.015	0.01	0.02	0.04
2	0.01	0.01	0.01	0.015	0.01	0.02	0.04
3	0.01	0.01	0.01	0.015	0.01	0.02	0.04
4	0.01	0.01	0.01	0.015	0.01	0.02	0.04
5	0.01	0.01	0.01	0.015	0.01	0.02	0.04
6	0.01	0.01	0.01	0.015	0.01	0.02	0.04
7	0.01	0.01	0.01	0.015	0.01	0.02	0.04
8	0.01	0.01	0.01	0.01	0.01	0.02	0.04
9	0.01	0.01	0.01	0.01	0.01	0.02	0.04
10	0.01	0.01	0.01	0.01	0.01	0.02	0.04
11	0.01	0.01	0.01	0.01	0.01	0.02	0.04
12	0.01	0.01	0.01	0.01	0.01	0.02	0.04
13	0.01	0.01	0.01	0.01	0.01	0.02	0.04
14	0.01	0.01	0.01	0.01	0.015	0.02	0.045
15	0.01	0.015	0.01	0.01	0.015	0.02	0.045
16	0.015	0.015	0.01	0.01	0.015	0.025	0.045
17	0.015	0.015	0.01	0.01	0.015	0.025	0.045
18	0.015	0.015	0.01	0.01	0.015	0.025	0.045
19	0.015	0.015	0.015	0.01	0.015	0.025	0.045
20	0.015	0.011	0.015	0.01	0.015	0.025	0.045

Table A.61: Execution time of AES 256 using CUDA on GPU with granularity 10 (Data size 32000, k=1)

CUDA for AES 256 Data Size=32000 Granularity 10 (Total threads used=200) k=10							
			Grid Dimension				
S.No	<20,10>	<10,20>	<8,25>	<25,8>	<2,100>	<100,2>	<200,1>
1	0.11	0.1	0.095	0.11	0.1	0.215	0.415
2	0.11	0.1	0.095	0.11	0.1	0.215	0.415
3	0.11	0.1	0.095	0.11	0.1	0.215	0.415
4	0.11	0.1	0.095	0.11	0.1	0.215	0.415
5	0.11	0.1	0.095	0.11	0.1	0.215	0.415
6	0.11	0.1	0.095	0.11	0.1	0.215	0.415
7	0.11	0.1	0.095	0.11	0.1	0.215	0.415
8	0.11	0.1	0.095	0.11	0.1	0.215	0.415
9	0.105	0.1	0.095	0.11	0.1	0.215	0.415
10	0.105	0.1	0.095	0.11	0.1	0.215	0.415
11	0.105	0.1	0.095	0.11	0.1	0.215	0.415
12	0.105	0.1	0.095	0.11	0.1	0.215	0.415
13	0.105	0.1	0.1	0.11	0.1	0.215	0.415
14	0.105	0.1	0.1	0.11	0.1	0.215	0.415
15	0.105	0.1	0.1	0.11	0.095	0.215	0.415
16	0.105	0.1	0.1	0.105	0.095	0.215	0.415
17	0.105	0.1	0.1	0.115	0.095	0.22	0.415
18	0.105	0.095	0.1	0.115	0.095	0.22	0.41
19	0.105	0.095	0.1	0.115	0.095	0.21	0.41
20	0.105	0.095	0.1	0.115	0.095	0.22	0.41

Table A.62: Execution time of AES 256 using CUDA on GPU with granularity 10 (Data size 32000, k=10)

	AES 256 on GPU using CUDA Data Size = 32000 Granularity 10						
	k=10						
Grid Dimension	**<20,10>**	**<10,20>**	**<8,25>**	**<25,8>**	**<2,100>**	**<100,2>**	**<200,1>**
Average	0.107	0.09925	0.097	0.11075	0.0985	0.2155	0.41425
Standard Deviation	0.002513	0.001832	0.002513	0.002447	0.002351	0.002236	0.001832
Confidence Interval	0.105899 - 0.108101	0.098447 - 0.100053	0.095899 - 0.098101	0.109678 - 0.111822	0.09747 - 0.099530	0.21452 - 0.216480	0.413447 - 0.415053

Table A.63: Calculations for AES 256 using CUDA on GPU with granularity 10 (Data size 32000, k=10)

CUDA for AES 256 Data Size=32000 Granularity 100 (Total threads used=20) k=1						
Grid Dimension						
S.No	<2,10>	<10,2>	<5,4>	<4,5>	<20,1>	<1,20>
1	0.08	0.085	0.08	0.08	0.09	0.085
2	0.08	0.085	0.08	0.08	0.09	0.085
3	0.08	0.085	0.08	0.08	0.09	0.085
4	0.08	0.085	0.08	0.08	0.09	0.085
5	0.08	0.085	0.08	0.08	0.09	0.085
6	0.08	0.085	0.08	0.08	0.09	0.085
7	0.08	0.085	0.08	0.08	0.09	0.085
8	0.08	0.085	0.08	0.08	0.09	0.08
9	0.08	0.085	0.08	0.08	0.09	0.08
10	0.08	0.085	0.08	0.08	0.09	0.08
11	0.08	0.085	0.08	0.08	0.085	0.08
12	0.08	0.085	0.08	0.08	0.085	0.08
13	0.08	0.08	0.08	0.08	0.085	0.08
14	0.08	0.08	0.085	0.08	0.085	0.08
15	0.08	0.08	0.085	0.08	0.085	0.08
16	0.085	0.08	0.085	0.08	0.085	0.08
17	0.085	0.08	0.085	0.081	0.085	0.08
18	0.085	0.08	0.085	0.085	0.085	0.08
19	0.085	0.08	0.085	0.085	0.085	0.08
20	0.085	0.08	0.085	0.085	0.085	0.08

Table A.64: Execution time of AES 256 using CUDA on GPU with granularity 100 (Data size 32000, k=1)

CUDA for AES 256 Data Size=32000 Granularity 100 (Total threads used=20) k=10						
	Grid Dimension					
S.No	<2,10>	<10,2>	<5,4>	<4,5>	<20,1>	<1,20>
1	0.8	0.81	0.815	0.805	0.865	0.815
2	0.8	0.81	0.815	0.805	0.865	0.815
3	0.8	0.81	0.815	0.805	0.865	0.815
4	0.8	0.81	0.815	0.805	0.865	0.815
5	0.8	0.81	0.815	0.805	0.865	0.815
6	0.8	0.81	0.815	0.805	0.865	0.815
7	0.81	0.81	0.815	0.805	0.865	0.815
8	0.805	0.81	0.815	0.805	0.865	0.815
9	0.805	0.81	0.815	0.805	0.865	0.815
10	0.805	0.81	0.81	0.805	0.865	0.815
11	0.805	0.81	0.81	0.805	0.87	0.815
12	0.805	0.81	0.81	0.805	0.87	0.815
13	0.805	0.81	0.81	0.805	0.87	0.82
14	0.805	0.81	0.81	0.805	0.87	0.82
15	0.805	0.81	0.81	0.805	0.87	0.82
16	0.805	0.81	0.81	0.81	0.87	0.82
17	0.805	0.81	0.81	0.81	0.87	0.82
18	0.805	0.81	0.81	0.8	0.87	0.82
19	0.805	0.81	0.81	0.8	0.87	0.82
20	0.805	0.815	0.81	0.8	0.875	0.82

Table A.65: Execution time of AES 256 using CUDA on GPU with granularity 100 (Data size 32000, k=10)

	AES 256 on GPU using CUDA Data Size = 32000 Granularity 100					
	k=10					
Grid Dimension	**<2,10>**	**<10,2>**	**<5,4>**	**<4,5>**	**<20,1>**	**<1,20>**
Average	0.80375	0.81025	0.81225	0.80475	0.86775	0.817
Standard Deviation	0.002751	0.001118	0.002552	0.002552	0.003024	0.002513
Confidence Interval	0.802545 - 0.804955	0.80976 - 0.81074	0.811132 - 0.813368	0.803632 - 0.805868	0.866425 - 0.869075	0.815899 - 0.818101

Table A.66: Calculations for AES 256 using CUDA on GPU with granularity 100 (Data size 32000, k=10)

CUDA for AES 256 Data Size=32000*5 Granularity 1 (Total threads used=2000) k=1							
Grid Dimension							
S.No	<20,100>	<100,20>	<50,40>	<40,50>	<1000,2>	<2,1000>	<2000,1>
1	0.02	0.03	0.02	0.02	0.12	0.025	0.235
2	0.02	0.03	0.02	0.02	0.12	0.025	0.235
3	0.02	0.03	0.02	0.02	0.12	0.025	0.235
4	0.02	0.03	0.02	0.02	0.12	0.025	0.235
5	0.02	0.03	0.02	0.02	0.12	0.025	0.235
6	0.02	0.03	0.02	0.02	0.12	0.025	0.235
7	0.02	0.03	0.02	0.02	0.12	0.025	0.235
8	0.02	0.03	0.02	0.02	0.12	0.025	0.235
9	0.02	0.03	0.02	0.02	0.12	0.025	0.235
10	0.02	0.03	0.02	0.02	0.12	0.025	0.235
11	0.02	0.03	0.02	0.02	0.12	0.025	0.235
12	0.02	0.025	0.02	0.02	0.12	0.02	0.23
13	0.02	0.025	0.02	0.02	0.12	0.02	0.23
14	0.02	0.025	0.02	0.02	0.12	0.02	0.23
15	0.02	0.025	0.025	0.02	0.15	0.02	0.23
16	0.02	0.025	0.025	0.015	0.125	0.02	0.23
17	0.015	0.025	0.025	0.015	0.125	0.02	0.23
18	0.015	0.025	0.025	0.015	0.115	0.02	0.23
19	0.015	0.025	0.025	0.015	0.115	0.02	0.23
20	0.015	0.025	0.015	0.015	0.115	0.03	0.23

Table A.67: Execution time of AES 256 using CUDA on GPU with granularity 1 (Data size 32000*5, k=1)

CUDA for AES 256 Data Size=32000*5 Granularity 1 (Total threads used=2000) k=10							
Grid Dimension							
S.No	<20,100>	<100,20>	<50,40>	<40,50>	<1000,2>	<2,1000>	<2000,1>
1	0.185	0.26	0.2	0.185	1.185	0.225	2.32
2	0.185	0.26	0.2	0.185	1.185	0.225	2.32
3	0.185	0.26	0.2	0.185	1.185	0.225	2.32
4	0.18	0.26	0.2	0.185	1.185	0.225	2.32
5	0.18	0.26	0.205	0.185	1.185	0.225	2.32
6	0.18	0.265	0.205	0.185	1.185	0.225	2.32
7	0.18	0.265	0.205	0.18	1.185	0.225	2.32
8	0.18	0.265	0.205	0.18	1.185	0.225	2.32
9	0.18	0.265	0.205	0.18	1.185	0.235	2.32
10	0.18	0.265	0.205	0.18	1.185	0.22	2.32
11	0.18	0.265	0.205	0.18	1.185	0.22	2.32
12	0.18	0.27	0.205	0.18	1.185	0.22	2.325
13	0.18	0.27	0.205	0.17	1.185	0.23	2.325
14	0.18	0.27	0.205	0.17	1.19	0.23	2.325
15	0.175	0.27	0.195	0.19	1.19	0.23	2.325
16	0.175	0.27	0.195	0.19	1.19	0.23	2.325
17	0.175	0.27	0.195	0.19	1.19	0.215	2.325
18	0.175	0.27	0.195	0.19	1.19	0.215	2.325
19	0.165	0.255	0.19	0.175	1.19	0.215	2.325
20	0.19	0.255	0.19	0.175	1.19	0.215	2.325

Table A.68: Execution time of AES 256 using CUDA on GPU with granularity 1 (Data size 32000*5, k=10)

	AES 256 on GPU using CUDA Data Size = 32000*5 Granularity 1						
	k=10						
Grid Dimension	**<20,100>**	**<100,20>**	**<50,40>**	**<40,50>**	**<1000,2>**	**<2,1000>**	**<2000,1>**
Average	0.1795	0.2645	0.2005	0.182	1.18675	0.22375	2.32225
Standard Deviation	0.005104	0.005104	0.005356	0.006156	0.002447	0.005821	0.002552
Confidence Interval	0.177263 - 0.181737	0.262263 - 0.266737	0.198153 - 0.202847	0.179302 - 0.184698	1.185678 - 1.187822	0.221199 - 0.226301	2.321132 - 2.321132

Table A.69: Calculations for AES 256 using CUDA on GPU with granularity 1 (Data size 32000*5, k=10)

CUDA for AES 256 Data Size=32000*5 Granularity 2 (Total threads used=1000) k=1							
Grid Dimension							
S.No	<10,100>	<100,10>	<50,20>	<20,50>	<40,25>	<25,40>	<1000,1>
1	0.020	0.030	0.020	0.020	0.015	0.020	0.205
2	0.015	0.030	0.015	0.015	0.020	0.020	0.200
3	0.020	0.030	0.015	0.020	0.020	0.015	0.205
4	0.020	0.030	0.015	0.015	0.015	0.015	0.205
5	0.015	0.030	0.020	0.015	0.015	0.020	0.200
6	0.020	0.030	0.020	0.015	0.015	0.020	0.205
7	0.020	0.030	0.015	0.015	0.020	0.015	0.200
8	0.020	0.030	0.020	0.015	0.020	0.015	0.200
9	0.015	0.030	0.015	0.015	0.020	0.020	0.200
10	0.015	0.030	0.015	0.015	0.020	0.015	0.200
11	0.015	0.030	0.015	0.015	0.020	0.015	0.205
12	0.015	0.030	0.015	0.020	0.020	0.020	0.200
13	0.015	0.030	0.020	0.020	0.020	0.015	0.200
14	0.020	0.030	0.015	0.015	0.015	0.020	0.205
15	0.015	0.030	0.015	0.020	0.015	0.015	0.200
16	0.020	0.030	0.015	0.020	0.015	0.015	0.205
17	0.020	0.030	0.020	0.020	0.015	0.015	0.205
18	0.020	0.030	0.020	0.015	0.015	0.015	0.200
19	0.020	0.030	0.015	0.015	0.020	0.020	0.200
20	0.015	0.025	0.020	0.015	0.020	0.015	0.200

Table A.70: Execution time of AES 256 using CUDA on GPU with granularity 2 (Data size 32000*5, k=1)

CUDA for AES 256 Data Size=32000*5 Granularity 2 (Total threads used=1000) k=10							
Grid Dimension							
S.No	<10,100>	<100,10>	<50,20>	<20,50>	<40,25>	<25,40>	<1000,1>
1	0.155	0.295	0.175	0.160	0.165	0.160	2.030
2	0.160	0.290	0.165	0.155	0.155	0.170	2.015
3	0.170	0.290	0.160	0.160	0.160	0.165	2.025
4	0.165	0.290	0.160	0.155	0.155	0.165	2.020
5	0.160	0.280	0.165	0.160	0.155	0.165	2.020
6	0.160	0.285	0.155	0.160	0.155	0.170	2.045
7	0.155	0.295	0.170	0.165	0.150	0.165	2.030
8	0.160	0.295	0.165	0.165	0.155	0.160	2.015
9	0.160	0.285	0.165	0.160	0.165	0.165	2.025
10	0.165	0.280	0.175	0.150	0.155	0.150	2.020
11	0.165	0.280	0.165	0.165	0.150	0.160	2.020
12	0.165	0.285	0.165	0.150	0.165	0.165	2.045
13	0.155	0.290	0.170	0.155	0.160	0.165	2.030
14	0.160	0.285	0.165	0.165	0.170	0.165	2.015
15	0.160	0.300	0.165	0.160	0.160	0.165	2.025
16	0.165	0.295	0.160	0.155	0.160	0.155	2.020
17	0.170	0.280	0.160	0.165	0.160	0.165	2.020
18	0.160	0.295	0.150	0.150	0.165	0.175	2.045
19	0.170	0.290	0.165	0.155	0.155	0.160	2.030
20	0.160	0.280	0.160	0.160	0.165	0.165	2.015

Table A.71: Execution time of AES 256 using CUDA on GPU with granularity 2 (Data size 32000*5, k=10)

	AES 256 on GPU using CUDA Data Size = 32000*5 Granularity 2						
	k=10						
Grid Dimension	**<10,100>**	**<100,10>**	**<50,20>**	**<20,50>**	**<40,25>**	**<25,40>**	**<1000,1>**
Average	0.162	0.28825	0.164	0.1585	0.159	0.16375	2.0255
Standard Deviation	0.004702	0.00634	0.005982	0.005155	0.005525	0.00535	0.009854
Confidence Interval	0.159939 - 0.164061	0.285471 - 0.291029	0.161378 - 0.166622	0.156241 - 0.160759	0.156579 - 0.161421	0.161405 - 0.166095	2.021181 - 2.029819

Table A.72: Calculations for AES 256 using CUDA on GPU with granularity 2 (Data size 32000*5, k=10)

CUDA for AES 256 Data Size=32000*5 Granularity 10 (Total threads used=200) k=1							
Grid Dimension							
S.No	<20,10>	<10,20>	<8,25>	<25,8>	<2,100>	<100,2>	<200,1>
1	0.055	0.05	0.05	0.06	0.05	0.11	0.21
2	0.055	0.05	0.05	0.06	0.05	0.11	0.21
3	0.055	0.05	0.05	0.06	0.05	0.11	0.21
4	0.055	0.05	0.05	0.06	0.05	0.11	0.21
5	0.055	0.05	0.05	0.06	0.05	0.11	0.21
6	0.055	0.05	0.05	0.06	0.05	0.11	0.21
7	0.055	0.05	0.05	0.06	0.05	0.11	0.21
8	0.055	0.05	0.05	0.06	0.05	0.11	0.21
9	0.055	0.05	0.05	0.06	0.05	0.11	0.21
10	0.055	0.05	0.05	0.055	0.05	0.11	0.205
11	0.055	0.05	0.05	0.055	0.05	0.11	0.205
12	0.055	0.05	0.05	0.055	0.05	0.11	0.205
13	0.055	0.05	0.05	0.055	0.05	0.11	0.205
14	0.055	0.05	0.05	0.055	0.05	0.105	0.205
15	0.055	0.05	0.05	0.055	0.05	0.105	0.205
16	0.05	0.05	0.05	0.055	0.05	0.105	0.205
17	0.05	0.05	0.05	0.055	0.05	0.105	0.205
18	0.05	0.05	0.05	0.055	0.055	0.105	0.205
19	0.05	0.045	0.05	0.055	0.055	0.105	0.205
20	0.05	0.055	0.05	0.055	0.045	0.115	0.205

Table A.73: Execution time of AES 256 using CUDA on GPU with granularity 10 (Data size 32000*5, k=1)

CUDA for AES 256 Data Size=32000*5 Granularity 10 (Total threads used=200) k=10							
Grid Dimension							
S.No	<20,10>	<10,20>	<8,25>	<25,8>	<2,100>	<100,2>	<200,1>
1	0.53	0.5	0.495	0.555	0.505	1.075	2.07
2	0.53	0.5	0.495	0.555	0.505	1.075	2.07
3	0.53	0.5	0.495	0.555	0.505	1.075	2.07
4	0.53	0.5	0.495	0.555	0.505	1.075	2.07
5	0.53	0.5	0.495	0.555	0.505	1.075	2.07
6	0.54	0.5	0.495	0.555	0.505	1.075	2.07
7	0.54	0.49	0.49	0.55	0.505	1.075	2.07
8	0.54	0.49	0.49	0.55	0.505	1.075	2.07
9	0.54	0.49	0.49	0.55	0.505	1.075	2.075
10	0.54	0.49	0.49	0.55	0.5	1.08	2.075
11	0.525	0.49	0.49	0.55	0.5	1.08	2.075
12	0.525	0.495	0.49	0.55	0.5	1.08	2.065
13	0.525	0.495	0.49	0.56	0.5	1.08	2.065
14	0.525	0.495	0.485	0.56	0.5	1.08	2.065
15	0.535	0.495	0.485	0.56	0.5	1.08	2.065
16	0.535	0.495	0.485	0.56	0.495	1.08	2.065
17	0.535	0.505	0.485	0.555	0.495	1.09	2.065
18	0.535	0.505	0.485	0.555	0.495	1.085	2.065
19	0.535	0.505	0.48	0.555	0.495	1.085	2.065
20	0.535	0.505	0.5	0.555	0.51	1.085	2.085

Table A.74: Execution time of AES 256 using CUDA on GPU with granularity 10 (Data size 32000*5, k=10).

	AES 256 on GPU using CUDA Data Size = 32000*5 Granularity 10						
	k=10						
Grid Dimension	**<20,10>**	**<10,20>**	**<8,25>**	**<25,8>**	**<2,100>**	**<100,2>**	**<200,1>**
Average	0.533	0.49725	0.49025	0.5545	0.50175	1.079	2.0695
Standard Deviation	0.005477	0.005495	0.004993	0.003591	0.004375	0.004472	0.005104
Confidence Interval	0.530600 - 0.535400	0.494842 - 0.499658	0.488062 - 0.492438	0.552926 - 0.556074	0.4998320 - 0.503668	1.077040 - 1.080960	2.067263 - 2.071737

Table A.75: Calculations for AES 256 using CUDA on GPU with granularity 10 (Data size 32000*5, k=10)

CUDA for AES 256 Data Size=32000*5 Granularity 100 (Total threads used=20) k=1						
Grid Dimension						
S.No	<2,10>	<10,2>	<5,4>	<4,5>	<20,1>	<1,20>
1	0.405	0.41	0.405	0.405	0.435	0.41
2	0.405	0.41	0.405	0.405	0.435	0.41
3	0.405	0.41	0.405	0.405	0.435	0.41
4	0.405	0.41	0.405	0.405	0.435	0.41
5	0.405	0.41	0.405	0.405	0.435	0.41
6	0.405	0.41	0.405	0.405	0.435	0.41
7	0.405	0.405	0.405	0.405	0.435	0.41
8	0.405	0.405	0.405	0.405	0.435	0.41
9	0.405	0.405	0.405	0.405	0.435	0.41
10	0.405	0.405	0.405	0.405	0.435	0.41
11	0.405	0.405	0.41	0.4	0.435	0.41
12	0.405	0.405	0.41	0.4	0.435	0.41
13	0.405	0.405	0.41	0.4	0.435	0.41
14	0.405	0.405	0.41	0.4	0.43	0.41
15	0.405	0.405	0.41	0.4	0.43	0.41
16	0.4	0.405	0.41	0.4	0.43	0.41
17	0.4	0.405	0.41	0.4	0.43	0.41
18	0.4	0.405	0.41	0.4	0.43	0.405
19	0.4	0.405	0.41	0.4	0.43	0.405
20	0.4	0.405	0.41	0.4	0.43	0.405

Table A.76: Execution time of AES 256 using CUDA on GPU with granularity 100 (Data size 32000*5, k=1)

CUDA for AES 256 Data Size=32000*5 Granularity 100 (Total threads used=20) k=10						
	Grid Dimension					
S.No	<2,10>	<10,2>	<5,4>	<4,5>	<20,1>	<1,20>
1	4.05	4.1	4.05	4.05	4.35	4.1
2	4.05	4.1	4.05	4.05	4.35	4.1
3	4.05	4.1	4.05	4.05	4.35	4.1
4	4.05	4.1	4.05	4.05	4.35	4.1
5	4.05	4.1	4.05	4.05	4.35	4.1
6	4.05	4.1	4.05	4.05	4.35	4.1
7	4.05	4.05	4.05	4.05	4.35	4.1
8	4.05	4.05	4.05	4.05	4.35	4.1
9	4.05	4.05	4.05	4.05	4.35	4.1
10	4.05	4.05	4.05	4.05	4.35	4.1
11	4.05	4.05	4.1	4	4.35	4.1
12	4.05	4.05	4.1	4	4.35	4.1
13	4.05	4.05	4.1	4	4.35	4.1
14	4.05	4.05	4.1	4	4.3	4.1
15	4.05	4.05	4.1	4	4.3	4.1
16	4	4.05	4.1	4	4.3	4.1
17	4	4.05	4.1	4	4.3	4.1
18	4	4.05	4.1	4	4.3	4.05
19	4	4.05	4.1	4	4.3	4.05
20	4	4.05	4.1	4	4.3	4.05

Table A.77: Execution time of AES 256 using CUDA on GPU with granularity 100 (Data size 32000*5, k=10)

	AES 256 on GPU using CUDA Data Size = 32000*5 Granularity 100					
	k=10					
Grid Dimension	**<2,10>**	**<10,2>**	**<5,4>**	**<4,5>**	**<20,1>**	**<1,20>**
Average	4.0375	4.065	4.075	4.025	4.3325	4.0925
Standard Deviation	0.022213	0.023508	0.025649	0.025649	0.024468	0.018317
Confidence Interval	4.027765 - 4.047235	4.054697 - 4.075303	4.063759 - 4.086241	4.013759 - 4.036241	4.321777 - 4.343223	4.084472 - 4.100528

Table A.78: Calculations for AES 256 using CUDA on GPU with granularity 100 (Data size 32000*5, k=10)

Multi threaded C using POSIX THREAD for AES 128 No of threads=12				
	Data Size=32000		Data Size=32000*5	
S.No	k=1	k=10	k=1	k=10
1	0.01	0.075	0.035	0.275
2	0.01	0.065	0.035	0.265
3	0.01	0.055	0.035	0.26
4	0.005	0.06	0.035	0.28
5	0.01	0.07	0.035	0.265
6	0.015	0.07	0.035	0.265
7	0.005	0.075	0.035	0.275
8	0.01	0.065	0.035	0.29
9	0.005	0.07	0.035	0.275
10	0.01	0.055	0.035	0.28
11	0.01	0.07	0.035	0.27
12	0.01	0.07	0.035	0.275
13	0.01	0.065	0.035	0.26
14	0.01	0.07	0.04	0.285
15	0.005	0.07	0.035	0.26
16	0.005	0.075	0.035	0.275
17	0.01	0.065	0.03	0.255
18	0.005	0.07	0.025	0.28
19	0.005	0.07	0.035	0.27
20	0.01	0.07	0.04	0.275

Table A.79: Execution time of AES 128 using POSIX thread on Multi core CPU

Multi threaded C using POSIX THREAD for AES 128 No of threads=12		
	k=10	
	Data Size=32000	Data Size=32000*5
Average	0.06775	0.27175
Standard Deviation	0.00573	0.009216
Confidence Interval	0.065239 - 0.070261	0.267711 - 0.275789

Table A.80: Calculations for AES 128 using POSIX thread on Multi core CPU

Multi threaded C using POSIX THREAD for AES 192 No of threads=12				
	Data Size=32000		Data Size=32000*5	
S.No	**k=1**	**k=10**	**k=1**	**k=10**
1	0.02	0.08	0.03	0.335
2	0.01	0.085	0.04	0.36
3	0.01	0.085	0.04	0.295
4	0.01	0.09	0.03	0.325
5	0.005	0.09	0.035	0.32
6	0.01	0.08	0.035	0.33
7	0.005	0.08	0.035	0.33
8	0.005	0.085	0.035	0.345
9	0.01	0.075	0.03	0.325
10	0.005	0.09	0.04	0.315
11	0.005	0.09	0.04	0.31
12	0.01	0.09	0.04	0.335
13	0.01	0.075	0.04	0.315
14	0.015	0.085	0.035	0.35
15	0.01	0.1	0.04	0.32
16	0.005	0.07	0.04	0.325
17	0.01	0.08	0.045	0.335
18	0.01	0.08	0.04	0.32
19	0.015	0.08	0.04	0.32
20	0.01	0.08	0.04	0.325

Table A.81: Execution time of AES 192 using POSIX thread on Multi core CPU

Multi threaded C using POSIX THREAD for AES 192 No of threads=12		
	k=10	
	Data Size=32000	Data Size=32000*5
Average	0.0835	0.32675
Standard Deviation	0.006902	0.014444
Confidence Interval	0.080475 - 0.086525	0.32042 - 0.33308

Table A.82: Calculations for AES 192 using POSIX thread on Multi core CPU

Multi threaded C using POSIX THREAD for AES 256 No of threads=12				
	Data Size=32000		Data Size=32000*5	
S.No	**k=1**	**k=10**	**k=1**	**k=10**
1	0.015	0.085	0.04	0.385
2	0.01	0.085	0.045	0.365
3	0.015	0.095	0.05	0.42
4	0.01	0.095	0.04	0.0435
5	0.01	0.095	0.055	0.43
6	0.01	0.085	0.04	0.42
7	0.015	0.09	0.04	0.38
8	0.01	0.09	0.035	0.38
9	0.01	0.095	0.045	0.442
10	0.01	0.09	0.05	0.38
11	0.005	0.085	0.045	0.35
12	0.01	0.1	0.045	0.35
13	0.01	0.095	0.035	0.35
14	0.015	0.095	0.045	0.39
15	0.01	0.09	0.04	0.365
16	0.01	0.09	0.03	0.375
17	0.001	0.09	0.035	0.36
18	0.01	0.085	0.05	0.375
19	0.01	0.09	0.035	0.4
20	0.01	0.09	0.035	0.37

Table A.83: Execution time of AES 256 using POSIX thread on Multi core CPU

Multi threaded C using POSIX THREAD for AES 256 No of threads=12		
	k=10	
	Data Size=32000	Data Size=32000*5
Average	0.09075	0.366525
Standard Deviation	0.004375	0.08058
Confidence Interval	0.088832 - 0.092668	0.33121 - 0.40184

Table A.84: Calculations for AES 256 using POSIX thread on Multi core CPU

CUDA using CUDA STREAMS for AES 128 Total Data Size=32000 bytes No of Streams=2 Each stream=16000 bytes		
S.No	k=1	k=10
1	0.000	0.027
2	0.000	0.030
3	0.000	0.030
4	0.005	0.030
5	0.005	0.025
6	0.005	0.025
7	0.000	0.030
8	0.000	0.030
9	0.005	0.025
10	0.005	0.030
11	0.005	0.030
12	0.000	0.030
13	0.000	0.025
14	0.000	0.030
15	0.005	0.030
16	0.000	0.030
17	0.005	0.025
18	0.000	0.025
19	0.000	0.025
20	0.000	0.025

Table A.85: Execution time of AES 128 using CUDA STREAMS on GPU (Data size 32000)

CUDA using CUDA STREAMS for AES 128 Total Data Size=32000*2 bytes No of Streams=2 Each stream=32000 bytes		
S.No	k=1	k=10
1	0.000	0.035
2	0.005	0.035
3	0.000	0.035
4	0.005	0.030
5	0.005	0.035
6	0.000	0.030
7	0.005	0.035
8	0.005	0.030
9	0.000	0.35
10	0.005	0.035
11	0.000	0.035
12	0.000	0.035
13	0.005	0.035
14	0.000	0.035
15	0.000	0.035
16	0.000	0.035
17	0.005	0.030
18	0.005	0.035
19	0.005	0.035
20	0.005	0.030

Table A.86: Execution time of AES 128 using CUDA STREAMS on GPU (Data size 32000*2)

CUDA using CUDA STREAMS for AES 128 **Total Data Size=32000*5 bytes** **No of Streams=5 Each stream=32000 bytes**		
S.No	**k=1**	**k=10**
1	0.010	0.105
2	0.010	0.100
3	0.010	0.110
4	0.010	0.105
5	0.010	0.110
6	0.010	0.100
7	0.010	0.105
8	0.010	0.110
9	0.015	0.105
10	0.010	0.105
11	0.010	0.105
12	0.010	0.105
13	0.010	0.110
14	0.010	0.105
15	0.015	0.105
16	0.010	0.105
17	0.010	0.100
18	0.010	0.105
19	0.010	0.105
20	0.010	0.105

Table A.87: Execution time of AES 128 using CUDA STREAMS on GPU (Data size 32000*5)

CUDA using CUDA STREAMS for AES 192 **Total Data Size=32000 bytes** **No of Streams=2 Each stream=16000 bytes**		
S.No	**k=1**	**k=10**
1	0.005	0.035
2	0.005	0.030
3	0.005	0.030
4	0.000	0.030
5	0.005	0.030
6	0.000	0.040
7	0.005	0.035
8	0.005	0.030
9	0.005	0.035
10	0.000	0.035
11	0.005	0.035
12	0.005	0.030
13	0.000	0.035
14	0.005	0.035
15	0.005	0.035
16	0.000	0.035
17	0.000	0.035
18	0.005	0.035
19	0.005	0.030
20	0.005	0.035

Table A.88: Execution time of AES 192 using CUDA STREAMS on GPU (Data size 32000)

CUDA using CUDA STREAMS for AES 192 Total Data Size=32000*2 bytes No of Streams=2 Each stream=32000 bytes		
S.No	k=1	k=10
1	0.005	0.065
2	0.005	0.065
3	0.005	0.065
4	0.01	0.065
5	0.005	0.06
6	0.005	0.06
7	0.010	0.065
8	0.010	0.065
9	0.005	0.06
10	0.010	0.065
11	0.005	0.065
12	0.005	0.06
13	0.005	0.065
14	0.010	0.06
15	0.005	0.06
16	0.005	0.065
17	0.005	0.06
18	0.010	0.065
19	0.010	0.065
20	0.010	0.065

Table A.89: Execution time of AES 192 using CUDA STREAMS on GPU (Data size 32000*2)

CUDA using CUDA STREAMS for AES 192 Total Data Size=32000*5 bytes No of Streams=5 Each stream=32000 bytes		
S.No	k=1	k=10
1	0.015	0.13
2	0.01	0.13
3	0.015	0.13
4	0.015	0.125
5	0.015	0.13
6	0.01	0.13
7	0.015	0.13
8	0.015	0.13
9	0.015	0.135
10	0.015	0.13
11	0.015	0.13
12	0.01	0.13
13	0.015	0.125
14	0.015	0.13
15	0.015	0.125
16	0.01	0.13
17	0.015	0.125
18	0.015	0.13
19	0.01	0.13
20	0.01	0.13

Table A.90: Execution time of AES 192 using CUDA STREAMS on GPU (Data size 32000*5)

CUDA using CUDA STREAMS for AES 256
Total Data Size=32000 bytes
No of Streams=2 Each stream=16000 bytes

S.No	k=1	k=10
1	0.005	0.035
2	0.005	0.04
3	0.005	0.035
4	0.005	0.040
5	0.000	0.035
6	0.005	0.04
7	0.005	0.035
8	0.005	0.040
9	0.000	0.040
10	0.005	0.03
11	0.005	0.04
12	0.000	0.035
13	0.005	0.040
14	0.000	0.035
15	0.005	0.040
16	0.005	0.040
17	0.005	0.040
18	0.005	0.040
19	0.005	0.035
20	0.005	0.035

Table A.91: Execution time of AES 256 using CUDA STREAMS on GPU (Data size 32000)

CUDA using CUDA STREAMS for AES 256
Total Data Size=32000*2 bytes
No of Streams=2 Each stream=32000 bytes

S.No	k=1	k=10
1	0.01	0.07
2	0.01	0.075
3	0.01	0.07
4	0.01	0.07
5	0.005	0.075
6	0.005	0.07
7	0.005	0.075
8	0.005	0.07
9	0.01	0.07
10	0.01	0.075
11	0.005	0.075
12	0.005	0.075
13	0.01	0.07
14	0.005	0.07
15	0.01	0.075
16	0.005	0.075
17	0.01	0.07
18	0.01	0.07
19	0.005	0.075
20	0.01	0.07

Table A.92: Execution time of AES 256 using CUDA STREAMS on GPU (Data size 32000*2)

CUDA using CUDA STREAMS for AES 256
Total Data Size=32000*5 bytes
No of Streams=5 Each stream=32000 bytes

S.No	k=1	k=10
1	0.015	0.155
2	0.015	0.15
3	0.015	0.15
4	0.020	0.155
5	0.015	0.15
6	0.015	0.15
7	0.015	0.15
8	0.020	0.15
9	0.020	0.15
10	0.015	0.15
11	0.015	0.145
12	0.015	0.145
13	0.020	0.155
14	0.015	0.15
15	0.015	0.15
16	0.020	0.15
17	0.015	0.15
18	0.015	0.15
19	0.015	0.145
20	0.015	0.15

Table A.93: Execution time of AES 256 using CUDA STREAMS on GPU (Data size 32000*5)

APPENDIX B

This section delivers the program to implement AES-128, AES-192 and AES-256 using C, Pthreads, CUDA and CUDA STREAMS.

B.1 AES-128 SINGLE THREADED C ON CPU

The three versions of AES, namely AES-128, AES-192 and AES-256 operate on an input data size of 16 bytes or 128 bites and a key size of 16 bytes, 24 bytes and 32 bytes respectively. AES-128 has 10 transformation rounds, whereas AES-192 and AES-256 have 12 transformation rounds and 14 transformation rounds respectively. The key scheduling also known as the round key generation method is different for the three versions of AES algorithm.

The program for AES-128 on single threaded CPU is shown below. It is followed by key expansion methods of AES-192 and AES-256.

```
int key[16]={12, 52, 41, 23, 25, 14, 25, 12, 14, 16, 25, 14, 12, 57, 4, 25};                    //Cipher-Key
int rk1[16]; int rk2[16]; int rk3[16]; int rk4[16]; int rk5[16]; int rk6[16];                   //Round-Keys
int rk7[16]; int rk8[16]; int rk9[16]; int rk10[16];
                                                                                                   // S-Box
const int sbox[16][16] = {
        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, 4,   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,  108, 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, 236, 66,  104, 65,  153, 45,  15,  176, 84,   187, 22};

                                                                          // Multiplication 2 table


const int m2[16][16]={
        0,  2,  4,  6,  8,  10,  12,  14,  16,  18,  20,  22,  24,  26,  28,  30,
        32, 34, 36, 38, 40, 42,  44,  46,  48,  50,  52,  54,  56,  58,  60,  62,
        64, 66, 68, 70, 72, 74,  76,  78,  80,  82,  84,  86,  88,  90,  92,  94,
        96, 98, 100,102,104,106,108,110,112, 114, 116, 118, 120,122, 124, 126,
        128,130,132,134,136,138,140,142,144,146,148, 150, 152, 154, 156, 158,
        160,162,164,166,168,170,172,174,176,178,180, 182, 184, 186, 188, 190,
        192,194,196,198,200,202,204,206,208, 210, 212, 214, 216, 218, 220, 222,
        224,226,228,230,232,234,236,238, 240, 242, 244,   246,   248, 250, 252,254,
        27,   25,   31,   29,   19,   17,   23,   21,   11,   9,   15,   13,   3,   1,    7,   5,
        59,   57,   63,   61,   51,   49,   55,   53,   43,   41,   47,   45,   35,   33, 39, 37,
        91,   89,   95,   93,   83,    81,   87,   85,   75,   73,    79,   77,   67, 65, 71, 69,
        123, 121, 127,125,115,113,119,117,107, 105, 111,109, 99, 97, 103,101,
        155, 153, 159,157,147,145,151,149,139, 137, 143,141,131,129,135,133,
        187, 185, 191,189,179,177,183,181,171, 169, 175,173,163,161,167,165,
        219, 217, 223,221,211,209,215,213,203, 201, 207,205,195,193,199,197,
        251, 249, 255, 253, 243,   241, 247, 245, 235,233, 239, 237, 227, 225, 231,  229};

                                                                // MULTIPLICATION 3 TABLE

const int m3[16][16]={
        0,    3,   6,   5,   12,   15,   10,   9,   24,   27,    30,   29,    20,    23,    18,    17,
        48,  51, 54, 53, 60,   63,   58,    57,40,    43,    46,   45,    36,    39,     34,   33,
        96,  99, 102, 101, 108, 111,106, 105, 120, 123, 126, 125, 116, 119, 114, 113,
        80,    83,   86,   85,   92,   95,  90,  89,  72,  75,    78,  77,    68,   71,   66,   65,
        192,  195, 198, 197, 204, 207, 202, 201, 216, 219, 222, 221, 212, 215,210,209,
        240,  243, 246, 245, 252, 255, 250, 249, 232, 235, 238, 237, 228,231,226,225,
        160, 163, 166, 165, 172, 175, 170, 169, 184, 187, 190, 189, 180, 183,178,177,
        144, 147, 150, 149, 156,  159, 154, 153, 136, 139, 142, 141, 132, 135,130,129,
        155, 152, 157, 158, 151, 148, 145, 146, 131, 128, 133, 134, 143, 140, 137, 138,
        171, 168, 173, 174, 167, 164, 161, 162, 179, 176, 181, 182, 191, 188, 185, 186,
        251, 248, 253, 254, 247, 244, 241, 242, 227, 224, 229, 230, 239, 236, 233, 234,
        203, 200, 205, 206, 199, 196, 193, 194, 211, 208, 213, 214, 223, 220, 217, 218,
        91,   88,   93,    94,  87,   84,    81,   82,   67,   64,   69,  70,    79,   76,    73,    74,
        107, 104, 109, 110, 103, 100,    97,    98, 115, 112, 117, 118, 127, 124, 121,  122,
        59,   56,   61,  62,    55,   52,   49,   50,   35,   32,   37,    38,    47,    44,    41,    42,
        11,   8,    13,   14,    7,   4,   1,   2,   19,   16,    21,    22,    31,    28,    25,    26   };


                                                                        //Add Round Key

void addroundkey(int *a, int *b)
{
    int i=0;
    while(i<16){
        a[i]=a[i]^b[i];
        i++;
     }
}

                                                                        //Shift Row
void shift(int *a) {
    int i=0;
    int temp1;
    temp1=a[0];
    while(i<4){
        a[i]=a[i+1]; i++;
        }
        a[3]=temp1;
}

void shiftrow(int *a){

        int i;
        int temp1[4],temp2[4],temp3[4],temp4[4];
        i=0;
        while(i<4){
                 temp1[i]=a[i];
                temp2[i]=a[i+4];
                temp3[i]=a[i+8];
                temp4[i]=a[i+12];
                i++;
        }
        shift(temp2);
        shift(temp3);
        shift(temp3);
        shift(temp4);
        shift(temp4);
        shift(temp4);
        i=0;
        while(i<4) {
                a[i]=temp1[i];
                a[i+4]=temp2[i];
                a[i+8]=temp3[i];
                a[i+12]=temp4[i];
                i++;
        }
}
                                                                                          //Substitute Byte
void subbyte(int *a) {
        int i;
        int pt1,pt2;
        for (i = 0; i < 16; i++){
                pt1 = ((240 & a[i]) / 16);
                pt2 = (15 & a[i]);
                a[i] = sbox[pt1][pt2];
         }
}

                                                                                            //Mix Column
void mixcolumn(int *a)  {
        int m1,n1,o1,p1,m2,n2,o2,p2;
        m1=a[0]; n1=a[4];o1=a[8];p1=a[12];
        m2=m1; n2=n1;o2=o1;p2=p1;

        mult2(m1);  mult2(o1);
```

```
    mult3(m2);  mult3(o2);
    mult2(n1);   mult2(p1);
    mult3(n2);   mult3(p2);

    a[0]= m1 ^ n2 ^ a[8] ^ a[12];
    a[1]= a[0] ^ n1 ^ o2 ^ a[12];
    a[2]= a[0]^ a[4] ^ o1 ^ p2;
    a[3] = m2 ^ a[4] ^ a[8] ^ p1;

    m1=a[1]; n1=a[5];o1=a[9];p1=a[13];
    m2=m1; n2=n1;o2=o1;p2=p1;

    mult2(m1);  mult2(o1);
    mult3(m2);  mult3(o2);
    mult2(n1);   mult2(p1);
    mult3(n2);   mult3(p2);

    a[1]=m1 ^ n2 ^ a[9] ^ a[13];
    a[5]=a[1] ^ n1 ^ o2 ^ a[13];
    a[9]=a[1]^ a[5] ^ o1 ^ p2;
    a[13]=m2 ^ a[5] ^ a[9] ^ p1;

    m1=a[2]; n1=a[6];o1=a[10];p1=a[14];
    m2=m1; n2=n1;o2=o1;p2=p1;

    mult2(m1);  mult2(o1);
    mult3(m2);  mult3(o2);
    mult2(n1);   mult2(p1);
    mult3(n2);   mult3(p2);

    a[2]=m1 ^ n2 ^ a[10] ^ a[14];
    a[6]=a[2] ^ n1 ^ o2 ^ a[14];
    a[10]=a[2]^ a[6] ^ o1 ^ p2;
    a[14]=m2 ^ a[6] ^ a[10] ^ p1;

    m1=a[3]; n1=a[7];o1=a[11];p1=a[15];
    m2=m1; n2=n1;o2=o1;p2=p1;

    mult2(m1);  mult2(o1);
    mult3(m2);  mult3(o2);
    mult2(n1);   mult2(p1);
    mult3(n2);   mult3(p2);

    a[3]=m1 ^ n2 ^ a[11] ^ a[15];
    a[7]=a[3] ^ n1 ^ o2 ^ a[15];
    a[11]=a[3]^ a[7] ^ o1 ^ p2;
    a[15]=m2 ^ a[7] ^ a[11] ^ p1;
}
```

```
void nextrk(int *rk1, int *rk2, int*rcon) {                                   //Key-Expansion
        int i;
        int rot[4];
        int tempa[4], tempb[4], tempc[4], tempd[4];

        tempa[0] = rk1[0];
        tempb[0] = rk1[1];
        tempc[0] = rk1[2];
        tempd[0] = rk1[3];
        tempa[1] = rk1[4];
        tempb[1] = rk1[5];
        tempc[1] = rk1[6];
        tempd[1] = rk1[7];
        tempa[2] = rk1[8];
        tempb[2] = rk1[9];
        tempc[2] = rk1[10];
        tempd[2] = rk1[11];
        tempa[3] = rk1[12];
        tempb[3] = rk1[13];
        tempc[3] = rk1[14];
        tempd[3] = rk1[15];

        for (i = 0; i < 4; i++) {
                rot[i] = tempd[i];
        }

        shift(rot);
        subbyte4(rot);

        for (i = 0; i < 4; i++) {
                rot[i] = rot[i] ^ tempa[i] ^ rcon[i];
        }
        rk2[0] = rot[0];
        rk2[4] = rot[1];
        rk2[8] = rot[2];
        rk2[12] = rot[3];

        for (i = 0; i < 4; i++) {
                 rot[i] = rot[i] ^ tempb[i];
        }
        rk2[1] = rot[0];
        rk2[5] = rot[1];
        rk2[9] = rot[2];
        rk2[13] = rot[3];

        for (i = 0; i < 4; i++) {
                rot[i] = rot[i] ^ tempc[i];
        }
```

```
    rk2[2] = rot[0];
    rk2[6] = rot[1];
    rk2[10] = rot[2];
    rk2[14] = rot[3];

    for (i = 0; i < 4; i++) {
        rot[i] = rot[i] ^ tempd[i];
     }rk2[3] = rot[0];
    rk2[7] = rot[1];
    rk2[11] = rot[2];
    rk2[15] = rot[3];
}

void keyexp(int *cipherkey, int *rk1, int *rk2, int *rk3, int *rk4, int *rk5, int *rk6, int *rk7, int *rk8,
                    int *rk9, int *rk10) {
    int rcon1[4] = { 1, 0, 0, 0 };
    int rcon2[4] = { 2, 0, 0, 0 };
    int rcon3[4] = { 4, 0, 0, 0 };
    int rcon4[4] = { 8, 0, 0, 0 };
    int rcon5[4] = { 16, 0, 0, 0 };
    int rcon6[4] = { 32, 0, 0, 0 };
    int rcon7[4] = { 64, 0, 0, 0 };
    int rcon8[4] = { 128, 0, 0, 0 };
    int rcon9[4] = { 27, 0, 0, 0 };
    int rcon10[4] = { 54, 0, 0, 0 };

    nextrk(cipherkey, rk1, rcon1);
    nextrk(rk1, rk2, rcon2);
    nextrk(rk2, rk3, rcon3);
    nextrk(rk3, rk4, rcon4);
    nextrk(rk4, rk5, rcon5);
    nextrk(rk5, rk6, rcon6);
    nextrk(rk6, rk7, rcon7);
    nextrk(rk7, rk8, rcon8);
    nextrk(rk8, rk9, rcon9);
    nextrk(rk9, rk10, rcon10);
}


void encryption(int state[16], int cipherkey[16]) {

    addroundkey(state, cipherkey);                                  //Initial Round

    subbyte(state);                                                 //Round 1
    shiftrow(state);
    mixcolumn(state);
    addroundkey(state, rk1);

    subbyte(state);                                                 //Round 2
```

```
      shiftrow(state);
    mixcolumn(state);
    addroundkey(state, rk2);

    subbyte(state);                                              //Round 3
    shiftrow(state);
    mixcolumn(state);
    addroundkey(state, rk3);

    subbyte(state);                                              //Round 4
    shiftrow(state);
    mixcolumn(state);
    addroundkey(state, rk4);

    subbyte(state);                                              //Round 5
    shiftrow(state);
    mixcolumn(state);
    addroundkey(state, rk5);

    subbyte(state);                                              //Round 6
    shiftrow(state);
    mixcolumn(state);
    addroundkey(state, rk6);

    subbyte(state);                                              //Round 7
    shiftrow(state);
    mixcolumn(state);
    addroundkey(state, rk7);

    subbyte(state);                                              //Round 8
    shiftrow(state);
    mixcolumn(state);
    addroundkey(state, rk8);

    subbyte(state);                                              //Round 9
    shiftrow(state);
    mixcolumn(state);
    addroundkey(state, rk9);

    subbyte(state);                                             //Round 10
    shiftrow(state);
    addroundkey(state, rk10);
}
```

```
int main()
{
    int state[N],take[16];
    int i, j, k;
    double total_time;
    clock_t start, end;
    for(i=0;i<N ;i++){
        state[i]=i;
    }
    start = clock();                             //time record starts
    keyexp(cipherkey, rk1, rk2, rk3, rk4, rk5, rk6, rk7, rk8, rk9, rk10); //key expansion
    for(k=0; k<n ;k++) {                         //encryption performed 'n' no of times'
        for(i=0;i<(N);i+=16) {                    //value passed in sets of 16 for each run
            for(j=0;j<16;j++) {
                take[j]=state[i+j];
            }
            encryption(take, cipherkey);
            for(j=0;j<16;j++) {
                state[i+j]=take[j];
            }
        }
    }                                     // k ends here
    end = clock();          //time record stops
    total_time = ((double)(end - start)) / CLK_TCK;
    printf("\n Time taken to for aes128 execution in c is: %f\n", total_time);
    return 0;
}
```

B.2 KEY EXPANSION FOR AES-128

```
void nextrk(int *rk1, int *rk2, int *rcon) {
    int i;
    int tempa[4], tempb[4], tempc[4];
    int tempd[4], tempe[4], tempf[4];
    int rot[4];

    tempa[0] = rk1[0];
    tempb[0] = rk1[1];
    tempc[0] = rk1[2];
    tempa[1] = rk1[6];
    tempb[1] = rk1[7];
    tempc[1] = rk1[8];
    tempa[2] = rk1[12];
    tempb[2] = rk1[13];
    tempc[2] = rk1[14];
    tempa[3] = rk1[18];
    tempb[3] = rk1[19];
    tempc[3] = rk1[20];
    tempd[0] = rk1[3];
    tempe[0] = rk1[4];
    tempf[0] = rk1[5];
    tempd[1] = rk1[9];
    tempe[1] = rk1[10];
    tempf[1] = rk1[11];
    tempd[2] = rk1[15];
    tempe[2] = rk1[16];
    tempf[2] = rk1[17];
    tempd[3] = rk1[21];
    tempe[3] = rk1[22];
    tempf[3] = rk1[23];
    for (i = 0; i < 4; i++)  {
        rot[i] = tempf[i];
     }
    shift(rot);
     subbyte4(rot);
    for (i = 0; i < 4; i++) {
        rot[i] = rot[i] ^ tempa[i] ^ rcon[i];
    }
    rk2[0] = rot[0];
    rk2[6] = rot[1];
    rk2[12] = rot[2];
    rk2[18] = rot[3];
    for (i = 0; i < 4; i++) {
        rot[i] = rot[i] ^ tempb[i];
    }
    rk2[1] = rot[0];
    rk2[7] = rot[1];
```

```
        rk2[13] = rot[2];
        rk2[19] = rot[3];
        for (i = 0; i < 4; i++){
                rot[i] = rot[i] ^ tempc[i];
         }
        rk2[2] = rot[0];
        rk2[8] = rot[1];
        rk2[14] = rot[2];
        rk2[20] = rot[3];
        for (i = 0; i < 4; i++) {
                rot[i] = rot[i] ^ tempd[i];
        }
        rk2[3] = rot[0];
        rk2[9] = rot[1];
        rk2[15] = rot[2];
        rk2[21] = rot[3];
        for (i = 0; i < 4; i++) {
                rot[i] = rot[i] ^ tempe[i];
        }
        rk2[4] = rot[0];
        rk2[10] = rot[1];
         rk2[16] = rot[2];
        rk2[22] = rot[3];
        for (i = 0; i < 4; i++) {
               rot[i] = rot[i] ^ tempf[i];
        }
        rk2[5] = rot[0];
        rk2[11] = rot[1];
        rk2[17] = rot[2];
        rk2[23] = rot[3];
}
void keyexp(int *key, int *inr, int *r1,int *r2,int *r3,int *r4,int *r5,int *r6,
                              int *r7,int *r8,int *r9,int *r10,int *r11,int *r12 ) {
        int rk1[24],rk2[24],rk3[24],rk4[24],rk5[24],rk6[24],rk7[24],rk8[24];int i;
        int rcon1[4] = { 1, 0, 0, 0 };
        int rcon2[4] = { 2, 0, 0, 0 };
        int rcon3[4] = { 4, 0, 0, 0 };
        int rcon4[4] = { 8, 0, 0, 0 };
        int rcon5[4] = { 16, 0, 0, 0 };
        int rcon6[4] = { 32, 0, 0, 0 };
        int rcon7[4] = { 64, 0, 0, 0 };
        int rcon8[4] = { 128, 0, 0, 0 };
        nextrk(key,rk1,rcon1);
        nextrk(rk1,rk2,rcon2);
        nextrk(rk2,rk3,rcon3);
        nextrk(rk3,rk4,rcon4);
        nextrk(rk4,rk5,rcon5);
        nextrk(rk5,rk6,rcon6);
        nextrk(rk6,rk7,rcon7);
nextrk(rk7,rk8,rcon8);
for(i=0;i<4;i++) {
        inr[i]=key[i];
         r3[i]=rk2[i];
        r6[i]=rk4[i];
        r9[i]=rk6[i];
        r12[i]=rk8[i];

        inr[i+4]=key[i+4+2];
        r3[i+4]=rk2[i+4+2];
        r6[i+4]=rk4[i+4+2];
        r9[i+4]=rk6[i+4+2];
        r12[i+4]=rk8[i+4+2];

       inr[i+8]=key[i+8+4];
       r3[i+8]=rk2[i+8+4];
       r6[i+8]=rk4[i+8+4];
       r9[i+8]=rk6[i+8+4];
       r12[i+8]=rk8[i+8+4];

        inr[i+12]=key[i+12+6];
        r3[i+12]=rk2[i+12+6];
        r6[i+12]=rk4[i+12+6];
        r9[i+12]=rk6[i+12+6];
        r12[i+12]=rk8[i+12+6];

        r2[i]=rk1[i+2];
        r5[i]=rk3[i+2];
        r8[i]=rk5[i+2];
        r11[i]=rk7[i+2];

        r2[i+4]=rk1[i+4+4];
        r5[i+4]=rk3[i+4+4];
        r8[i+4]=rk5[i+4+4];
        r11[i+4]=rk7[i+4+4];

        r2[i+8]=rk1[i+8+6];
        r5[i+8]=rk3[i+8+6];
        r8[i+8]=rk5[i+8+6];
        r11[i+8]=rk7[i+8+6];

        r2[i+12]=rk1[i+12+8];
        r5[i+12]=rk3[i+12+8];
        r8[i+12]=rk5[i+12+8];
        r11[i+12]=rk7[i+12+8];

}
```

```
    for(i=0;i<2;i++) {
        r1[i]=key[i+4];
        r4[i]=rk2[i+4];
        r7[i]=rk4[i+4];
        r10[i]=rk6[i+4];

        r1[i+4]=key[i+10];
        r4[i+4]=rk2[i+10];
        r7[i+4]=rk4[i+10];
        r10[i+4]=rk6[i+10];

        r1[i+8]=key[i+16];
        r4[i+8]=rk2[i+16];
        r7[i+8]=rk4[i+16];
        r10[i+8]=rk6[i+16];

        r1[i+12]=key[i+22];
        r4[i+12]=rk2[i+22];
        r7[i+12]=rk4[i+22];
        r10[i+12]=rk6[i+22];

        r1[i+2]=rk1[i];
        r4[i+2]=rk3[i];
        r7[i+2]=rk5[i];
        r10[i+2]=rk7[i];

        r1[i+6]=rk1[i+6];
        r4[i+6]=rk3[i+6];
        r7[i+6]=rk5[i+6];
        r10[i+6]=rk7[i+6];

        r1[i+10]=rk1[i+12];
        r4[i+10]=rk3[i+12];
        r7[i+10]=rk5[i+12];
        r10[i+10]=rk7[i+12];

        r1[i+14]=rk1[18];
        r4[i+14]=rk3[18];
        r7[i+14]=rk5[18];
        r10[i+14]=rk7[18];
    }
}
```

B.3 KEY EXPANSION FOR AES-256

```
void nextrk(int *rk1, int *rk2, int *rcon) {
    int i;
    int tempa[4], tempb[4], tempc[4];
    int tempd[4], tempe[4], tempf[4],tempg[4],temph[4];
    int rot[4];

    tempa[0] = rk1[0];tempb[0] = rk1[1];tempc[0] = rk1[2];
    tempa[1] = rk1[8];tempb[1] = rk1[9];tempc[1] = rk1[10];
    tempa[2] = rk1[16];tempb[2] = rk1[17];tempc[2] = rk1[18];
    tempa[3] = rk1[24];tempb[3] = rk1[25];tempc[3] = rk1[26];
    tempd[0] = rk1[3];tempe[0] = rk1[4];tempf[0] = rk1[5];
    tempd[1] = rk1[11];tempe[1] = rk1[12];tempf[1] = rk1[13];
    tempd[2] = rk1[19];tempe[2] = rk1[20];tempf[2] = rk1[21];
    tempd[3] = rk1[27];tempe[3] = rk1[28];tempf[3] = rk1[29];
    tempg[0] = rk1[6];temph[0] = rk1[7];
    tempg[1] = rk1[14];temph[1] = rk1[15];
    tempg[2] = rk1[22];temph[2] = rk1[23];
    tempg[3] = rk1[30];temph[3] = rk1[31];

    for (i = 0; i < 4; i++)  {
        rot[i] = temph[i];
    }
    shift(rot);
    subbyte4(rot);
    for (i = 0; i < 4; i++) {
        rot[i] = rot[i] ^ tempa[i] ^ rcon[i];
    }
    rk2[0] = rot[0];
    rk2[8] = rot[1];
    rk2[16] = rot[2];
    rk2[24] = rot[3];
    for (i = 0; i < 4; i++) {
        rot[i] = rot[i] ^ tempb[i];
     }
    rk2[1] = rot[0];
    rk2[9] = rot[1];
    rk2[17] = rot[2];
    rk2[25] = rot[3];
    for (i = 0; i < 4; i++) {
        rot[i] = rot[i] ^ tempc[i];
     }

    rk2[2] = rot[0];
    rk2[10] = rot[1];
    rk2[18] = rot[2];
    rk2[26] = rot[3];
```

```
        for (i = 0; i < 4; i++) {
               rot[i] = rot[i] ^ tempd[i];
         }
        rk2[3] = rot[0];
        rk2[11] = rot[1];
        rk2[19] = rot[2];
        rk2[27] = rot[3];
        subbyte4(rot);
        for (i = 0; i < 4; i++) {
               rot[i] = rot[i] ^ tempe[i];
        }
        rk2[4] = rot[0];
        rk2[12] = rot[1];
        rk2[20] = rot[2];
        rk2[28] = rot[3];
        for (i = 0; i < 4; i++) {
               rot[i] = rot[i] ^ tempf[i];
         }
        rk2[5] = rot[0];
        rk2[13] = rot[1];
        rk2[21] = rot[2];
        rk2[29] = rot[3];
        for (i = 0; i < 4; i++) {
               rot[i] = rot[i] ^ tempg[i];
        }
        rk2[6] = rot[0];
        rk2[14] = rot[1];
        rk2[22] = rot[2];
        rk2[30] = rot[3];
        for (i = 0; i < 4; i++) {
               rot[i] = rot[i] ^ temph[i];
         }
        rk2[7] = rot[0];
        rk2[15] = rot[1];
        rk2[23] = rot[2];
        rk2[31] = rot[3];
}
void keyexp(int *b, int *inr, int *r1,int *r2,int *r3,int *r4,int *r5,int *r6,
                          int *r7,int *r8,int *r9,int *r10,int *r11,int *r12,int *r13,int *r14 ) {
        int rk1[32],rk2[32],rk3[32],rk4[32],rk5[32],rk6[32],rk7[32];
        int rcon1[4] = { 1, 0, 0, 0 }; int rcon2[4] = { 2, 0, 0, 0 };
        int rcon3[4] = { 4, 0, 0, 0 }; int rcon4[4] = { 8, 0, 0, 0 };
        int rcon5[4] = { 16, 0, 0, 0 }; int rcon6[4] = { 32, 0, 0, 0 };
        int rcon7[4] = { 64, 0, 0, 0 };
        int i;
        nextrk(b,rk1,rcon1);
        nextrk(rk1,rk2,rcon2);
        nextrk(rk2,rk3,rcon3);
        nextrk(rk3,rk4,rcon4);
        nextrk(rk4,rk5,rcon5);
        nextrk(rk5,rk6,rcon6);
        nextrk(rk6,rk7,rcon7);
        for(i=0;i<4;i++) {
            inr[i]=b[i];
            inr[i+4]=b[i+8];
            inr[i+8]=b[i+16];
            inr[i+12]=b[i+24];

               r2[i]=rk1[i];
               r2[i+4]=rk1[i+8];
               r2[i+8]=rk1[i+16];
               r2[i+12]=rk1[i+24];

               r4[i]=rk2[i];
               r4[i+4]=rk2[i+8];
               r4[i+8]=rk2[i+16];
               r4[i+12]=rk2[i+24];

               r6[i]=rk3[i];
               r6[i+4]=rk3[i+8];
               r6[i+8]=rk3[i+16];
               r6[i+12]=rk3[i+24];

               r8[i]=rk4[i];
               r8[i+4]=rk4[i+8];
               r8[i+8]=rk4[i+16];
               r8[i+12]=rk4[i+24];

               r10[i]=rk5[i];
               r10[i+4]=rk5[i+8];
               r10[i+8]=rk5[i+16];
               r10[i+12]=rk5[i+24];

               r12[i]=rk6[i];
               r12[i+4]=rk6[i+8];
               r12[i+8]=rk6[i+16];
               r12[i+12]=rk6[i+24];

               r14[i]=rk7[i];
               r14[i+4]=rk7[i+8];
               r14[i+8]=rk7[i+16];
               r14[i+12]=rk7[i+24];

               r3[i]=rk1[i+4];
               r3[i+4]=rk1[i+12];
               r3[i+8]=rk1[i+20];
               r3[i+12]=rk1[i+28];
```

```
            r5[i]=rk2[i+4];
            r5[i+4]=rk2[i+12];
            r5[i+8]=rk2[i+20];
            r5[i+12]=rk2[i+28];

            r7[i]=rk3[i+4];
            r7[i+4]=rk3[i+12];
            r7[i+8]=rk3[i+20];
            r7[i+12]=rk3[i+28];

            r1[i]=b[i+4];
            r1[i+4]=b[i+12];
            r1[i+8]=b[i+20];
            r1[i+12]=b[i+28];


            r9[i]=rk4[i+4];
            r9[i+4]=rk4[i+12];
            r9[i+8]=rk4[i+20];
            r9[i+12]=rk4[i+28];

            r11[i]=rk5[i+4];
            r11[i+4]=rk5[i+12];
            r11[i+8]=rk5[i+20];
            r11[i+12]=rk5[i+28];

            r13[i]=rk6[i+4];
            r13[i+4]=rk6[i+12];
            r13[i+8]=rk6[i+20];
            r13[i+12]=rk6[i+28];

        }

}
```

B.4 AES-128 USING CUDA ON GPU

This section shows the CUDA program for AES-128 on GPU for granularity level 1, such that each thread takes 16 bytes input data and 16 bytes cipher-key. The definition for the functions is same as implemented on the CPU.

AES-192 and AES-256 are implemented similarly for key size 24 bytes and 32 bytes respectively. The key expansion as mentioned in previous section is different for the three versions. The three version of AES algorithm is implemented using granularity 1, granularity 2 (Each thread takes 32 bytes data), granularity 10 (Each thread takes 160 bytes data) and granularity 100 (Each thread takes 1600 bytes data). For every granularity level, the experiment is repeated with different grid dimensions as described in Chapter 3.

```
#define N 32000
int key[16];
int rk1[16]; int rk2[16]; int rk3[16]; int rk4[16]; int rk5[16]; int rk6[16];
int rk7[16]; int rk8[16]; int rk9[16]; int rk10[16];

__device__ const int sbox[16][16]                                   //S-Box
__device__ const int m2[16][16                                      //Multiplication 2
table
__device__ const int m3[16][16                                      //Multiplication 3
table

__device__ void subbyte(int *a)                                     //Substitute byte
__device__ void addroundkey(int *a, int *b)                         //Add Round Key
__device__ void shiftrow(int *a)                                    //Shift Row
__device__ void mixcolumn(int *a)                                   // Mix Column
__device__ void keyexp(int *cipherkey, int *rk1, int *rk2, int *rk3, int *rk4, int *rk5, int *rk6, int *rk7, int
*rk8, int *rk9, int *rk10)                                          //Key Expansion

__global__ void encryption(int *a, int *b) {                        //Encryption
        int idx=blockIdx.x*blockDim.x+threadIdx.x;
        int *data=&a[16*idx];

        keyexp(key,rk1,rk2,rk3,rk4,rk5,rk6,rk7,rk8,rk9,rk10);        //Key Expansion

        addroundkey(data, key);                                     //Initial Round

        subbyte(data);                                              //Round 1
        shiftrow(data);
        mixcolumn(data);
        addroundkey(data,rk1);

        subbyte(data);                                              //Round 2
        shiftrow(data);
        mixcolumn(data);
        addroundkey(data,rk2);
        subbyte(data);                                          //Round 3
        shiftrow(data);
        mixcolumn(data);
        addroundkey(data,rk3);

        subbyte(data);                                          //Round 4
        shiftrow(data);
        mixcolumn(data);
        addroundkey(data,rk4);


        subbyte(data);                                          //Round 5
        shiftrow(data);
        mixcolumn(data);
        addroundkey(data,rk5);

        subbyte(data);                                          //Round 6
        shiftrow(data);
        mixcolumn(data);
        addroundkey(data,rk6);

        subbyte(data);                                          //Round 7
        shiftrow(data);
        mixcolumn(data);
        addroundkey(data,rk7);

        subbyte(data);                                          //Round 8
        shiftrow(data);
        mixcolumn(data);
        addroundkey(data,rk8);

        subbyte(data);                                          //Round 9
        shiftrow(data);
        mixcolumn(data);
        addroundkey(data,rk9);

        subbyte(data);                                          //Round 10
        shiftrow(data);
        addroundkey(data,rk10);

}
```

```
int main() {
        int i;
        int a[N],k;
        for(i=0;i<N;i++){
            a[i]=i;
        }

        int *d_a,*d_b;

        cudaMalloc((void**)&d_a,N*sizeof(int));
        cudaMalloc((void**)&d_b,16*sizeof(int));

        double total_time;
        clock_t start, end;
        start = clock();
        for(k=0;k<n;k++){

        cudaMemcpy(d_a,&a,N*sizeof(int),cudaMemcpyHostToDevice);
        cudaMemcpy(d_b,&b,16*sizeof(int),cudaMemcpyHostToDevice);

        encryption<<<B,T>>>(d_a,d_b);

        cudaMemcpy(&a,d_a,N*sizeof(int),cudaMemcpyDeviceToHost);}

        end = clock();//time count stops
        total_time = ((double)(end - start)) / CLK_TCK;//calulate total time
        printf("\n\nTime taken to for aes128 encryption in cuda is: %f\n", total_time);

}
```

B.5 MULTI-THREADED PROGRAM USING POSIX THREADS

The function definitions remain same for this implementation as that in single threaded C program for all AES version.

This section shows the main program of multi-threaded C using POSIX THREADS for data size of 32000 bytes. The 2000 chunks of data is divided such that among the 12 virtual threads utilized, 10 threads handle 167 chunks each and 2 threads utilize 165 chunks each.

For data of size 32000*5 bytes, the data is divided such that among the 12 virtual threads utilized

```
#include <pthread.h>
#define state_ele 32000
#define num_threads 12
#define state_ele_s1 26720
#define state_ele_s2 5280
int state_main[state_ele],c[16];
int encryption(int);
void init() {
        int i;
        for(i=0;i<state_ele;i++)
        {
                state_main[i]=i;
        }
        return;
}
void *threadfunc(void *t) {
        int tmp,startpt, endpt;
        int thid, tmpstate[16];
        int i,j,*pmain,k;
        int *tp,*cp;
        thid=(int)t;
        if(thid<10)
        {
                tmp=state_ele_s1/(num_threads-2);
                startpt=thid*tmp;
                endpt=startpt+tmp;
        }
        else
        {
                tmp=(state_ele_s2/(num_threads-10));
                startpt=state_ele_s1 + (thid % 10)*tmp;
                endpt=startpt+tmp;
        }
        for(i=startpt;i<endpt;i=i+16)
        {
                for(j=0;j<16;j++)
        tmpstate[j]=state_main[i+j];
        tp=&tmpstate[0];
        pmain=encryption(tp);
        for(k=0;k<16;k++)
                c[k]=*(pmain+k);
        }
        pthread_exit((void*) t);
        }

int main (int argc, char *argv[]) {
        pthread_t thread[num_threads];
        pthread_attr_t attr;
        double total_time;
        clock_t start, end;
        int t, rc,l,k;
        void *status;
        init();
        start = clock();                                                    //time record starts
         for(k=0; k<n ;k++) {                                    //encryption performed 'n' no of times'
            pthread_attr_init(&attr);
            pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);

            for(t=0; t<num_threads; t++) {
                rc = pthread_create(&thread[t], &attr, threadfunc, (void *)t);
                if (rc)
                {
                        printf("ERROR; return code from pthread_create() is %d\n", rc);
                        exit(-1);
                }
             }
            pthread_attr_destroy(&attr);
            for(t=0; t<num_threads; t++)
            {
                rc = pthread_join(thread[t], &status);
                if (rc)
                {
                        printf("ERROR; return code from pthread_join() is %d\n", rc);
                        exit(-1);
                }
             }
        }                                                                           // k ends
        end = clock();                                                            //time record stops
        total_time = ((double)(end - start)) / CLK_TCK;
        printf("\n\nTime taken to for aes execution using pthreads is: %f\n", total_time);
        pthread_exit(NULL);
}
```

B.4 CUDA PROGRAM USING CUDA STREAMS

The function definitions remain same as that of CUDA program without streams. This section shows the main program for CUDA using CUDA STREAMS. The number of streams utilized is 2 such that each stream processes 32000 bytes of data.

```
int main() {
        int i;
        int h_a0[32000],h_a1[32000],k;
        cudaStream_t stream0,stream1;
        cudaStreamCreate(&stream0);
        cudaStreamCreate(&stream1);
        int *d_a0,*d_b;
        int *d_a1;
        for(i=0;i<32000;i++){
                h_a0[i]=i;
        }
        for(i=0;i<32000;i++){
                h_a1[i]=i+16000;
        }
        cudaMalloc((void**)&d_a0,32000*sizeof(int));
        cudaMalloc((void**)&d_a1,32000*sizeof(int));
        cudaMalloc((void**)&d_b,16*sizeof(int));
        double total_time;
        clock_t start, end;
        start = clock();
        for(k=0;k<10;k++){
                cudaMemcpy(d_b,&b,16*sizeof(int),cudaMemcpyHostToDevice);

                cudaMemcpyAsync(d_a0,&h_a0,32000*sizeof(int),cudaMemcpyHostToDevice,stream0);
                cudaMemcpyAsync(d_a1,&h_a1,32000*sizeof(int),cudaMemcpyHostToDevice,stream1);

                encryption<<<B,T,0,stream0>>>(d_a0,d_b);
                encryption<<<B,T,0,stream1>>>(d_a1,d_b);

                cudaMemcpyAsync(&h_a0,d_a0,32000*sizeof(int),cudaMemcpyDeviceToHost,stream0);
                cudaMemcpyAsync(&h_a1,d_a1,32000*sizeof(int),cudaMemcpyDeviceToHost,stream1);
        }
        end = clock();                                                  //time count stops
        total_time = ((double)(end - start)) / CLK_TCK;                 //calculate total time
        printf("\n\nTime taken to for aes128 encryption in cudastreams is: %f\n", total_time);
}
```