NATIONAL AND KAPODISTRIAN UNIVERSITY OF ATHENS

SCHOOL OF SCIENCE
DEPARTMENT OF INFORMATICS AND TELECOMMUNICATIONS

THESIS

IMPLEMENTATION OF THE AES ENCRYPTION ALGORITHM IN PARALLEL GPU AND CPU ARCHITECTURES

George Nikolaou Gousios
Nikolaos Anastasiou Dimizas

Advisor: Dimitris Gizopoulos, Professor

ATHENS

NOVEMBER 2015

ΕΘΝΙΚΟ ΚΑΙ ΚΑΠΟΔΙΣΤΡΙΑΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ

ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ
ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

ΥΛΟΠΟΙΗΣΗ ΤΟΥ ΑΛΓΟΡΙΘΜΟΥ ΚΡΥΠΤΟΓΡΑΦΗΣΗΣ AES ΣΕ ΠΑΡΑΛΛΗΛΕΣ ΑΡΧΙΤΕΚΤΟΝΙΚΕΣ CPU ΚΑΙ GPU

Γεώργιος Νικολάου Γούσιος
Νικόλαος Αναστασίου Δήμιζας

Επιβλέπων: Δημήτρης Γκιζόπουλος, Καθηγητής

ΑΘΗΝΑ

ΝΟΕΜΒΡΙΟΣ 2015

THESIS

IMPLEMENTATION OF THE AES ENCRYPTION ALGORITHM IN PARALLEL GPU AND CPU ARCHITECTURES

GOUSIOS GEORGE

A.M.: 1115201000031

DIMIZAS NIKOLAOS

A.M: 1115201000017

ADVISOR: **Dimitris Gizopoulos, Professor**

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

ΥΛΟΠΟΙΗΣΗ ΤΟΥ ΑΛΓΟΡΙΘΜΟΥ ΚΡΥΠΤΟΓΡΑΦΗΣΗΣ AES ΣΕ ΠΑΡΑΛΛΗΛΕΣ ΑΡΧΙΤΕΚΤΟΝΙΚΕΣ CPU ΚΑΙ GPU

ΓΟΥΣΙΟΣ ΓΕΩΡΓΙΟΣ

Α.Μ.: 1115201000031

ΔΗΜΙΖΑΣ ΝΙΚΟΛΑΟΣ

Α.Μ: 1115201000017

ΕΠΙΒΛΕΠΩΝ: **Δημήτρης Γκιζόπουλος, Καθηγητής**

ABSTRACT

The subject of this thesis is the implementation of the AES encryption algorithm in CUDA parallel code, aiming a significant acceleration over the original serial (C language) code. Parallel software development was realized using a baseline serial C code for the AES algorithm, though many changes have taken place, in spite of the similarity of the two implementations.

In the beginning of the thesis, we were called to find a way to write code which would have identical functionality to the C code used as a baseline. Though the initial code was in C and CUDA supports C and C++ code, which is something that would make the production of new code seem easy, the main problem was finding a way to make proper use of all available CUDA threads and obtain the best possible acceleration, without removing any features of the algorithm or reducing its functionality.

After the finalization and validation of the CUDA code, we implemented performance optimizations. Finally we developed some tests to determine the actual (real-time, not theoretical) acceleration to an Encryption-Decryption procedure, performed several (10/100/1000) times. Results confirmed our intuition. In conclusion, certain variants of the AES encryption algorithm can be accelerated by GPUs obtaining significantly improved performance, which could reach acceleration levels up to 70 times compared to the baseline serial code.

SYBJECT AREAS: Computer Architecture, Parallel Programming, Cryptography

KEY WORDS: Many-core Processors, GPGPU computing, Encryption/Decryption, CUDA language, Performance computing, AES

ΠΕΡΙΛΗΨΗ

Το αντικείμενο της παρούσας πτυχιακής εργασίας είναι η υλοποίηση του αλγορίθμου κρυπτογράφησης AES με χρήση CUDA παράλληλου κώδικα, με κύριο στόχο την επίτευξη σημαντικής επιτάχυνσης του αλγορίθμου, σε σχέση με την σειριακή υλοποίησή του. Για την υλοποίηση του λογισμικού, χρησιμοποιήθηκε ο αντίστοιχος κώδικας σε C ως βάση, αν και ενσωματώθηκαν αρκετές αλλαγές, παρ'όλη την συνάφεια που παρουσιάζει η C με την CUDA ως γλώσσες προγραμματισμού.

Στην αρχή της ανάπτυξης του κώδικα, καλούμασταν να βρούμε έναν τρόπο να χρησιμοποιήσουμε την CUDA για να παράγουμε ένα πρόγραμμα το οποίο θα είχε ακριβώς την ίδια λειτουργικότητα με τον αρχικό σειριακό. Παρ'ότι αυτό μπορεί να φαίνεται απλό λόγω της ομοιότητας της C με την CUDA, το πραγματικό ζήτημα ήταν να βρούμε έναν τρόπο ώστε να αξιοποιήσουμε όσο δυνατόν καλύτερα το πλήθος των CUDA threads έτσι ώστε να πετύχουμε την καλύτερη δυνατή επιτάχυνση, χωρίς όμως παράλληλα να θυσιαστούν οποιεσδήποτε λειτουργίες του λογισμικού ή να μειωθεί η λειτουργικότητά του.

Μετά την ανάπτυξη του CUDA κώδικα, συμπεριλήφθησαν κάποιες διορθώσεις και βελτιστοποιήσεις στο πρόγραμμά μας, έτσι ώστε να μειωθούν κατά το δυνατό οι περιττές και χρονοβόρες διαδικασίες. Στη συνέχεια, συμπεριλάβαμε κάποια εκτελέσιμα tests με σκοπό να μετρήσουμε στην πράξη την επιτάχυνση σε έναν επαναλαμβανόμενο κύκλο Κρυπτογράφισης-Αποκρυπτογράφισης. Τα αποτελέσματα επαλήθευσαν τις αρχικές μας εκτιμήσεις. Τέλος, καταλήξαμε ότι ορισμένες μορφές του AES αλγορίθμου μπορούν να επιταχυνθούν σε σημαντικό βαθμό, έτσι ώστε να ολοκληρώνονται ακόμα και 70 φορές πιο γρήγορα απ'τον σειριακό C κώδικα.

ΘΕΜΑΤΙΚΗ ΕΝΟΤΗΤΑ: Αρχιτεκτονική Υπολογιστών, Παράλληλος Προγραμματισμός, Κρυπτογραφία

KEY WORDS: Πολυπύρηνοι επεξεργαστές, Προγραμματισμός GPGPU, Κρυπτογράφηση/Αποκρυπτογράφηση, Γλώσσα προγραμματισμού CUDA, Επιτάχυνση, AES

TABLE OF CONTENTS

FIGURES’ INDEX

TABLES' INDEX

PROLOGUE

The current document is the thesis of George Gousios and Nikolaos Dimizas, as part of the undergraduate study program of the Department of Informatics and Telecommunications (D.I.T) of the National and Kapodistrian University of Athens (abbreviated in Greek as “ΕΚΠΑ”). The current project was developed and tested using a remote server equipped with a CUDA-enabled NVIDIA GPU. On our end, we used Linux based distributions to develop the code and connected to the aforementioned server via the ssh protocol.

For the completion of the current thesis, we would like to thank our advisor Professor, Dimitris Gizopoulos and the department’s PhD candidate Stamos Katsigiannis for their cooperation, advice and their valuable contribution to the successful completion of this project.

1. INTRODUCTION

Subject and Goals of the Thesis

The current thesis focuses on the development of parallel programs for AES-based encryption in the CUDA language for GPUs. The variant of AES algorithm we have used is the AES ECB, which has been implemented previously in C language **[4].**

With the vast amount of data in PCs, servers, laptops, smartphones, tablets, etc. today, and the continuous expansion of the Internet and the way people use it, (data transfers, instant messaging, etc.), it's been clearer than ever that there is a growing need for security and data privacy against attackers. Personal information leakage is becoming more dangerous than ever, and unreliability of used tools is rendered unacceptable, to such extend that cryptography is implemented in the vast majority of the world's applications that use the Internet (or any type of network) as a means of communication. For example, Skype uses end-to-end cryptography using the AES algorithm among other methods **[5]**. It's even worth mentioning that cryptography is also used on offline applications as a means of extra privacy.

How can we make sure that such a compute-intensive task can be performed as fast as possible? While the CPU frequency is a significant factor on a system's processing power and therefore execution capability, the effort of the computing industry to keep Moore's Law valid for more years to come has driven the integration of more processing cores per chip (either CPU or GPU chip), in order for the hardware to be able to keep up with the execution needs.

Throughout the years, different approaches of optimization have been made, many of those concerning not only the CPU, but also the GPU. Such approaches can be useful for many applications. In fact, many systems nowadays include several dozens (or even hundreds) of CPUs and GPUs to meet the execution demands. The hardware compatibility is not enough though; there has to be proper software development in order to take full advantage of the techniques that can be used. Software APIs (such as POSIX, OpenMP, CUDA, OpenCL, etc, depending on the goals of the developer) are commonly included in nowadays' software from the start of their production.

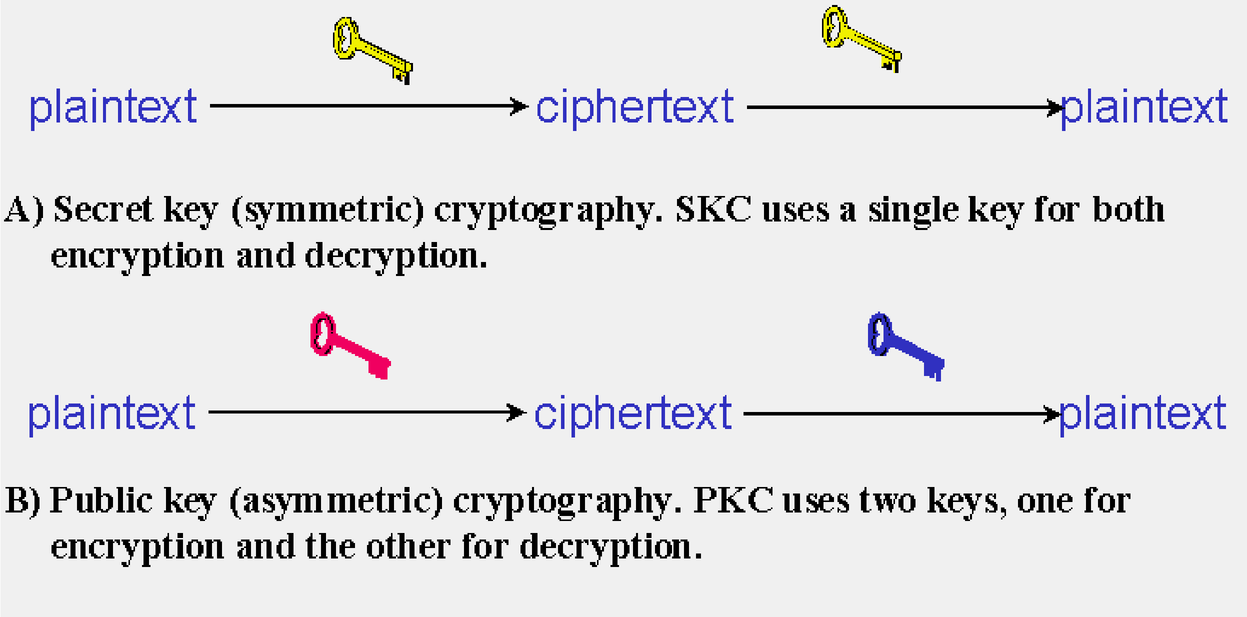
Cryptography is usually a service extension (or even a service on its own), thus it is a burden on the computer (client computer or server depending on whether client-side or server-side encryption is used). Parallel programming is a way to optimize (often rewrite) the serial code of the task in order to make better use of the hardware and as a result complete computing tasks faster. So how can we use parallel programming to achieve that in the case of cryptography? That is what remains to be examined through the rest of this thesis.

2. ENCRYPTION ALGORITHMS

Since the very first days that encryption started to be implemented in applications, many efforts have taken place to find a secure way to encrypt data, but also in a way that the whole process is efficient. Once again different approaches resulted in several cipher categories:

2.1 Encryption Algorithm Categories

a)Symmetric/Asymmetric Ciphers



A) Secret key (symmetric) cryptography. SKC uses a single key for both encryption and decryption.

B) Public key (asymmetric) cryptography. PKC uses two keys, one for encryption and the other for decryption.

Figure 1: Encryption process

i) Symmetric encryption is the oldest and best-known technique. A secret key, which can be a number, a word, or just a string of random letters, is applied to the text of a message to change the content in a particular way. This might be as simple as shifting each letter by a number of positions in the alphabet. As long as both the sender and the recipient know the secret key, they can encrypt and decrypt all messages that use this key. That means that anyone who has the key can use it to decrypt the cipher and have access to the data the two participants of the communication try to protect. As a result, Symmetric Encryption is quite simple as well as very dangerous. Usually the key is being distributed via a public network (e.g. the Internet) and a privacy breach is very likely, should the key fall into the wrong hands.

ii) Asymmetric encryption solves the aforementioned security problem. In an Asymmetric algorithm there are two keys instead of one, which are related in a way that they are considered a pair. The first key – the *public* key – is made available to anyone who wants to send a message to person A. The second key – the *private* key – is kept secret so that only person A knows it. Any message encrypted with the public key can be decrypted (using exactly the same algorithm) with the private key, and vice versa.

So, if Asymmetric encryption is so much safer, why are Symmetric algorithms way more popular? That is because Asymmetric encryption is much slower and requires far more processing power for both encryption and decryption of the message.

b) Block and Stream Ciphers

Block and Stream ciphers are a sub-category of Symmetric Ciphers.

i) Block ciphers encrypt a group of plaintext symbols (called a block) with a fixed size (e.g. 128-bit). The encoding of each block may or may not depend on any of the previous blocks. It should be noted that the same key is used to encrypt every block of the text. The DES (Data Encryption Standard) and the AES (Advanced Encryption Standard **[9]**) algorithms are perfect examples of Symmetric Block Ciphers.

ii) Stream ciphers convert one symbol of plaintext directly into a symbol of ciphertext. The encoding of each block may or may not depend on any of the previous blocks, as well. For each symbol, a different key is generated and used.

Both Block and Stream ciphers have their pros and cons and that is why they are used in somewhat different situations, according to the needs of the application.

Block ciphers have high diffusion (information from one plaintext symbol affects several ciphertext symbols – the whole block it belongs to) and they have higher immunity to tampering (it's more difficult to insert symbols without detection). On the other hand, they are slower, (the entire block must be accumulated before encryption or decryption can begin), and an error in one symbol can corrupt the entire block.

Stream ciphers are faster in general (linear in time and constant in space) and have low error propagation (an error in encrypting one symbol will most likely not affect subsequent symbols). Their disadvantage lies on the fact that they have low diffusion (all information of a plaintext symbol is contained in a single ciphertext symbol) and that they are susceptible to insertions/modifications (a potential attacker can insert spurious text that looks authentic).

There are also other categories of cipher algorithms that are out of the scope of this thesis and therefore will not be examined.

2.2 Block Cipher Modes

An Encryption algorithm can be paired with a block cipher mode of operation to determine the way the algorithm is being applied to a file that contains more than 1 blocks of data. Below we present some of the basic categories of Block Cipher Modes **[12]**.

i)ECB (Electronic Codebook): the file is divided into blocks and each block is encrypted separately (which means there are no dependencies between blocks of the file). ECB has the disadvantage that identical plaintext blocks are encrypted into identical ciphertext blocks, which means that it does not hide data patters well. It is considered the simplest cipher mode though.

ii)CBC (Cipher Block Chaining): an IBM invention from 1976. In this mode, each block of plaintext is XORed with the previous ciphertext block before being encrypted. This way, each ciphertext block depends on all plaintext blocks processed up to that point. CBC also makes use of an initialization vector in the first block, in order to make each message unique.

iii)GCM (Galois/Counter Mode): widely adopted because of its efficiency and performance. It also includes authentication code for the message and is designed to provide both data authenticity (integrity) and confidentiality. It has minimum latency, minimum operation overhead and its throughput rates are considered state of the art,

2.3 The AES encryption algorithm

Figure 2: AES logo

As we mentioned above, the AES (Advanced Encryption Standard) Encryption Algorithm is a symmetric block cipher used on many applications throughout the world. It is also known as the Rijndael algorithm though that name refers to a family of cipher algorithms with different block sizes and key lengths. As far as the AES algorithm is concerned, it includes three members of the Rijndael family, each one having a 128-bit, 196-bit and 256-bit key respectively, as well as a fixed block size of 16 bytes (=128 bits).

From a historical point of view, the AES algorithm is a straight evolution from the DES (Data Encryption Standard) algorithm. The DES algorithm (a symmetric block cipher that has a Feistel structure **[16]**), which was published as the Federal Information Processing Standards (FIPS) 46 standard in 1977, used a fixed 56-bit key and 64-bit block size. Attempts had been made through the years to crack it and several of these were successful and within reasonable time limits. Through the years the DES's security was questioned, with the main argument being that the 56-bit key used was too short. During the 90's, the RSA (Rivest, Shamir and Adelman) conducted a series of cipher crack challenges to determine whether the algorithm was sufficient in terms of security. As a result, in 1999 (the 3rd and final RSA challenge to crack the DES), the message

"See you in Rome (Second AES Candidate Conference, March 22-23, 1999" was cracked in a little more than 22 hours, indicating the redundancy of DES and pointing out the need of a more advanced and secure cipher – DES belonged to the past.

There were certain approaches to address DES's security issues, the most important of which being the 3DES algorithm. On this approach, the DES algorithm is applied three times on each block, and the key size is increased in most cases, since three separate keys of 56 bits are used that may or may not be identical. As a result the key size of the algorithm can be 56 bits (all three keys are identical), 112 bits (two of the keys are identical) or 168 bits (all keys are different). The increase of the key size did not solve the problem however, as due to certain vulnerabilities when reapplying the same encryption three times, using 168 bits has a reduced security equivalent to 112 bits and using 112 bits has a reduced security equivalent to 56 bits. That is one of the reasons 3DES was questioned in terms of security, and therefore wasn't a preferable option when other algorithms emerged.

In 2000, the AES algorithm was introduced with several advantages over its predecessor, such as the choice between 128-, 196- and 256-bit key sizes and a more mathematically efficient and elegant cryptographic algorithm. Since then, the AES algorithm is implemented on both software and hardware units and is considered fairly secure up to this day, which means that is admittedly difficult to break using conventional computing resources. Its reliability is verified by the fact that it was initially selected for use within the US government **[9]** and nowadays it is used almost everywhere, including most wireless networks.

The cipher optimally uses some pre-calculated tables that store values which are being used throughout the encryption rounds. These tables are often called S-boxes, Rcons, etc. The reversed tables are being used in the decryption stage of the algorithm. Each of these tables has a certain purpose in the program. In fact it is accessed on a specific step of the algorithm.

One reason that AES was better in general than 3DES is that 3DES uses 64 bit blocks, the same as DES, while AES uses 128 bit blocks, which means that using AES provides additional insurance that it is harder to sniff leaked data from identical blocks. When using 3DES, the user needs to switch encryption keys every 32GB of data transfer to minimize the possibility of leaks; identical to when using the standard DES encryption. Last but not least, AES proved itself to be much faster than 3DES. Of course, this is also a matter of hardware configuration as well as optimization, but in general, that point stands **[31].**

KEY SIZE	POSSIBLE COMBINATIONS
1-bit	2
2-bit	4
4-bit	16
8-bit	256
16-bit	65536
32-bit	4.2×10^{9}
54-bit (DES)	7.2×10^{16}
64-bit	1.8×10^{19}
128-bit (AES)	3.4×10^{38}
192-bit (AES)	6.2×10^{57}
256-bit (AES)	1.1×10^{77}

Table 1: Key size vs Possible Combinations to break the cipher using brute force attack[32]

2.3.1 Steps of the AES cipher

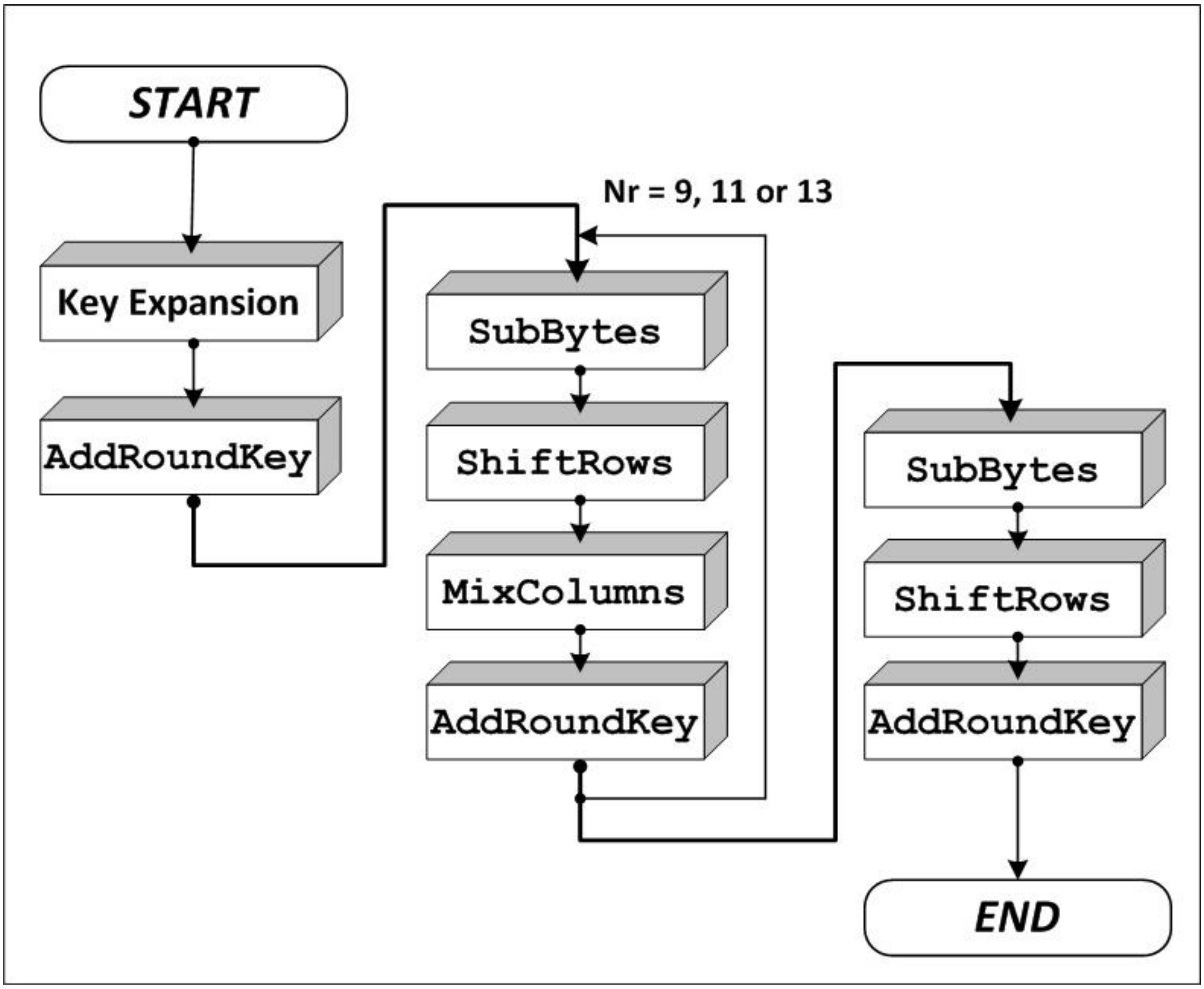


Figure 3: Steps of AES algorithm

The AES algorithm consists of a certain number of rounds of encryption on each block, which is 10 for 128-bit keys, 12 for 192-bit keys and 14 for 256-bit keys. These rounds consist of four steps and are all identical except for the last round in each case. For future reference, the state table is referred to the state of the block that is being encrypted at a specific moment of the encryption process.

Before the rounds start, an initialization process is performed, which is called **Key Expansion**. During this process, the key is expanded into another key (though the resulting key is not always longer), whose parts are used through different iterations. This key is often referred to as the expanded key. The size of the new key can be calculated by multiplying 16-bits with the number of rounds that are going to be performed plus 1 (an initial AddRoundKey operation, will be explained later). So we have:

176-byte for an initial 128-bit key :(16*(10+1))

208-byte for an initial 192-bit key :(16*(12+1))

240-byte for an initial 256-bit key :(16*(14+1))

Key size (words/bytes/bits)	4/16/128	6/24/192	8/32/256
Plaintext block size (words/bytes/bits)	4/16/128	4/16/128	4/16/128
Number of rounds	10	12	14
Round key size (words/bytes/bits)	4/16/128	4/16/128	4/16/128
Expanded key size (words/bytes)	44/176	52/208	60/240

Table 2: AES key specifications

A quick review of the basic steps of each round includes the following:

i)SubBytes: the first step of each round, the algorithm uses the S-box lookup table to perform a byte-by-byte substitution of the block.

ii)ShiftRows: the second step of each round. Shift Rows is a simple permutation to scramble the byte order inside each 128-bit block.

iii)MixColumns: The third step aims to mix up the bytes in each column separately. It further scrambles up the 128-bit input block, using the arithmetic GF (2^8)

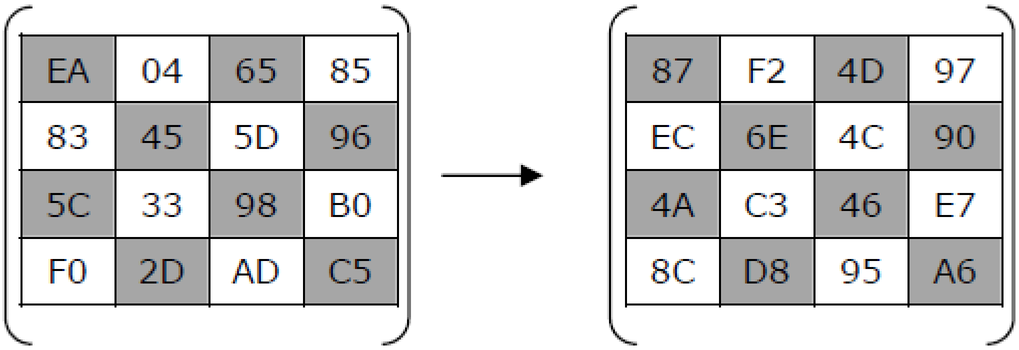
Note: Steps 2 and 3 causes each bit of the ciphertext to depend on every bit of the plain-text after 10 rounds of processing.

iv)AddRoundKey: The final step of each round. Each of the 16 bytes of the state is XORed against each of the 16 bytes of a portion of the expanded key for the current round. The Expanded Key bytes are never reused. So once the first 16 bytes are XORed against the first 16 bytes of the expanded key then the expanded key bytes 1-16 are never used again. The next time the Add Round Key function is called bytes 17-32 are XORed against the state. This step is also executed once in the start of the cipher , after the Key Expansion step.

Now let’s take a further insight on each of those operations:

a)SubBytes Transformation

The forward substitute byte transformation, called SubBytes, is a simple table lookup. AES defines a 16X16 matrix of byte values (the S-box we mentioned earlier). This table contains a permutation of all possible 256 8-bit values. Each individual byte of State is directly mapped into a new byte in the S-box in the following way: The leftmost 4 bits of the byte are used as a row value and the rightmost 4 bits are used as a column value. These row and column values serve as indexes into the S-box to select a unique 8-bit output value. For example, the hexadecimal value {95} references row 9, column 5 of the S-box, which contains the value {2A}. Accordingly, the value {95} is mapped into the value {2A}. Here is an example of the SubBytes transformation:



EA	04	65	85
83	45	5D	96
5C	33	98	B0
F0	2D	AD	C5

87	F2	4D	97
EC	6E	4C	90
4A	C3	46	E7
8C	D8	95	A6

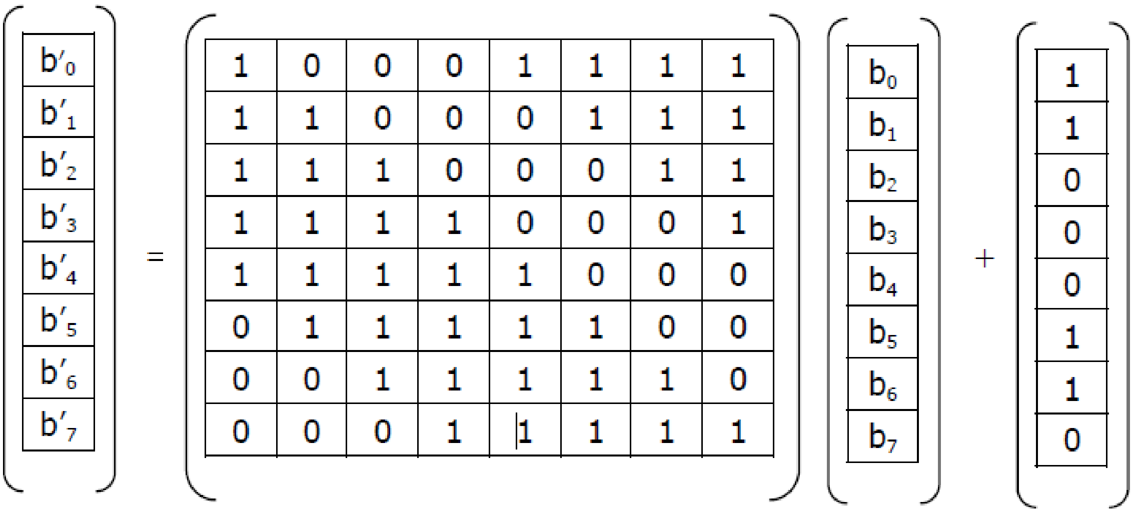
Figure 4: SubBytes 1

The S-box is constructed in the following way:

1. Initialize the S-box with the byte values in ascending sequence row by row. The first row contains {00}, {01},…,{0F}, the second row contains {10}, {11},…,{1F} and so on. Thus, the value of the byte at row x, column y is {xy}.
2. Map each byte in the S-box to its multiplicative inverse in the finite field of GF(2^8). The value {00} is mapped to itself,
3. Each byte in the S-box consists of 8 bits labeled {b7,b6.b5,b4,b3,b2,b1,b0}. The following transformation is applied to each bit of each byte in the S-box:

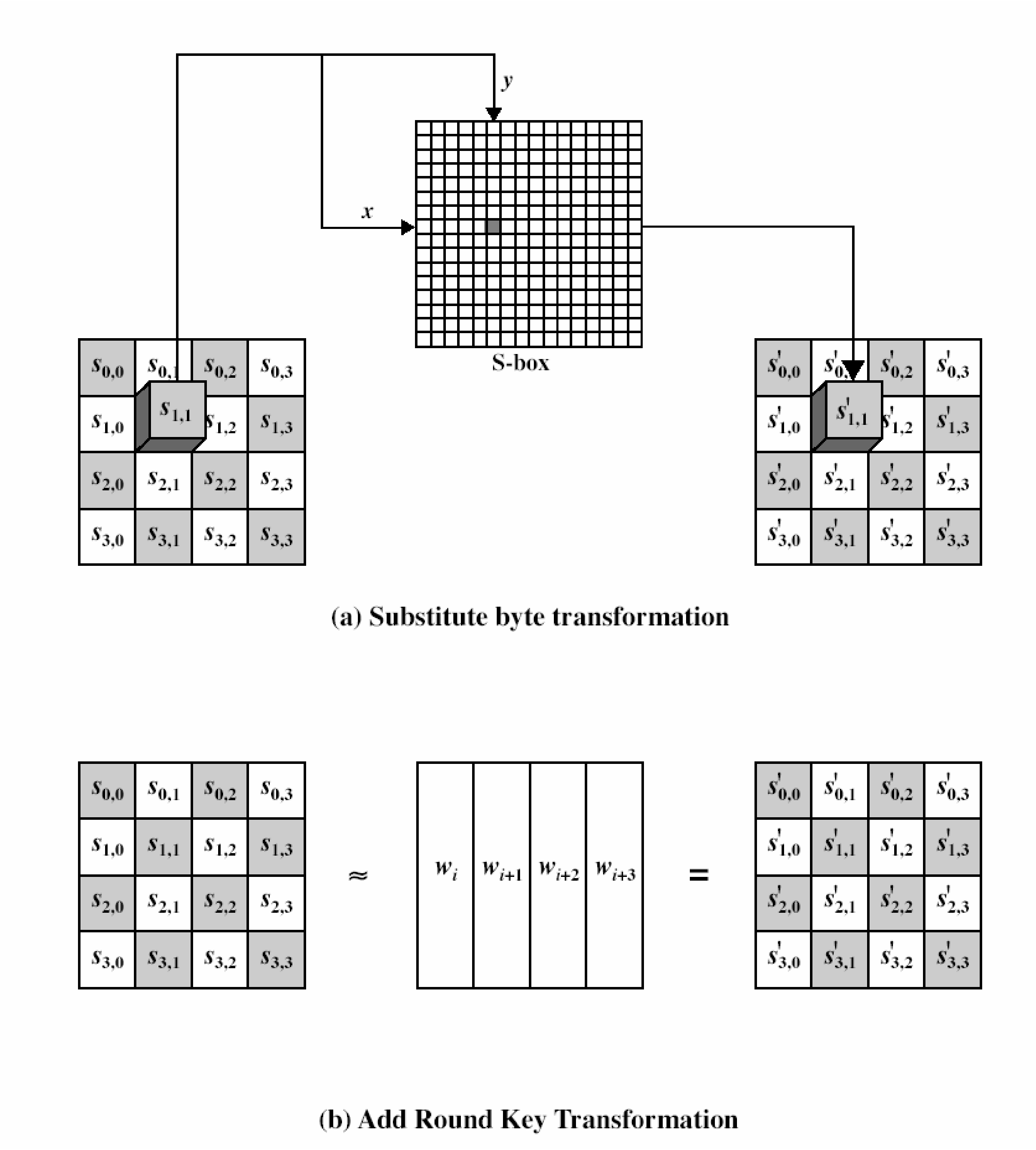
   $$b_i' = b_i \oplus b_{(i+4) \bmod 8} \oplus b_{(i+5) \bmod 8} \oplus b_{(i+6) \bmod 8} \oplus b_{(i+7) \bmod 8} \oplus c_i ,$$

   Where Ci is the i-th bit of byte C with the value {63}. That is (C7C6C5C4C3C2C1C0) = (01100011). The prime (‘) indicates that the variable is to be updated by the value on the right. The AES standard depicts this transformation in matrix form as follows:



$$\begin{pmatrix} b'_0 \\ b'_1 \\ b'_2 \\ b'_3 \\ b'_4 \\ b'_5 \\ b'_6 \\ b'_7 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \end{pmatrix} \begin{pmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \\ b_6 \\ b_7 \end{pmatrix} + \begin{pmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{pmatrix}$$

Figure 5: SubBytes 2

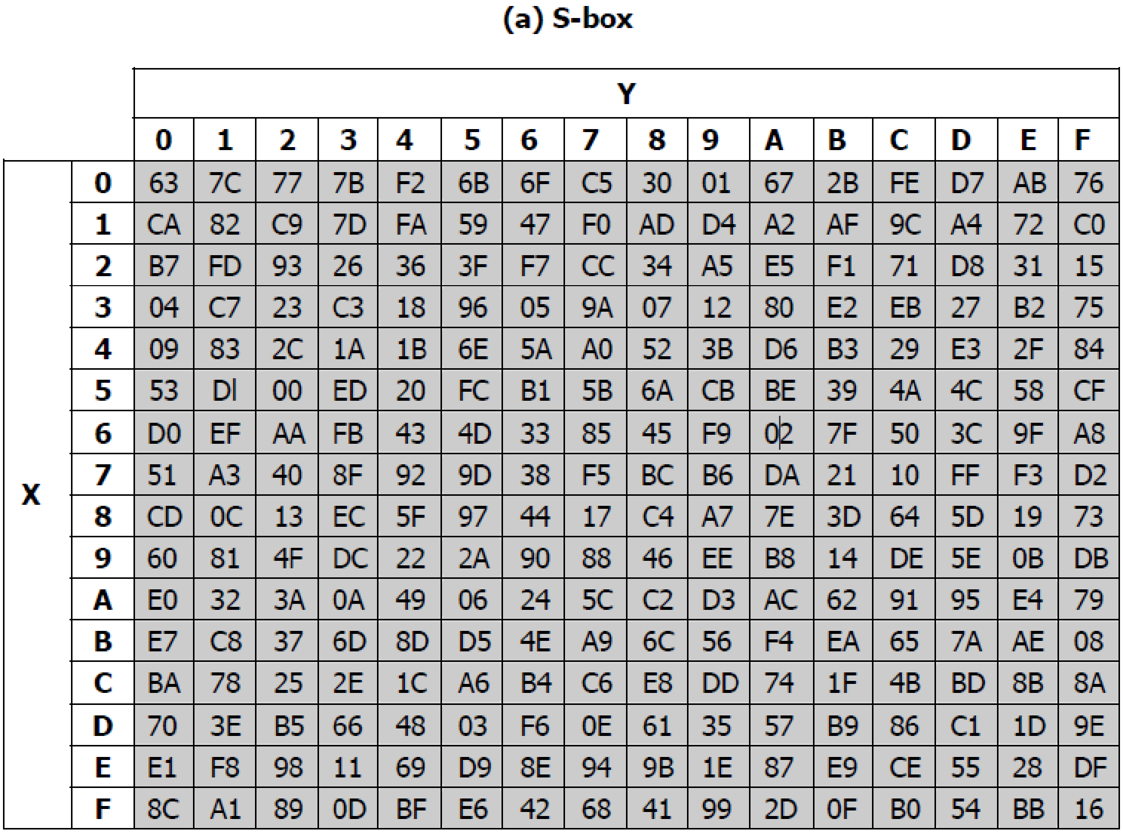

y
x
S-box
$s_{0,0}$ $s_{0,1}$ $s_{0,2}$ $s_{0,3}$
$s_{1,0}$ $s_{1,1}$ $s_{1,2}$ $s_{1,3}$
$s_{2,0}$ $s_{2,1}$ $s_{2,2}$ $s_{2,3}$
$s_{3,0}$ $s_{3,1}$ $s_{3,2}$ $s_{3,3}$
$s'_{0,0}$ $s'_{0,1}$ $s'_{0,2}$ $s'_{0,3}$
$s'_{1,0}$ $s'_{1,1}$ $s'_{1,2}$ $s'_{1,3}$
$s'_{2,0}$ $s'_{2,1}$ $s'_{2,2}$ $s'_{2,3}$
$s'_{3,0}$ $s'_{3,1}$ $s'_{3,2}$ $s'_{3,3}$
≈
w_i w_{i+1} w_{i+2} w_{i+3}
=

(a) Substitute byte transformation

(b) Add Round Key Transformation

Figure 6: SubBytes 3 and AddRoundKey

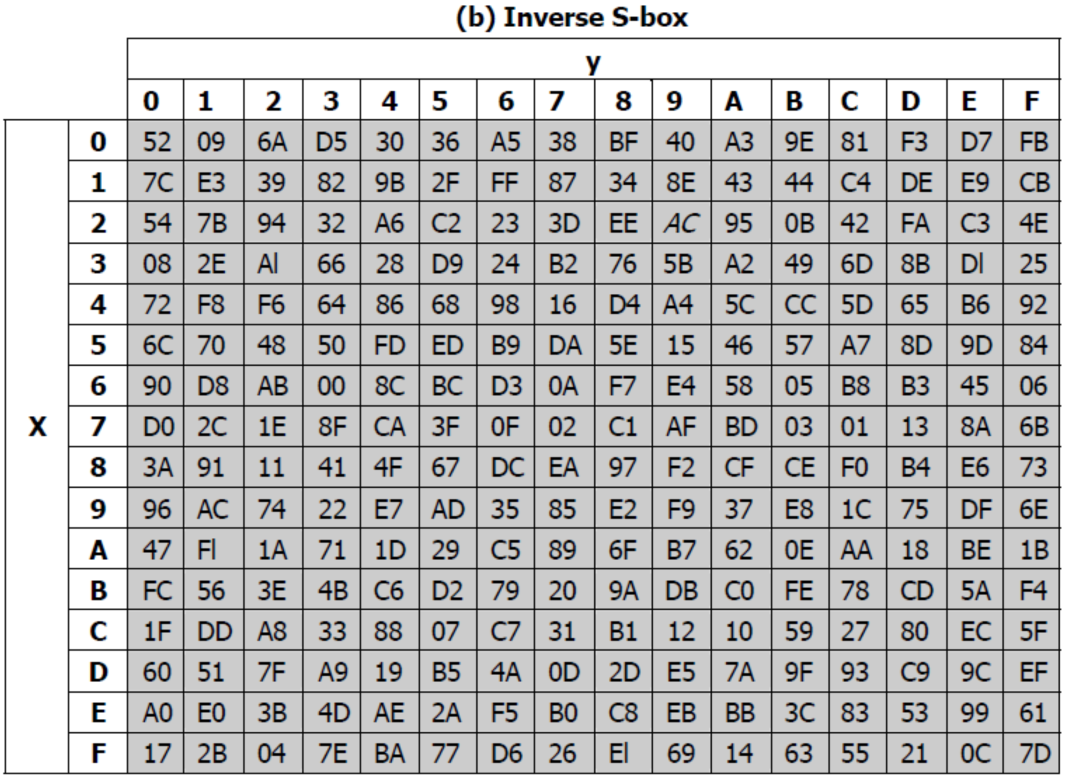
In ordinary matrix multiplication, each element in the product matrix is the sum of products of the elements or one row and one column. In this case, each element in the product matrix is the bitwise XOR of products of elements of one row and one column. Further, the final addition is a bitwise XOR.



(a) S-box

		Y															
		0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**A**	**B**	**C**	**D**	**E**	**F**
X	**0**	63	7C	77	7B	F2	6B	6F	C5	30	01	67	2B	FE	D7	AB	76
	1	CA	82	C9	7D	FA	59	47	F0	AD	D4	A2	AF	9C	A4	72	C0
	2	B7	FD	93	26	36	3F	F7	CC	34	A5	E5	F1	71	D8	31	15
	3	04	C7	23	C3	18	96	05	9A	07	12	80	E2	EB	27	B2	75
	4	09	83	2C	1A	1B	6E	5A	A0	52	3B	D6	B3	29	E3	2F	84
	5	53	Dl	00	ED	20	FC	B1	5B	6A	CB	BE	39	4A	4C	58	CF
	6	D0	EF	AA	FB	43	4D	33	85	45	F9	02	7F	50	3C	9F	A8
	7	51	A3	40	8F	92	9D	38	F5	BC	B6	DA	21	10	FF	F3	D2
	8	CD	0C	13	EC	5F	97	44	17	C4	A7	7E	3D	64	5D	19	73
	9	60	81	4F	DC	22	2A	90	88	46	EE	B8	14	DE	5E	0B	DB
	A	E0	32	3A	0A	49	06	24	5C	C2	D3	AC	62	91	95	E4	79
	B	E7	C8	37	6D	8D	D5	4E	A9	6C	56	F4	EA	65	7A	AE	08
	C	BA	78	25	2E	1C	A6	B4	C6	E8	DD	74	1F	4B	BD	8B	8A
	D	70	3E	B5	66	48	03	F6	0E	61	35	57	B9	86	C1	1D	9E
	E	E1	F8	98	11	69	D9	8E	94	9B	1E	87	E9	CE	55	28	DF
	F	8C	A1	89	0D	BF	E6	42	68	41	99	2D	0F	B0	54	BB	16

Figure 7: S-Box



(b) Inverse S-box

		y															
		0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**A**	**B**	**C**	**D**	**E**	**F**
X	**0**	52	09	6A	D5	30	36	A5	38	BF	40	A3	9E	81	F3	D7	FB
	1	7C	E3	39	82	9B	2F	FF	87	34	8E	43	44	C4	DE	E9	CB
	2	54	7B	94	32	A6	C2	23	3D	EE	*AC*	95	0B	42	FA	C3	4E
	3	08	2E	Al	66	28	D9	24	B2	76	5B	A2	49	6D	8B	Dl	25
	4	72	F8	F6	64	86	68	98	16	D4	A4	5C	CC	5D	65	B6	92
	5	6C	70	48	50	FD	ED	B9	DA	5E	15	46	57	A7	8D	9D	84
	6	90	D8	AB	00	8C	BC	D3	0A	F7	E4	58	05	B8	B3	45	06
	7	D0	2C	1E	8F	CA	3F	0F	02	C1	AF	BD	03	01	13	8A	6B
	8	3A	91	11	41	4F	67	DC	EA	97	F2	CF	CE	F0	B4	E6	73
	9	96	AC	74	22	E7	AD	35	85	E2	F9	37	E8	1C	75	DF	6E
	A	47	Fl	1A	71	1D	29	C5	89	6F	B7	62	0E	AA	18	BE	1B
	B	FC	56	3E	4B	C6	D2	79	20	9A	DB	C0	FE	78	CD	5A	F4
	C	1F	DD	A8	33	88	07	C7	31	B1	12	10	59	27	80	EC	5F
	D	60	51	7F	A9	19	B5	4A	0D	2D	E5	7A	9F	93	C9	9C	EF
	E	A0	E0	3B	4D	AE	2A	F5	B0	C8	EB	BB	3C	83	53	99	61
	F	17	2B	04	7E	BA	77	D6	26	El	69	14	63	55	21	0C	7D

Figure 8: Inverse S-Box

As an example, the input value {95} is considered. The multiplication inverse in GF(2^8) is $\{95\}^{-1}$=(8A), which is 10001010 in binary. Using the above equation, the result is (2A), which will appear in row {09}, column {05} of the S-box.

The inverse substitute byte transformation, called InvSubBytes, makes use of the inverse S-box (figure 8). The input {2A} produces the output {95} and the input {95} to the S-box produces {2A}. The inverse S-box is constructed by applying the inverse of the transformation in our previous equation, followed by taking the multiplicative inverse in GF(2^8). The inverse transformation is:

$$b_i' = b_i \oplus b_{(i+2) \bmod 8} \oplus b_{(i+5) \bmod 8} \oplus b_{(i+7) \bmod 8} \oplus \mathbf{d}_i\ ,$$

Where byte d={05}, or 00000101. It can be represented as follows:

$$\begin{pmatrix} b'_0 \\ b'_1 \\ b'_2 \\ b'_3 \\ b'_4 \\ b'_5 \\ b'_6 \\ b'_7 \end{pmatrix} = \begin{pmatrix} 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \\ b_6 \\ b_7 \end{pmatrix} + \begin{pmatrix} 1 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix}$$

To verify that InvSubBytes is the inverse of SubBytes, the matrices in SubBytes and InvSubBytes are labeled as X and Y respectively, and the vector versions of constants c and d are labeled as C and D, respectively. For some 8-bit vector B, our previous equation gives:

$$B' = \mathbf{XB} \oplus C$$

It must be proved that:

$$\mathbf{Y}(\mathbf{XB} \oplus \mathrm{C}) \oplus \mathrm{D} = \mathbf{B}$$

Multiply out, it must satisfy that:

$$\mathbf{Y}(\mathbf{XB} \oplus \mathrm{C}) \oplus \mathrm{D} = \mathbf{B}$$

This becomes:

$$\left(\begin{pmatrix} 0&0&1&0&0&1&0&1\\ 1&0&0&1&0&0&1&0\\ 0&1&0&0&1&0&0&1\\ 1&0&1&0&0&1&0&0\\ 0&1&0&1&0&0&1&0\\ 0&0&1&0&1&0&0&1\\ 1&0&0&1&0&1&0&0\\ 0&1&0&0&1&0&1&0 \end{pmatrix}\right) \left(\begin{pmatrix} 1&0&0&0&1&1&1&1\\ 1&1&0&0&0&1&1&1\\ 1&1&1&0&0&0&1&1\\ 1&1&1&1&0&0&0&1\\ 1&1&1&1&1&0&0&0\\ 0&1&1&1&1&1&0&0\\ 0&0&1&1&1&1&1&0\\ 0&0&0&1&1&1&1&1 \end{pmatrix}\right) \begin{pmatrix} b_0\\ b_1\\ b_2\\ b_3\\ b_4\\ b_5\\ b_6\\ b_7 \end{pmatrix} \oplus$$

$$\left(\begin{pmatrix} 0&0&1&0&0&1&0&1\\ 1&0&0&1&0&0&1&0\\ 0&1&0&0&1&0&0&1\\ 1&0&1&0&0&1&0&0\\ 0&1&0&1&0&0&1&0\\ 0&0&1&0&1&0&0&1\\ 1&0&0&1&0&1&0&0\\ 0&1&0&0&1&0&1&0 \end{pmatrix}\right) \begin{pmatrix} 1\\ 1\\ 0\\ 0\\ 0\\ 1\\ 1\\ 0 \end{pmatrix} \oplus \begin{pmatrix} 1\\ 0\\ 1\\ 0\\ 0\\ 0\\ 0\\ 0 \end{pmatrix} =$$

$$\begin{pmatrix} 1&0&0&0&0&0&0&0\\ 0&1&0&0&0&0&0&0\\ 0&0&1&0&0&0&0&0\\ 0&0&0&1&0&0&0&0\\ 0&0&0&0&1&0&0&0\\ 0&0&0&0&0&1&0&0\\ 0&0&0&0&0&0&1&0\\ 0&0&0&0&0&0&0&1 \end{pmatrix} \begin{pmatrix} b_0\\ b_1\\ b_2\\ b_3\\ b_4\\ b_5\\ b_6\\ b_7 \end{pmatrix} \oplus \begin{pmatrix} 1\\0\\1\\0\\0\\0\\0\\0 \end{pmatrix} \oplus \begin{pmatrix} 1\\0\\1\\0\\0\\0\\0\\0 \end{pmatrix} = \begin{pmatrix} b_0\\ b_1\\ b_2\\ b_3\\ b_4\\ b_5\\ b_6\\ b_7 \end{pmatrix}$$

It is proved from the above equation that YX equals to the identity matrix, and the YX=D, so that

$$\mathbf{YC} \oplus \mathbf{D}$$

equals the null vector.

The S-box is designed to be resistant to known cryptanalytic attacks. Specifically, the Rijndael developers sought a design that has a low correlation between input bits and output bits, and the property that the output cannot be described as a simple mathematical function of the input. In addition, the constant in the initial equation is chosen so that the S-box has no fixed points and no opposite fixed points.

The S-box must be invertible, that is IS-Box[S-box(a)]=a. However, the S-box is not self-inverse, in the sense that it is not true that S-box(a)=IS-box(a). For example, S-box({95})={2A}, but IS-box({95})={AD}.

b)ShiftRow Transformation

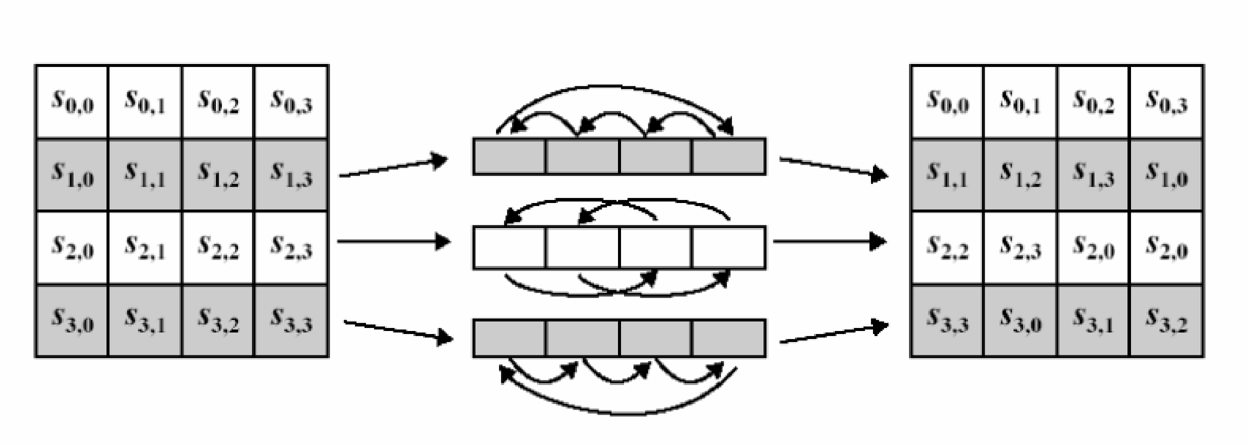

$s_{0,0}$ $s_{0,1}$ $s_{0,2}$ $s_{0,3}$
$s_{1,0}$ $s_{1,1}$ $s_{1,2}$ $s_{1,3}$
$s_{2,0}$ $s_{2,1}$ $s_{2,2}$ $s_{2,3}$
$s_{3,0}$ $s_{3,1}$ $s_{3,2}$ $s_{3,3}$
$s_{0,0}$ $s_{0,1}$ $s_{0,2}$ $s_{0,3}$
$s_{1,1}$ $s_{1,2}$ $s_{1,3}$ $s_{1,0}$
$s_{2,2}$ $s_{2,3}$ $s_{2,0}$ $s_{2,0}$
$s_{3,3}$ $s_{3,0}$ $s_{3,1}$ $s_{3,2}$

Figure 9: ShiftRows 1

In the forward shift row transformation, called ShiftRows, the first row of state is not altered. For the second row, a 1-byte circular left shift is performed. For the second row, a 1-byte circular left shift is performed. For the 3rd and 4th row, a 2-byte and 3-byte shift is performed respectively. Here is an example:

87	F2	4D	97
6E	4C	90	EC
46	E7	4A	C3
A6	8C	D8	95

87	F2	4D	97
EC	6E	4C	90
4A	C3	46	E7
8C	D8	95	A6

Figure 10: ShiftRows 2

The inverse ShiftRow transformation, called InvShiftRows, performs a circular shift in the opposite direction (right shift) for each of the last three rows.

The ShoftRow transformation is more substantial than it may first appear. This is because the State, as well as the cipher input and output is treated as an array of four

4-byte columns. Thus, on encryption, the first 4 bytes of the plaintext are copied to the first column of State, and so on. However the round key is applied to State column by column. Thus, a row shift moves an individual byte from one column to another, which is a linear distance of a multiple of 4 bytes. Moreover, the transformation ensures that the 4 bytes of one column are spread out to four different columns.

c) MixColumns Transformation

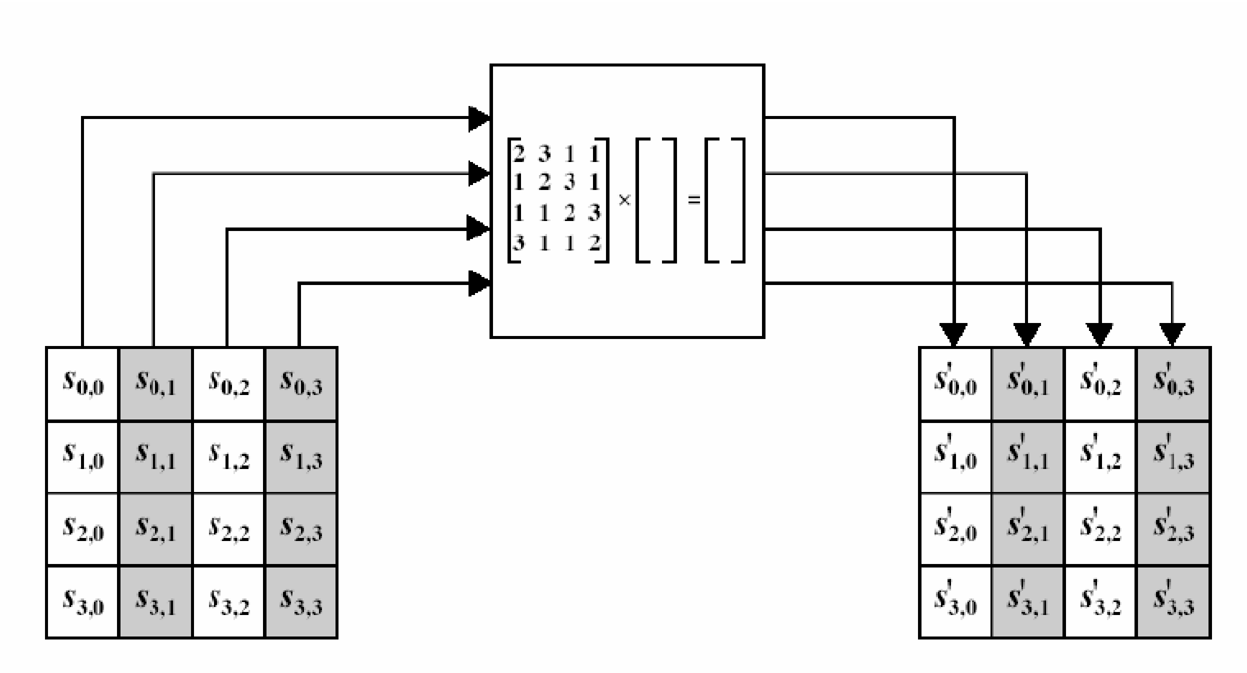


Figure 11: MixColumns 1

The forward mix column transformation, called MixColumns operates on each column individually. Each byte of a column is mapped into a new value that is a function of all four bytes in the column. The transformation can be defined by the following matric multiplication on State.

$$\begin{pmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{pmatrix} \begin{pmatrix} S_{0,0} & S_{0,1} & S_{0,2} & S_{0,3} \\ S_{1,0} & S_{1,1} & S_{1,2} & S_{1,3} \\ S_{2,0} & S_{2,1} & S_{2,2} & S_{2,3} \\ S_{3,0} & S_{3,1} & S_{3,2} & S_{3,3} \end{pmatrix} = \begin{pmatrix} S'_{0,0} & S'_{0,1} & S'_{0,2} & S'_{0,3} \\ S'_{1,0} & S'_{1,1} & S'_{1,2} & S'_{1,3} \\ S'_{2,0} & S'_{2,1} & S'_{2,2} & S'_{2,3} \\ S'_{3,0} & S'_{3,1} & S'_{3,2} & S'_{3,3} \end{pmatrix}$$

(3.3)

Each element in the product matric is the sum of products of elements of one row and one column. In this case, the individual additions and multiplications are performed in GF(2^8). The MixColumns transformation on a single column j ($0 \le j \le 3$) of State can be expressed as:

$$S'_{0,j} = (2 \bullet S_{0,j}) \oplus (3 \bullet S_{1,j}) \oplus S_{2,j} \oplus S_{3,j}$$

$$S'_{1,j} = S_{0,j} \oplus (2 \bullet S_{1,j}) \oplus (3 \bullet S_{2,j}) \oplus S_{3,j}$$

$$S'_{2,j} = S_{0,j} \oplus S_{1,j} \oplus (2 \bullet S_{2,j}) \oplus (3 \bullet S_{3,j})$$

$$S'_{2,j} = (3 \bullet S_{0,j}) \oplus S_{1,j} \oplus S_{2,j} \oplus (2 \bullet S_{3,j})$$

The following is an example of MixColumns:

87	F2	4D	97
6E	4C	90	EC
46	E7	4A	C3
A6	8C	D8	95

→

47	40	A3	4C
37	D4	70	9F
94	E4	3A	42
ED	A5	A6	BC

The first column of the above example will be verified now. In GF (2^8), addition can be implemented by bitwise XOR operation and multiplication by a value (i.e by {02}) can be implemented as a 1-bit left shift followed by a conditional bitwise XOR with (00011011) if the leftmost bit of the original value (prior to the shift) is 1. Thus, to verify the MixColumns transformation on the first column, these equations must be verified:

$$\begin{aligned}
(\{02\} \bullet \{87\}) \oplus (\{03\} \bullet \{6E\}) \oplus \{46\} \oplus \{A6\} &= \{47\} \\
\{87\}) \oplus (\{02\} \bullet \{6E\}) \oplus (\{03\} \bullet \{46\}) \oplus \{A6\} &= \{37\} \\
\{87\}) \oplus \{6E\} \oplus (\{02\} \bullet \{46\}) \oplus (\{02\} \bullet \{A6\}) &= \{94\} \\
(\{03\} \bullet \{87\}) \oplus \{6E\} \oplus \{46\} \oplus (\{02\} \bullet \{A6\}) &= \{ED\}
\end{aligned}$$

For the first equation,

$$\{02\} \bullet \{87\} = (0000\ 1110) \oplus (0001\ 1011) = (0001\ 0101)$$

and

$\{03\} \bullet \{6E\} = \{6E\} \oplus (\{02) \bullet \{6E\ \}) = (0110\ 1110) \oplus (1101\ 1100) = (1011\ 0010)$. Then

$$\begin{array}{lcr}
\{02\} \bullet \{87\} & = & 0001\ 0101 \\
\{03\} \bullet \{6E\} & = & 1011\ 0010 \\
\{46\} & = & 0100\ 0110 \\
\{A6\} & = & \underline{1010\ 0110} \\
 & & 0100\ 0111
\end{array}$$

The other equations can be similarly verified.

The inverse mix column transformation, called InvMixColumns, is defined by the following matrix multiplication:

$$\begin{pmatrix} 0E & 0B & 0D & 09 \\ 09 & 0E & 0B & 0D \\ 0D & 09 & 0E & 0B \\ 0B & 0D & 09 & 0E \end{pmatrix} \begin{pmatrix} S_{0,0} & S_{0,1} & S_{0,2} & S_{0,3} \\ S_{1,0} & S_{1,1} & S_{1,2} & S_{1,3} \\ S_{2,0} & S_{2,1} & S_{2,2} & S_{2,3} \\ S_{3,0} & S_{3,1} & S_{3,2} & S_{3,3} \end{pmatrix} = \begin{pmatrix} S'_{0,0} & S'_{0,1} & S'_{0,2} & S'_{0,3} \\ S'_{1,0} & S'_{1,1} & S'_{1,2} & S'_{1,3} \\ S'_{2,0} & S'_{2,1} & S'_{2,2} & S'_{2,3} \\ S'_{3,0} & S'_{3,1} & S'_{3,2} & S'_{3,3} \end{pmatrix}$$

(3.5)

It is not immediately clear that Equation 3.5 is the inverse of equation (3.3). To show that:

$$\begin{pmatrix} 0E & 0B & 0D & 09 \\ 09 & 0E & 0B & 0D \\ 0D & 09 & 0E & 0B \\ 0B & 0D & 09 & 0E \end{pmatrix} \begin{pmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{pmatrix} \begin{pmatrix} S_{0,0} & S_{0,1} & S_{0,2} & S_{0,3} \\ S_{1,0} & S_{1,1} & S_{1,2} & S_{1,3} \\ S_{2,0} & S_{2,1} & S_{2,2} & S_{2,3} \\ S_{3,0} & S_{3,1} & S_{3,2} & S_{3,3} \end{pmatrix} = \begin{pmatrix} S_{0,0} & S_{0,1} & S_{0,2} & S_{0,3} \\ S_{1,0} & S_{1,1} & S_{1,2} & S_{1,3} \\ S_{2,0} & S_{2,1} & S_{2,2} & S_{2,3} \\ S_{3,0} & S_{3,1} & S_{3,2} & S_{3,3} \end{pmatrix}$$

Which is equivalent to showing that:

$$\begin{pmatrix} 0E & 0B & 0D & 09 \\ 09 & 0E & 0B & 0D \\ 0D & 09 & 0E & 0B \\ 0B & 0D & 09 & 0E \end{pmatrix} \begin{pmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

(3.6)

That is, the inverse transformation matrix times the forward transformation matrix equals the identity matrix. To verify the first column of equation (3.6), the following equations must be verified:

$$(\{0E\} \bullet \{02\}) \oplus (\{0B\} \bullet \{0D\}) \oplus \{09\} \oplus \{03\} = \{01\}$$

$$(\{09\} \bullet \{02\}) \oplus (\{0E\} \bullet \{0B\}) \oplus \{0D\} \oplus \{03\} = \{00\}$$

$$(\{0D\} \bullet \{02\}) \oplus (\{09\} \bullet \{0E\}) \oplus \{0B\} \oplus \{03\} = \{00\}$$

$$(\{0B\} \bullet \{02\}) \oplus (\{0D\} \bullet \{09\}) \oplus \{0E\} \oplus \{03\} = \{00\}$$

For the first equation, $\{0E\} \bullet \{02\} = 0001\ 1100$; and $\{09\} \bullet \{03\} = \{09\} \oplus (\{09\} \bullet \{02\}) = 00001001 \oplus 00010010 = 00011011$. Then

$$\begin{array}{lcl} \{0E\} \bullet \{02\} & = & 0001\ 1100 \\ \{0B\} & = & 0000\ 1011 \\ \{0D\} & = & 0000\ 1101 \\ \{09\} \bullet \{03\} & = & \underline{0001\ 1011} \\ & & 0000\ 0001 \end{array}$$

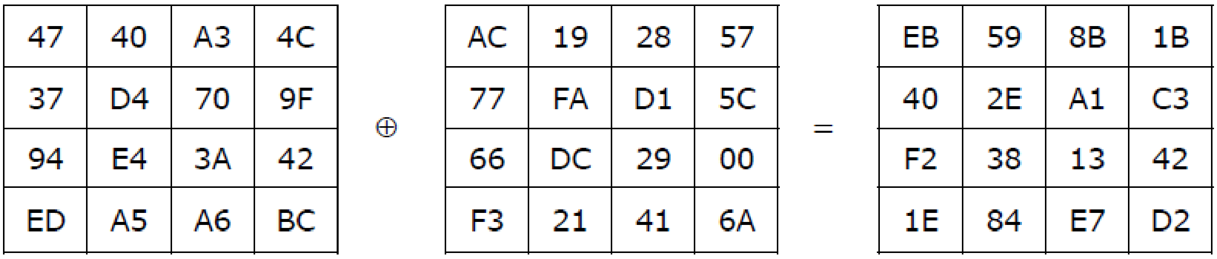
The other equations can be similarly verified.

The coefficients of the matrix in Equation (3.3) are based on a linear code with maximal distance between code words, which ensures a good mixing among the bytes of each column. The mix column transformation combined with the shift row transformation ensures that after a few rounds, all output bits depend on all input bits.

In addition, the choice of coefficients in MixColumns, which are all {01}, {02}, or {03}, was influenced by implementation considerations. As we discussed, multiplication by these coefficients involves at most a shift and a XOR. The coefficients in InvMixColumns are more formidable to implement. However, encryption was deemed more important that decryption. This is due to the fact that the CFB and OFB modes only use encryption, and also as with any block cipher, AES can be used to construct a message authentication code, and for this only encryption is used.

d) AddRoundKey Transformation

In the forward add round key transformation, called AddRoundKey, the 128 bits of State are bitwise XORed with the 128 bits of the round key. The operation is viewed as a columnwise operation between the 4 bytes of a State column and one word of the round key. It can also be viewed as a byte-level operation. The following is an example of AddRoundKey:



$$\begin{array}{|c|c|c|c|}\hline 47 & 40 & A3 & 4C \\ \hline 37 & D4 & 70 & 9F \\ \hline 94 & E4 & 3A & 42 \\ \hline ED & A5 & A6 & BC \\ \hline \end{array} \oplus \begin{array}{|c|c|c|c|}\hline AC & 19 & 28 & 57 \\ \hline 77 & FA & D1 & 5C \\ \hline 66 & DC & 29 & 00 \\ \hline F3 & 21 & 41 & 6A \\ \hline \end{array} = \begin{array}{|c|c|c|c|}\hline EB & 59 & 8B & 1B \\ \hline 40 & 2E & A1 & C3 \\ \hline F2 & 38 & 13 & 42 \\ \hline 1E & 84 & E7 & D2 \\ \hline \end{array}$$

Figure 12: AddRoundKey

The first matrix is State, and the second matrix is the round key.

The inverse add round key transformation is identical to the forward add round key transformation because XOR operation is its own inverse.

The add round key transformation is as simple as possible and affects every bit of State. The complexity of the round key expansion, plus the complexity of the other stages of AES ensure security.

2.4 Concluding Remarks

The AES algorithm was developed by two Belgian cryptographers, Joan Daemen and Vincent Rijmen, based on their previous design called “Square” **[34]**. Unlike its predecessor (DES), it is not a Feistel network, but rather a substitution-permutation network.

It is definitely a state of the art algorithm for encryption. Combining it with the appropriate block cipher mode will result in ciphers that can cover most (if not all) needs, whether it is elevated security we are focusing on, or increased performance. Its reliability has been proven over the years and it is not by chance that it has been used for decades for classified document encryption by the U.S government. It is also worth noting that all discovered security holes on this algorithm up to this day didn't prove anything but the fact that someone would be able to crack it in several billions of years.

3. PARALLEL PROGRAMMING

3.1 Introduction

Parallel programming is the idea of the simultaneous use of multiple compute resources to solve a computational problem. The resources may coexist inside a CPU (or other computer components), a whole computer system or even multiple computers.

Traditionally, software has been written for serial computation, rather than parallel. In order to understand parallel programming, it is necessary to have a complete picture of how a serial program works. These are the main characteristics of a serial program:

- A problem is broken into a discrete series of instructions
- Instructions are executed sequentially one after another
- Executed on a single processor
- Only one instruction may execute at any moment in time

A simple example of serial programming would be a C language program that reads some data from a text file, makes some calculations on that data and prints a result in the user’s screen. These steps are performed in a specific order or else there is going to be a problem with the output, a logical error.

Now imagine another program where we need to do two separate things: one is to read some data from a text file and print it, and the other is to make some separate calculations. These two tasks of the program are totally independent. A serial program would do either one of them first, and then proceed to finish the other. That is not necessary though, as we can make use of parallel programming to start both those tasks simultaneously and thus save time on the execution of the program.

If these examples seem too simple, there are more complicated reasons that one should consider implementing parallel programming on his software. For example, in certain situations, the program is waiting for some kind of input from the client. The use of serial programming on this example would automatically mean that the time until the person gives the input will be wasted (no other instructions can be executed).

However, there are other scenarios where we can use parallelism to further optimize the software we are working on. Except from finding the separate tasks of the program and assigning them to certain available resources, there is a very common case where we tend to divide a certain task into parts which will be treated by a different resource. A good example would be image processing. The image may contain many millions of pixels, resulting in really slow serial software implementations. It is relatively easy to divide the image in sub-images of a certain size (depending on our system resources), let each task run separately and combine the sub-images to form the final result. And the performance gains can be massive.

Another case where parallel programming is really meaningful is Event-Driven software. Event-driven programming is a programming paradigm in which the flow of the program is determined by events such as user actions (mouse clicks, key presses), sensor outputs or messages from other programs. In most cases where other code should be running, serial code would block the whole program (until an event gets triggered) and render it useless.

3.2 CPU parallel programming

When focusing on parallel programming on a single computer system, there are several computer components to consider. At first, the component often associated with command execution is the CPU, so one would probably think that parallel programming is making use of the CPU in such a way as to be able to execute many commands simultaneously, and would not be entirely wrong. Let’s take a look at how parallel programming in the CPU started and its evolution through the years.

The evolution of the CPU through the years was mainly focused on shrinking the area of the integrated circuit (IC), which drove down the cost per device on the IC while increasing functionality. At some point CPUs were created with stock frequencies of about 4Ghz which means they would get really hot on full load. Problems such as the ones we mentioned earlier, along with this one initiated the introduction of multi-core CPUs, which have more than one processing unit (core) and as a result have the capability of processing multiple instructions at the same time. Simultaneous execution of different independent tasks became possible. Around 2005-2006 most high-end commercially available CPUs were Dual Core. Nowadays, most mid-range computers use Quad Core CPUs, while there are also higher end Octa-Core CPUs (8 cores) and even 10 core CPUs (Intel Xeon E7-8870), especially designed for server use. Multi-core CPUs can be interpreted as the simplest form of parallel computing.

The basic element of parallel programming is the thread. Threads are one of several technologies that make it possible to execute multiple code paths concurrently inside a single application. A common case is that a CPU core can run one thread at a specific time. In other words, a quad core CPU is able to run four threads simultaneously. In more advanced cases there are CPUs with cores that support more than 1 thread at a time. Such an example is Intel's Hyper-threading technology, which allows each core to maintain two threads at each moment.

Parallel Programming in a CPU is based on a multi-threading approach of software. Back on the introduction's example, a software developer could make better use of a multi-core CPU if he created one thread that would be responsible for the user's input (I/O operation), and a second one that would do the independent calculations. These two threads would run in parallel on any CPU with at least 2 cores, and the problem would finish up significantly faster.

Historically, hardware vendors have implemented their own proprietary versions of threads. These implementations differed substantially from each other making it difficult for programmers to develop portable threaded applications. In order to take full advantage of the capabilities provided by threads, a standardized programming interface was required.

Since C++ 2011 release became official, there is native thread support for C++, while it was previously impossible to make use of threads in C++ without an external thread API. In practice, since C++ 2011 release, depending on the platform that the code is being compiled on, either pthreads (in case of Linux systems) or Windows threads (in case of Windows systems) can be used. This provided a standardized way to include multithreading in the C++ language and enabled programmers to include thead support for their software easily.

There are several thread implementations used, though the basic two are the following:

3.2.1 POSIX Threads

More commonly known as pthreads, it is a low-level API for working with threads. POSIX threads has been specified as an interface for UNIX systems by the IEEE POSIX 1003.1c standard in 1995, and has continued to evolve and undergo revisions and improvements.

Pthreads defines a set of C programming language types, functions and constants. It is implemented with a pthread.h header and a thread library. The procedures are divided into four basic groups:

- Thread management – creating, joining threads, etc
- Mutexes (objects used for thread synchronization)
- Condition Variables
- Synchronization between threads using read/write locks and barriers

What is really important about pthreads is that it provides fine-grained control over thread management (create, join, etc), shared memory and synchronization (mutexes). For that reason, it requires proper programming and manual setting of all operations by the programmer.

Last but not least, there are implementations of the API on many Unix-like POSIX-compatible operating systems, such as FreeBSD, NetBSD, OpenBSD, Linux, Mac OSX and Solaris. DR-DOS and Microsoft Windows implementations also exist (the SFU/SUA subsystem provides a native implementation of a number of POSIX APIs, and third-party packages such as *pthreads-w32*, implements pthreads on top of existing Windows API).

3.2.2 OpenMP

OpenMP stands for Open Multi-Processing and is an API that supports multi-platform shared memory multiprocessing programming in C, C++, and Fortran, on most processor architectures and operating systems, including Solaris, AIX, HP-UX, Linux, Mac OS X, and Windows platforms (the meaning of shared memory will be explained later). It consists of a set of compiler directives, library routines, and environment variables that influence run-time behavior.

Often paired with MPI implementations, OpenMP works at a much higher level than POSIX threads. Notably, MPI (Message-Passing Interface) is a standardized and portable message-passing system used widely in parallel programming.

OpenMP has the advantages of being cross platform, and simpler for some operations. It handles threading in a different manner, in that it gives you higher level threading options and is relatively easy to embed in existing code, unlike POSIX threads implementations.

We have included OpenMP in our software development as a means of comparison with both the serial and the CUDA code, performance-wise. This comparison has produced some interesting results, but more on that later.

As we mentioned earlier, in cases where much processing power is required, complex computing systems are used, containing up to hundreds of multi-core processors. These systems are often file/web servers that need to serve millions of requests in minimal time. Many corporate enterprises or high activity service providers may even contain huge areas filled with these systems, taking advantage of parallel programming as much as possible.

3.3 GPU parallel programming

We have already discussed threaded applications in the CPU, but there is more to parallel programming than CPU threading cases. General-purpose computing on graphics processing units (GPGPU) is the use of a graphics processing unit (GPU), which typically handles computation only for computer graphics, to perform computation in applications traditionally handled by the central processing unit (CPU). The use of multiple graphics cards in one computer, or large numbers of graphics chips, further parallelizes the already parallel nature of graphics processing. In addition, even a single GPU-CPU framework provides advantages that multiple CPUs on their own do not offer due to specialization in each chip. Two are the dominant implementations of general-purpose GPU programming:

3.3.1 OpenCL

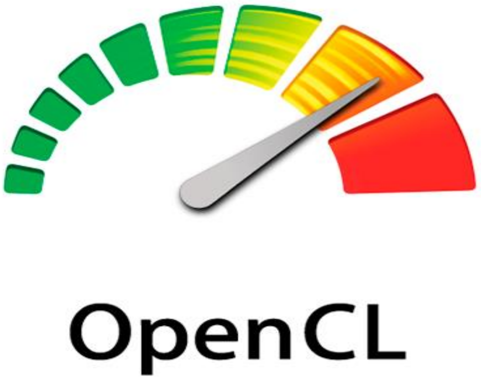


Figure 13: OpenCL

Open Computing Language (OpenCL) is a framework for writing programs that execute across heterogeneous platforms consisting of central processing units (CPUs), graphics processing units (GPUs), digital signal processors (DSPs), field-programmable gate arrays (FPGAs) and other processors. OpenCL specifies a language for programming these devices and application programming interfaces (APIs) to control the platform and execute programs on the compute devices. OpenCL provides parallel computing using task-based and data-based parallelism. OpenCL is an open standard maintained by the non-profit technology consortium Khronos Group.

OpenCL defines a C-like language for writing programs, called kernels, that execute on the compute devices. defines an application programming interface (API) that allows programs running on the host to launch kernels on the compute devices and manage device memory, which is (at least conceptually) separate from host memory. Programs in the OpenCL language are intended to be compiled at run-time, so that OpenCL-using applications are portable between implementations for various host devices.**[22]** The OpenCL standard defines host APIs for C and C++; third-party APIs exist for other programming languages such as Python, Java and .NET.

3.3.2 CUDA

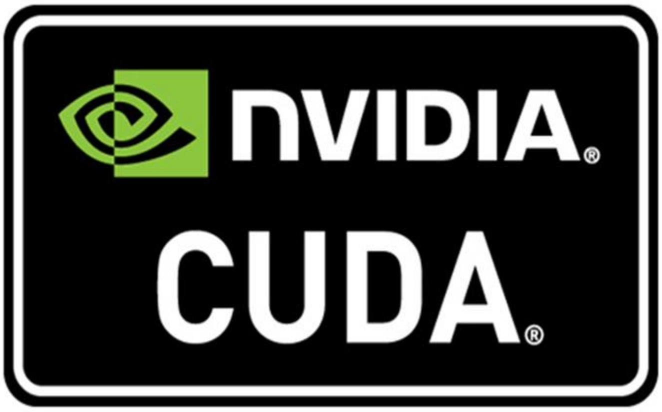


Figure 14: CUDA logo

CUDA, which stands for Compute Unified Device Architecture,**[23]** is a parallel computing platform and application programming interface (API) model created by NVIDIA. It allows software developers to use a CUDA-enabled graphics processing unit (GPU) for general purpose processing. The CUDA platform is a software layer that gives direct access to the GPU's virtual instruction set and parallel computational elements.

CUDA is designed to be able to work with programming languages such as C, C++ and Fortran. That enables programmers to use GPU resources without graphics knowledge, to achieve their tasks. This is a huge advantage mainly because previous API solutions like Direct3D and OpenGL required special skills and experience in graphics programming for someone to embed into his code.

CUDA was initially released in 2007, though it managed to become one of the most (if not the most) dominant GPGPU approach. It has some advantages over other approaches, the main of which are the following:

- Scattered reads – code can read from arbitrary addresses in memory.
- Unified virtual memory.
- Unified memory.

- Shared memory – CUDA exposes a fast shared memory region that can be shared amongst threads. This can be used as a user-managed cache, enabling higher bandwidth than is possible using texture lookups.
- Faster downloads and readbacks to and from the GPU.
- Full support for integer and bitwise operations, including integer texture lookups.

CUDA uses threads that run on the GPU area. Better yet, it uses sets of threads called blocks. A function that is called to run on the GPU is called a kernel. For example, the host code (CPU code e.g C) can invoke a kernel (that will run on the GPU) and that kernel could run with 3 blocks of 1024 threads each. Blocks are also grouped into grids, in such a way that a grid is a 2D array of blocks.

In addition, a warp size is the number of threads running concurrently on a multi-processor (GPU/CPU). In actuality, the threads are running both in parallel and pipelined. The total threads that are being executed are divided into warps, which run simultaneously. For example, if warp size is 32 and 58 threads need to be executed, the first warp will contain threads 0…31, and the second one will contain the remaining 32…57. **[33]**

Graphics cards have way more cores than CPUs in general, so it is normal to be able to run several thousands of threads in parallel. This alone gives hope for great potential with GPU parallel programming.

Finally, CUDA does not come without its limitations, such as the fact that only NVIDIA cards may support CUDA, the fact that CUDA does not support the whole C standard (because it runs host code through a C++ compiler), etc. Weighing the pros and cons makes CUDA a very good choice though, and throughout this thesis we'll be showing the acceleration it can provide to a serial piece of code.

3.4 Classification of parallel computers

As we move on to a larger scale and come to the point where we examine computer systems with multiple CPUs or GPUs, we move to a new informatics area referred to as parallel computing. There are many factors to examine on this area, and thus there are several ways to categorize. The most important of them are the following:

- **Classification based on the instruction and data streams**

The term ‘stream’ refers to a sequence or flow of either instructions or data operated on by the computer. In the complete cycle of instruction execution, a flow of instructions from main memory to the CPU is established. This flow of instructions is called instruction stream. Similarly, there is a flow of operands between processor and memory bi-directionally. This flow of operands is called data stream.

Flynn’s Classification

Flynn’s classification **[19]** is based on multiplicity of instruction streams and data streams observed by the CPU during program execution. Let Is and Ds are minimum number of streams flowing at any point in the execution, then the computer organisation can be categorized as follows:

a)Single Instruction and Single Data stream (SISD)

In this organisation, sequential execution of instructions is performed by one CPU containing a single processing element (PE), i.e., ALU under one control unit. Therefore, SISD machines are conventional serial computers that process only one stream of instructions and one stream of data.

b)Single Instruction and Multiple Data stream (SIMD)

In this organisation, multiple processing elements work under the control of a single control unit. It has one instruction and multiple data stream. All the processing elements of this organization receive the same instruction broadcast from the CU. Main memory can also be divided into modules for generating multiple data streams acting as a distributed memory. Therefore, all the processing elements simultaneously execute the same instruction and are said to be 'lock-stepped' together. Each processor takes the data from its own memory and hence it has on distinct data streams. Every processor must be allowed to complete its instruction before the next instruction is taken for execution. Thus, the execution of instructions is synchronous.

c)Multiple Instruction and Single Data stream (MISD)

In this organization, multiple processing elements are organised under the control of multiple control units. Each control unit is handling one instruction stream and processed through its corresponding processing element. But each processing element is processing only a single data stream at a time. Therefore, for handling multiple instruction streams and single data stream, multiple control units and multiple processing elements are organised in this classification. All processing elements are interacting with the common shared memory for the organisation of single data stream. The only known example of a computer capable of MISD operation is the C.mmp built by Carnegie-Mellon University.

d)Multiple Instruction and Multiple Data stream (MIMD)

In this organization, multiple processing elements and multiple control units are organized as in MISD. But the difference is that now in this organization multiple instruction streams operate on multiple data streams. Therefore, for handling multiple instruction streams, multiple control units and multiple processing elements are organized such that multiple processing elements are handling multiple data streams from the Main memory. The processors work on their own data with their own instructions. Tasks executed by different processors can start or finish at different times. They are not lock-stepped, as in SIMD computers, but run asynchronously. This classification actually recognizes the parallel computer. That means in the real sense MIMD organisation is said to be a Parallel computer. All multiprocessor systems fall under this classification.

- **Classification based on the structure of computers**

Flynn’s classification discusses the behavioral concept and does not take into consideration the computer’s structure. For reference, there are:

i)Shared Memory System / Tightly Coupled Systems:

A shared memory computer has multiple cores that have access to the same physical memory. The cores may be part of multicore processor chips, or they may be on discrete chips. We have several models to analyze, but the basic ones are:

1)Uniform Memory Access Model (UMA): The main memory is uniformly shared by all processors in multiprocessor systems and each processor has equal access time to shared memory. This model is used for time-sharing applications in a multi user environment.

2)Non-Uniform Memory Access Model (NUMA): In shared memory multiprocessor systems, local memories can be connected with every processor. The collection of all local memories form the global memory being shared. In this way, global memory is distributed to all the processors. In this case, the access to a local memory is uniform for its corresponding processor as it is attached to the local memory. But if one reference is to the local memory of some other remote processor, then 37 Elements of Parallel Computing and Architecture the access is not uniform. It depends on the location of the memory. Thus, all memory words are not accessed uniformly.

3)Distributed Memory Systems: These systems do not share the global memory because shared memory concept gives rise to the problem of memory conflicts, which in turn slows down the execution of instructions. Therefore, each processor is having a large local memory, not shared by any other processor

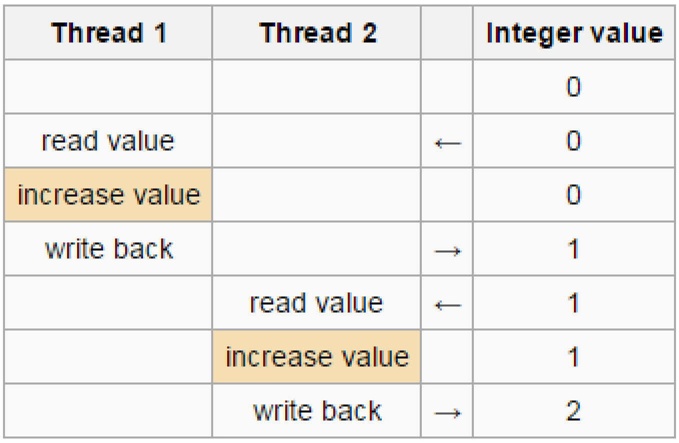
- **Classification based on the grain size**

This classification is based on recognizing the parallelism in a program to be executed on a multiprocessor system. The idea is to identify the sub-tasks or instructions in a program that can be executed in parallel. But it is not sufficient to check for the parallelism between statements or processes in a program. The decision of parallelism also depends on the number and types of processors available, memory organization and dependency of data, control and resources.

3.5 Parallel Programming Issues

Though parallel programming seems to offer much potential to software development, it surely has its hidden risks and difficulties. During a problem analysis, one should understand thoroughly the needs that need to be met, and afterwards he should locate the code that can be parallelized. Then there is the issue of dividing the available resources (e.g cores/threads). This can be as simple as dividing a big array into smaller pieces and assigning them to threads, but sometimes the problem is way more complicated.

Few are the times that the data is totally independent and this causes a common problem with the parallel code. A simple example is the case in which two threads try to increase the same variable by one. This operation is analyzed in three separate operations in assembly code: reading of the current value, increasing the value and writing it back. The expected output is shown below:



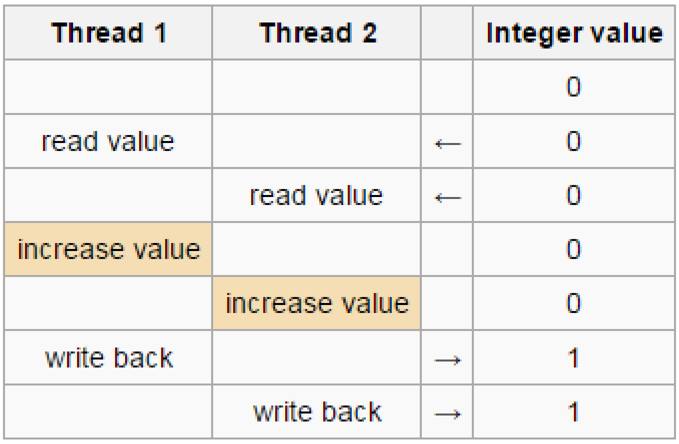
Thread 1	Thread 2		Integer value
			0
read value		←	0
increase value			0
write back		→	1
	read value	←	1
	increase value		1
	write back	→	2

Figure 15: Race Condition 1

However in parallel code there is no way to determine the exact way the threads will run, unless the programmer explicitly sets them to run in a specific way. That is, there is now way to predict the order that the six commands (three per thread) will be executed.

It is not impossible that both threads read the initial value of the variable, which will eventually increase the variable by one, not two.

As a result, the output of the program can be as wrong as the following:



Thread 1	Thread 2		Integer value
			0
read value		←	0
	read value	←	0
increase value			0
	increase value		0
write back		→	1
	write back	→	1

Figure 16: Race Condition 2

The aforementioned example is a simple case of what we call a Race Condition problem, and is generally one of the main problems in parallel programming. It is by no means unsolvable, but it requires attention proper set up on developer level. In complex problems, Race Condition can be really hard to face, not to mention that debugging parallel programs is significantly more difficult than serial ones.

Race condition problems can be addressed with thread synchronization. Depending on the implementation, there are several ways a program can achieve synchronization. For example, in java there is the “Synchronized” declaration, which indicates that when a thread invokes the method/function of an object, it will acquire a lock which will not allow another thread to have access to that object’s method. In C and Linux distributions, synchronization is handled by two mechanisms, semaphores and spinlocks. These are handled manually, which means it is the developer’s duty to set them up and handle them correctly throughout the whole execution of the program. They are almost the same as common variables and are functioning as a lock-unlock mechanism.

POSIX threads uses mutex locks (similar to semaphores), condition variables, barriers, spinlocks and read-write locks. CUDA, on the other hand uses native functions such as __syncthreads() , which pauses execution until all threads from the current block reach that point in the code. Other methods include cudaDeviceSynchronize() and __threadfence().

In a completely different perspective, parallel programming is not for every part of every algorithm. One should not try to parallelize everything on his code, as the result may often result in worse performance. In OpenMp implementations, overuse of parallelism could result – in the worst case – in minor performance decrease. That's not the case with CUDA, though. CUDA thread performance has been proven stellar especially in highly parallelizable programs. That is why a common mistake is jumping to the conclusion that the GPU is faster than the CPU for every calculation, which is plain wrong. In other words, CPUs and GPUs have significantly different architectures that make them better suited to different tasks.

A CPU core may be many times faster than a GPU core. It almost always runs at a much higher frequency, it uses technologies such as 3-Level cache memories (much faster than RAM and GPU memory), branch prediction, prefetch, micro op re-ordering and are out of order. CUDA cores are by no means that powerful. It is in the number of parallel threads that can be run where the power of the GPU truly shines. At the moment that this thesis was written, the highest-end commercially available CPU (Intel Xeon E7-8870) can run 20 threads in parallel, whereas the highest-end NVIDIA CUDA GPU (NVIDIA GTX TITAN Z) can handle as many as 5760. Last but not least, there is also the memory overhead that should not be ignored. CUDA kernel invocations require memory allocation and copy operations to and from the device memory (GPU memory). These operations are costly in terms of time and should not be overused for no reason, in order to avoid odd overheads. The use of some of them is a necessary additional time cost, though in order for our program to function properly.

Therefore, we can assume that in specific (non parallelizable) programs, a solid serial (CPU) code can be as efficient and effective as it gets. It is very important to be able to determine if a part or several parts of a program would actually benefit from a parallel implementation. That means that once again the developer has to come down to a conclusion about whether it makes sense to implement it in his code or not. Developing such skills takes time and experience.

3.6 Conclusion

Parallel Programming in general is considered to be the high end of computing, and there is a reason behind that statement. It has been used to model difficult problems in many areas of science and engineering, such as nuclear/particle physics, biotechnology, genetics, molecular sciences, Electrical Engineering, Circuit Design, Defense, Weapons, etc. Implementing it may sometimes be conceived mistakenly as an add-on to software or an “unnecessary” feature to a – otherwise – perfectly functional program, but the truth is, all optimization aside, there are many cases in which even the simplest/shortest program cannot function properly without its use. Sure, parallel programming introduces new difficulties in programming, but in the end it is well worth the time and effort.

To sum up, despite it being a relatively new concept, parallel programming is the new trend everywhere nowadays, and there is no denying that much of the existent software is rewritten to take advantage of its capabilities. Experts worldwide agree on its huge potential both through theoretical calculations and through real time measurements. Even given its few years in the field, parallel programming has vastly increased the possible calculations that can be achieved in a certain amount of time, since its first use. Of course that is both the result of hardware and software evolution. We should be optimistic that as the years go by, parallel code development will become easier, more widespread and more effective.

The following figure shows the evolution of computing performance throughout the last twenty years:

- ***The race is already on for Exascale Computing!***
  - Exaflop = 10^{18} calculations per second

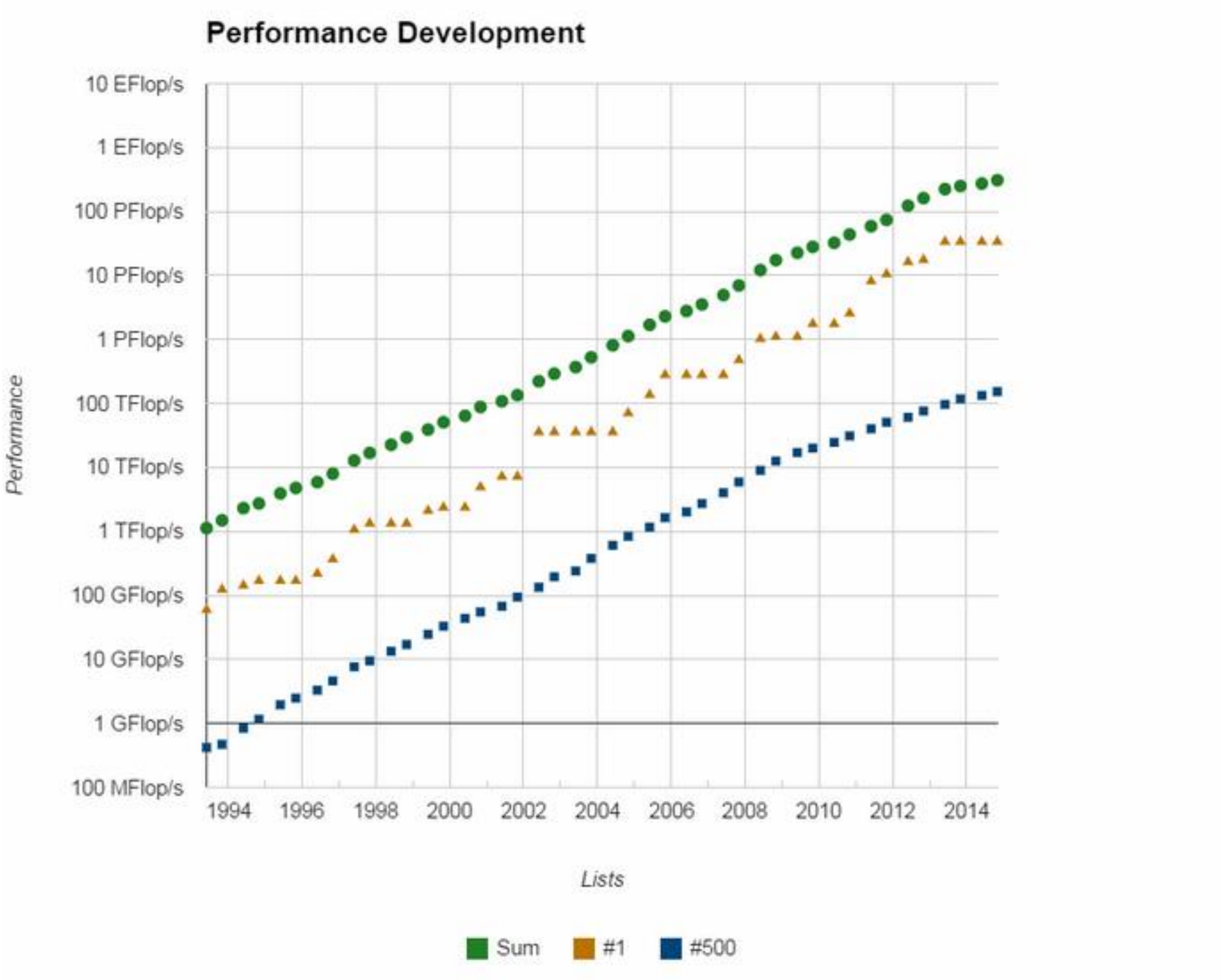


Figure 17: Evolution of computing performance [3]

4. IMPLEMENTING CUDA PROGRAMMING ON AES CODE

The main task we had to face during our thesis this year is the application of CUDA parallel programming techniques on existing C AES code **[4]**. The base C code we used was of course serial and quite a few changes had to be made in order for everything to be able to run normally. But first we should try to explain the reasons parallel programming could be embedded on our test code, as well as the main idea behind our implementation.

We have already mentioned in a previous chapter the several block cipher modes that can be combined with the AES algorithm, or any other cipher algorithm. Depending on the block cipher mode, there can be some parallelism on the algorithm, much parallelism or even no parallelism.

Some block cipher modes require that the encryption of a block of data uses data from its previous encrypted block (or blocks) in order for the encryption to be achieved. That means that the algorithm definitely has to encrypt each block in a serial pattern. As we have already mentioned earlier in this thesis, one of the most representative examples of this case is the CBC block cipher mode, which stands for Chain Block Chaining. The way in which the CBC mode operates can be explained briefly through the following diagram.

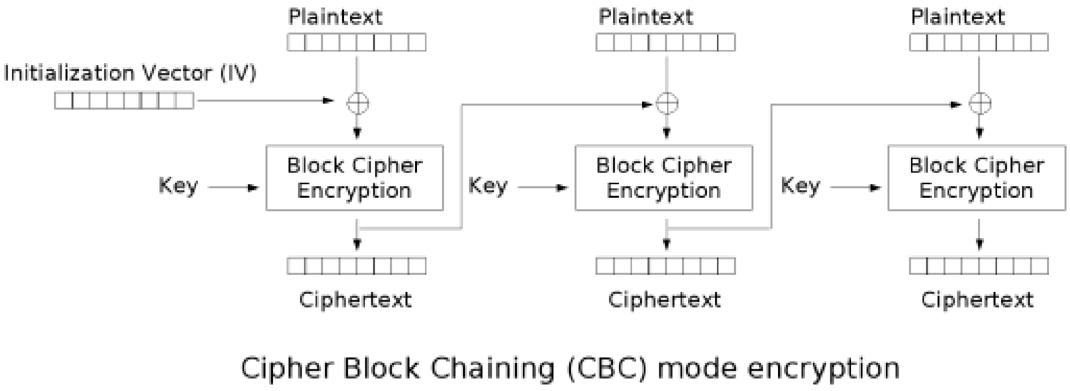


Figure 18: CBC

The first block of data is XOR-ed against the IV (initialization vector) to produce the first ciphertext block, and then the n-th block of data is XOR-ed against last encrypted block before you encrypt this block. It is obvious that no significant data parallelism can be applied on CBC, at least during the encryption process.

However this is not the case with every block cipher mode. There are cipher block modes that allow for data parallelism to be achieved, at least to some extent. The GCM (Galois/Counter Mode), for example can be parallelized (e.g using OpenMp or CUDA), at least for the part of encryption and decryption (some parts of the algorithm may not be easily parallelizable). Parallelism on the authentication section of any encryption algorithm is not considered important, and in some cases it is not even possible.

Most importantly for the context of the current thesis, the ECB mode (which stands for Electronic Code Block) can be implemented in parallel code relatively easy, due to the fact that it is basically a raw cipher. That means that each data block of input is encrypted separately and it produced some cipher block output. In the end all those cipher blocks are combined together to form the ciphertext. In the same way, during the decryption process, each cipher data block is decrypted to a plaintext block, and all those plaintext blocks will form the final plaintext.

For the aforementioned reasons, comparing serial and parallel performance on the ECB mode should produce some interesting results. ECB may be the simplest mode of all, but it is arguably the case that will most properly indicate the significant difference in performance, should the software take advantage of all available sources. And our test results agree with this statement.

For the encryption process, the main idea behind our CUDA implementation consists of the following steps:

- The initial plaintext is saved in a buffer
- A C function is called in order to initialize the CUDA parameters, determine the required blocks and threads that will be used and copy the input buffer to the device (GPU) memory. The total amount of threads that will be used is equal to the number of data blocks the input buffer contains. For example a 150bit plaintext consists of 9,3~10 blocks. In this perspective, each thread will be responsible for the encryption of its dedicated data block e.g thread 7 will encrypt the 7th block of the input file, etc. It is quite obvious that in our

case, there is absolute independence on the data and there is no need for synchronization on this point, nor a shared memory or inter-thread communication techniques.

- A CUDA kernel is called for the specified blocks and threads. Each thread on this invocation calls the block encryption function on the device (GPU) for its dedicated block, and writes back the resulting block (cipher block) on the buffer, after computing its correspondent offset.
- After the main encryption process is complete, the kernel function finishes and the C function that was mentioned on the 2nd step of the process copies the buffer (that now contains the whole ciphertext) back to host memory (RAM), that is in a C buffer.

Naturally, the decryption process uses the same logic to produce the final plain text that is identical to the initial plaintext that we received from input. As far as input is concerned, there are a few things we changed in order to increase the functionality of the program.

In the original C serial code, there were two types of input methods. The first one (test_encrypt) used the command line so the user had to manually type in a text that would be encrypted and then decrypted back to the plaintext. In our implementation, we changed the input method to file input, so that the input buffer would be filled via a .txt file. In our humble opinion, this method is way better for testing purposes because the user can fill in the input.txt file anyway he likes, instead of typing manually in the command line. Of course, this method introduces some I/O operation delays to the total run time, but these are not taken under consideration on our timings, because they are not related in any way to the cipher. So the displayed timings do not include these delays.

The second input method is random generation of a 1MB buffer that is later used as an input (test_performance). This method aims at measuring pure performance when data sets reach way bigger numbers, as user sets the data size for the cipher to be applied, as a command line parameter. For example, if a user sets 600mb as input data, the 1MB input buffer is randomly initialized, and the encryption-decryption circle is applied to it in a 600 –times loop. Then, some results are being displayed on the user, including the total run time. This method suited our needs perfectly, so we didn’t change it in any way, besides include more detailed timing results, such as total time, encryption-decryption time, memory operations’ time (cudaMemcpy both host-to-device and device-to-host, cudaMalloc, cudaMemset, cudaFree), etc. The statistical diagrams we include later on that include our test results are almost exclusively based on this method, as it was made possible to test the parallelized code against different sets of data.

Notably, the OpenMP implementation that we developed used the same serial code we used as a base on the CUDA code, with the addition of some “#pragma omp parallel/for” tags, which divides a piece of code/loop into separate tasks and assigns them to CPU threads on a high level. Though taking advantage of all available CPU threads also resulted in a definitely not insignificant performance increase with minimal effort software-wise, this only served as a comparative method to the CUDA code, which dominated the performance charts.

On the next chapter, we will present a detailed set of our results, comparing serial, OpenMP and CUDA timings (total, memory operations, etc), with a set of different parameters, including different key sizes and different data sets.

5. SOFTWARE IMPLEMENTATION AND RESULTS

5.1 Hardware Information

First we should take a look at the hardware we used for the completion of the current thesis. The full hardware list is shown below and it belongs to one of our university's computers. This is the hardware on which we made all our tests and drew our results. The hardware on the machines we used for developing the code is quite different but that is irrelevant to our results.

We transferred our files using WinSCP and connected remotely via SSH protocol to run remote compilation and execution commands. The computer was equipped with a CUDA-enabled graphics card (Tesla C2070) with 448 CUDA cores and compute capability 2.0 and the CPU has 4 cores supporting one thread each, so we had no problem testing both CPU threads on OpenMP and CUDA threads.

Notably, the computer runs Ubuntu, a Linux distribution, so our communication was via a command line environment used for remote connections, which is called Putty.

Processor:	AMD Phenom^{TM} II X4 965 @ 3.40 GHz [1 Processor, 4 Cores, 4 Threads]
L1 Instruction Cache:	64.0 KB x 4 (2-way set associative)
L1 Data Cache:	64.0 KB x 4 (2-way set associative)
L2 Cache:	512 KB x 4 (16-way set associative)
L3 Cache:	6.00 MB (48-way set associative)
Memory:	8 GB [4x2GB DDR3 1800 MHz]
Hard Drive:	Seagate Barracuda 1TB 3.5"
GPU:	NVIDIA Corporation GF100GL [Tesla C2050 / C2070] (rev a3), CUDA cores: 448 core clock: 1.15Ghz, Memory clock: 1.5Ghz, Compute capability: 2.0
Operating System:	Ubuntu 14.04.2 LTS 3.13.0-48-generic x86_64
GCC:	v4.8.2
NVidia Driver:	v340.29 , CUDA version 6.5.12

5.2 Implementation Analysis

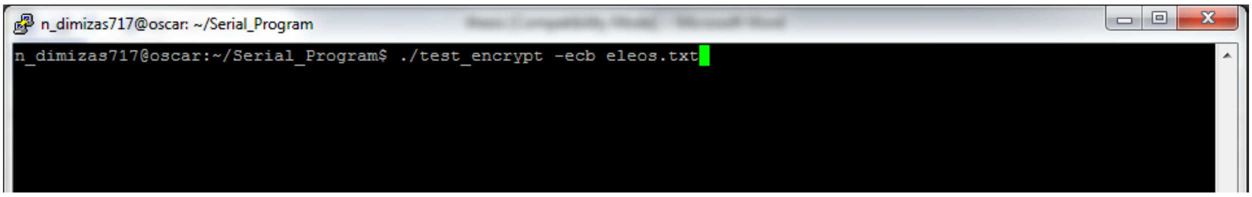
While the algorithm for the AES ECB was ready and written in C code as a base for our CUDA and OpenMP implementations, there were surely some changes to be made and some factors to be considered in order to be able to perform our assigned task. In the current chapter, we will present the main idea on which we relied to make the AES ECB parallel, using the OpenMP and the CUDA library.

What we have to explain initially is the way the algorithm works. The input file (either given by hand via the terminal, generated in the program or read from a text file) is divided in blocks of 16bytes each (128-bit, as the AES standard indicates). If the input size is not divisible by 16, the last block is partially filled with the input data, and the rest of the block is filled with some specific data that is added for that purpose only, and it is called pad.

The CBC block mode that we use on our implementation does not provide any dependencies between blocks. Therefore each one of these blocks is encrypted totally independently from the others and in the end of the process all encrypted blocks are joined to form the encrypted file, also known as the ciphertext. The same stands for the decryption process too.

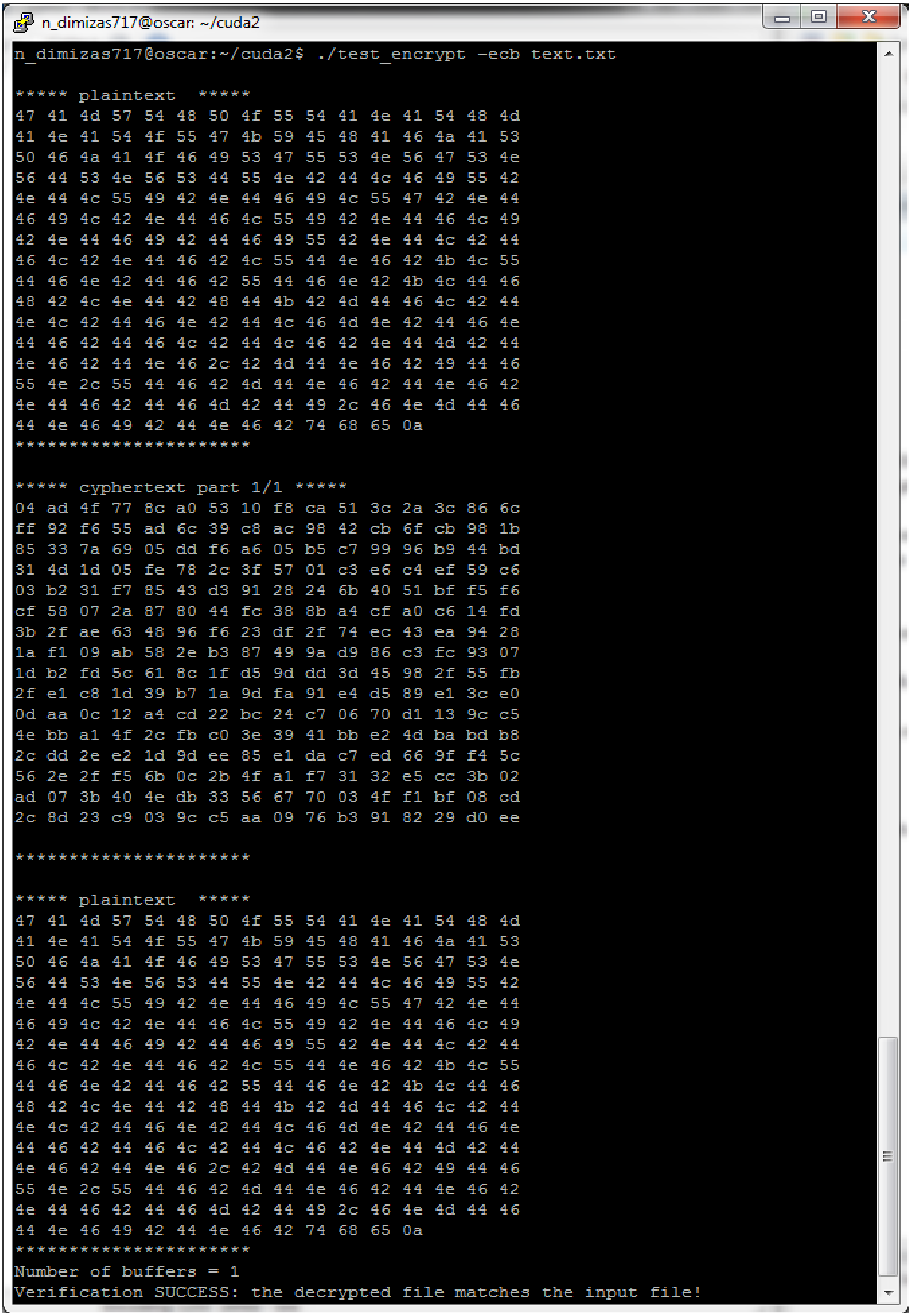
In addition, the C base code included two executable files, each working in quite a different way. The first one (named *test_encrypt*) was mainly aiming at showing how the whole process works. It received the input from the user using a terminal. Then it displayed the input file (also known as plaintext) in hexadecimal form, then the ciphertext (also in hexadecimal form) and in the end the plaintext again, in order to show that the initial and final plaintexts were equal, the way they should be. We included a verification method that checks the initial and ending plaintext to verify that the whole process was completed successfully.

The following figures depict an example of the test_encrypt execution:



```
n_dimizas717@oscar:~/Serial_Program$ ./test_encrypt -ecb eleos.txt
```

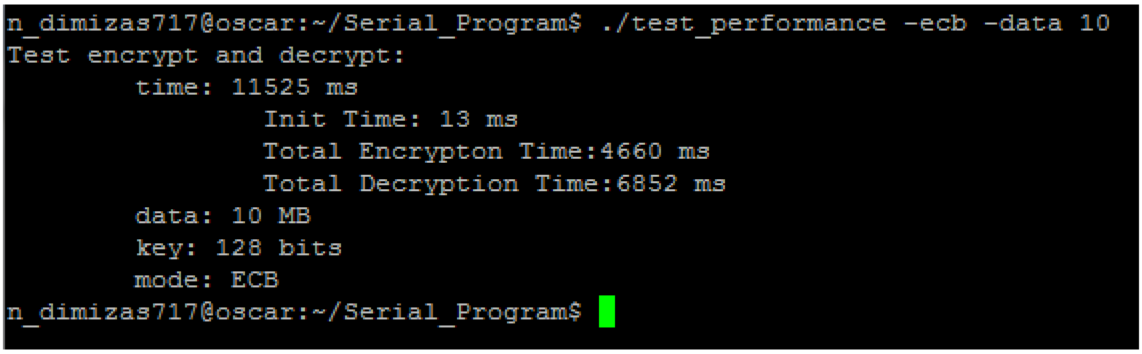
Figure 19: Execution example of test_encrypt executable 1



```
n_dimizas717@oscar:~/cuda2$ ./test_encrypt -ecb text.txt

***** plaintext  *****
47 41 4d 57 54 48 50 4f 55 54 41 4e 41 54 48 4d
41 4e 41 54 4f 55 47 4b 59 45 48 41 46 4a 41 53
50 46 4a 41 4f 46 49 53 47 55 53 4e 56 47 53 4e
56 44 53 4e 56 53 44 55 4e 42 44 4c 46 49 55 42
4e 44 4c 55 49 42 4e 44 46 49 4c 55 47 42 4e 44
46 49 4c 42 4e 44 46 4c 55 49 42 4e 44 46 4c 49
42 4e 44 46 49 42 44 46 49 55 42 4e 44 4c 42 44
46 4c 42 4e 44 46 42 4c 55 44 4e 46 42 4b 4c 55
44 46 4e 42 44 46 42 55 44 46 4e 42 4b 4c 44 46
48 42 4c 4e 44 42 48 44 4b 42 4d 44 46 4c 42 44
4e 4c 42 44 46 4e 42 44 4c 46 4d 4e 42 44 46 4e
44 46 42 44 46 4c 42 44 4c 46 42 4e 44 4d 42 44
4e 46 42 44 4e 46 2c 42 4d 44 4e 46 42 49 44 46
55 4e 2c 55 44 46 42 4d 44 4e 46 42 44 4e 46 42
4e 44 46 42 44 46 4d 42 44 49 2c 46 4e 4d 44 46
44 4e 46 49 42 44 4e 46 42 74 68 65 0a
**********************

***** cyphertext part 1/1 *****
04 ad 4f 77 8c a0 53 10 f8 ca 51 3c 2a 3c 86 6c
ff 92 f6 55 ad 6c 39 c8 ac 98 42 cb 6f cb 98 1b
85 33 7a 69 05 dd f6 a6 05 b5 c7 99 96 b9 44 bd
31 4d 1d 05 fe 78 2c 3f 57 01 c3 e6 c4 ef 59 c6
03 b2 31 f7 85 43 d3 91 28 24 6b 40 51 bf f5 f6
cf 58 07 2a 87 80 44 fc 38 8b a4 cf a0 c6 14 fd
3b 2f ae 63 48 96 f6 23 df 2f 74 ec 43 ea 94 28
1a f1 09 ab 58 2e b3 87 49 9a d9 86 c3 fc 93 07
1d b2 fd 5c 61 8c 1f d5 9d dd 3d 45 98 2f 55 fb
2f e1 c8 1d 39 b7 1a 9d fa 91 e4 d5 89 e1 3c e0
0d aa 0c 12 a4 cd 22 bc 24 c7 06 70 d1 13 9c c5
4e bb a1 4f 2c fb c0 3e 39 41 bb e2 4d ba bd b8
2c dd 2e e2 1d 9d ee 85 e1 da c7 ed 66 9f f4 5c
56 2e 2f f5 6b 0c 2b 4f a1 f7 31 32 e5 cc 3b 02
ad 07 3b 40 4e db 33 56 67 70 03 4f f1 bf 08 cd
2c 8d 23 c9 03 9c c5 aa 09 76 b3 91 82 29 d0 ee

**********************

***** plaintext  *****
47 41 4d 57 54 48 50 4f 55 54 41 4e 41 54 48 4d
41 4e 41 54 4f 55 47 4b 59 45 48 41 46 4a 41 53
50 46 4a 41 4f 46 49 53 47 55 53 4e 56 47 53 4e
56 44 53 4e 56 53 44 55 4e 42 44 4c 46 49 55 42
4e 44 4c 55 49 42 4e 44 46 49 4c 55 47 42 4e 44
46 49 4c 42 4e 44 46 4c 55 49 42 4e 44 46 4c 49
42 4e 44 46 49 42 44 46 49 55 42 4e 44 4c 42 44
46 4c 42 4e 44 46 42 4c 55 44 4e 46 42 4b 4c 55
44 46 4e 42 44 46 42 55 44 46 4e 42 4b 4c 44 46
48 42 4c 4e 44 42 48 44 4b 42 4d 44 46 4c 42 44
4e 4c 42 44 46 4e 42 44 4c 46 4d 4e 42 44 46 4e
44 46 42 44 46 4c 42 44 4c 46 42 4e 44 4d 42 44
4e 46 42 44 4e 46 2c 42 4d 44 4e 46 42 49 44 46
55 4e 2c 55 44 46 42 4d 44 4e 46 42 44 4e 46 42
4e 44 46 42 44 46 4d 42 44 49 2c 46 4e 4d 44 46
44 4e 46 49 42 44 4e 46 42 74 68 65 0a
**********************
Number of buffers = 1
Verification SUCCESS: the decrypted file matches the input file!
```

Figure 20: Execution example of test_encrypt executable 2

Moreover, the second executable (called *test_performance*) was intended for performance measurements exclusively, hence the name. The user provides no input data whatsoever, as it is automatically generated in the code. The only input the user is asked to give is the size of the file that will be encrypted (in megabytes), using the "–data X" flag. The X parameter can range between 1 to the maximum value of an int (integer) variable, which is 2,147,483,647 (we tested up to 10000mb ~= 10gb of data).

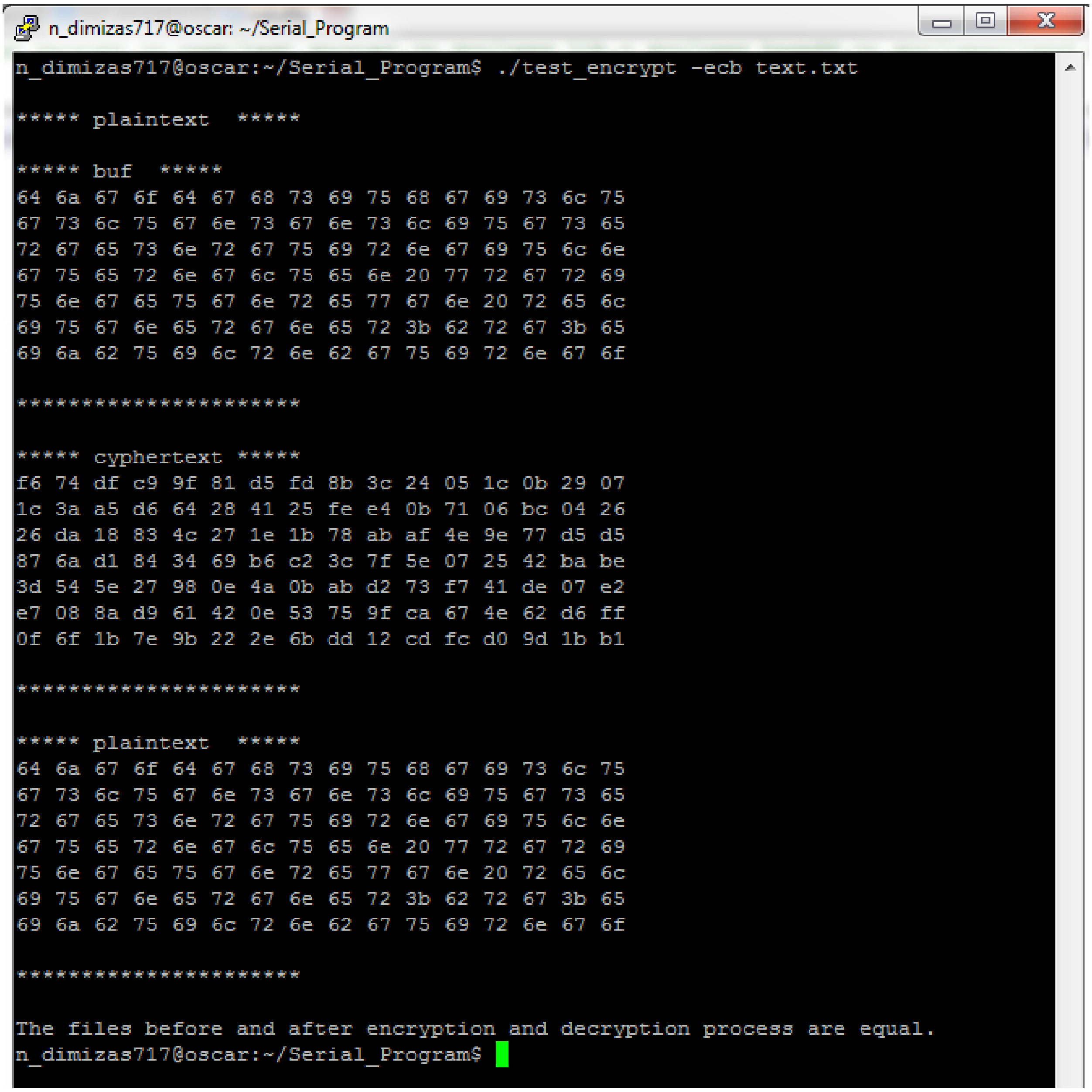


```
n_dimizas717@oscar:~/Serial_Program$ ./test_performance -ecb -data 10
Test encrypt and decrypt:
        time: 11525 ms
                Init Time: 13 ms
                Total Encrypton Time:4660 ms
                Total Decryption Time:6852 ms
        data: 10 MB
        key: 128 bits
        mode: ECB
n_dimizas717@oscar:~/Serial_Program$
```

Last but not least, the user is responsible to set the mode to ECB, using the "–ecb" flag, as the code was initially designed to work on other block cipher modes as well, such as CBC and GCM. There is also the optional choice of choosing a key size (in bits), using the "–key X" flag, with available values being 128, 192, and 256, as the AES standard allows for encryption. If the user doesn't specify a key size, the default value of 128-bit key is used.

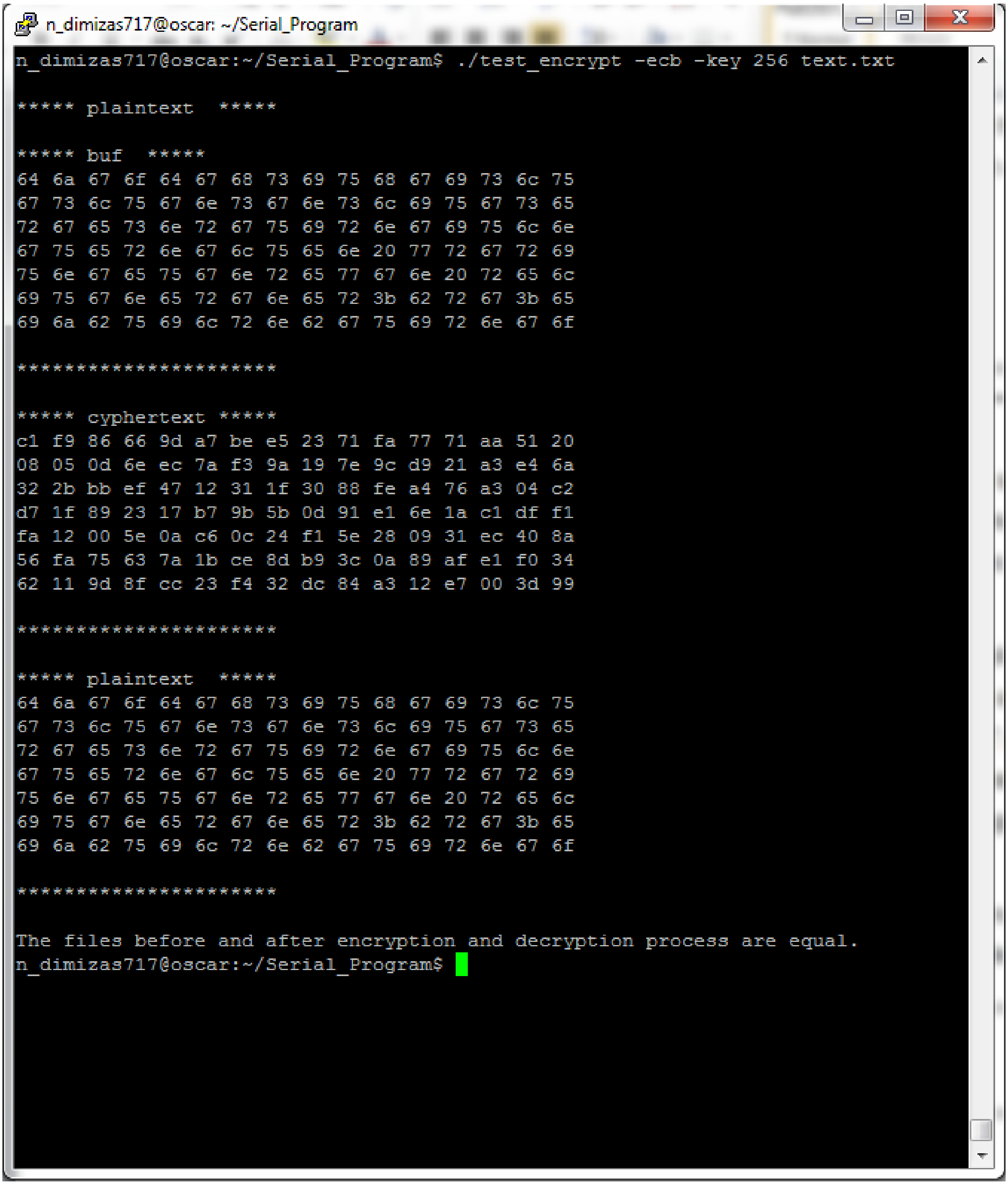
Naturally, tests performed on the sequential code were the longest by far, and larger keys led to an even longer test.

Below we show some cases of the execution of the serial code.



```
n_dimizas717@oscar:~/Serial_Program$ ./test_encrypt -ecb text.txt

***** plaintext  *****

***** buf  *****
64 6a 67 6f 64 67 68 73 69 75 68 67 69 73 6c 75
67 73 6c 75 67 6e 73 67 6e 73 6c 69 75 67 73 65
72 67 65 73 6e 72 67 75 69 72 6e 67 69 75 6c 6e
67 75 65 72 6e 67 6c 75 65 6e 20 77 72 67 72 69
75 6e 67 65 75 67 6e 72 65 77 67 6e 20 72 65 6c
69 75 67 6e 65 72 67 6e 65 72 3b 62 72 67 3b 65
69 6a 62 75 69 6c 72 6e 62 67 75 69 72 6e 67 6f

**********************

***** cyphertext *****
f6 74 df c9 9f 81 d5 fd 8b 3c 24 05 1c 0b 29 07
1c 3a a5 d6 64 28 41 25 fe e4 0b 71 06 bc 04 26
26 da 18 83 4c 27 1e 1b 78 ab af 4e 9e 77 d5 d5
87 6a d1 84 34 69 b6 c2 3c 7f 5e 07 25 42 ba be
3d 54 5e 27 98 0e 4a 0b ab d2 73 f7 41 de 07 e2
e7 08 8a d9 61 42 0e 53 75 9f ca 67 4e 62 d6 ff
0f 6f 1b 7e 9b 22 2e 6b dd 12 cd fc d0 9d 1b b1

**********************

***** plaintext  *****
64 6a 67 6f 64 67 68 73 69 75 68 67 69 73 6c 75
67 73 6c 75 67 6e 73 67 6e 73 6c 69 75 67 73 65
72 67 65 73 6e 72 67 75 69 72 6e 67 69 75 6c 6e
67 75 65 72 6e 67 6c 75 65 6e 20 77 72 67 72 69
75 6e 67 65 75 67 6e 72 65 77 67 6e 20 72 65 6c
69 75 67 6e 65 72 67 6e 65 72 3b 62 72 67 3b 65
69 6a 62 75 69 6c 72 6e 62 67 75 69 72 6e 67 6f

**********************

The files before and after encryption and decryption process are equal.
n_dimizas717@oscar:~/Serial_Program$
```

Figure 21: Baseline code execution 1



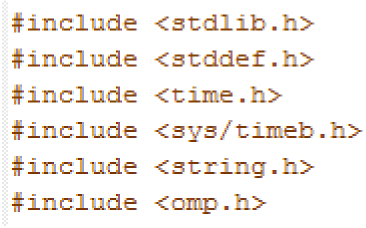
```
n_dimizas717@oscar:~/Serial_Program$ ./test_encrypt -ecb -key 256 text.txt

***** plaintext  *****

***** buf  *****
64 6a 67 6f 64 67 68 73 69 75 68 67 69 73 6c 75
67 73 6c 75 67 6e 73 67 6e 73 6c 69 75 67 73 65
72 67 65 73 6e 72 67 75 69 72 6e 67 69 75 6c 6e
67 75 65 72 6e 67 6c 75 65 6e 20 77 72 67 72 69
75 6e 67 65 75 67 6e 72 65 77 67 6e 20 72 65 6c
69 75 67 6e 65 72 67 6e 65 72 3b 62 72 67 3b 65
69 6a 62 75 69 6c 72 6e 62 67 75 69 72 6e 67 6f

**********************

***** cyphertext *****
c1 f9 86 66 9d a7 be e5 23 71 fa 77 71 aa 51 20
08 05 0d 6e ec 7a f3 9a 19 7e 9c d9 21 a3 e4 6a
32 2b bb ef 47 12 31 1f 30 88 fe a4 76 a3 04 c2
d7 1f 89 23 17 b7 9b 5b 0d 91 e1 6e 1a c1 df f1
fa 12 00 5e 0a c6 0c 24 f1 5e 28 09 31 ec 40 8a
56 fa 75 63 7a 1b ce 8d b9 3c 0a 89 af e1 f0 34
62 11 9d 8f cc 23 f4 32 dc 84 a3 12 e7 00 3d 99

**********************

***** plaintext  *****
64 6a 67 6f 64 67 68 73 69 75 68 67 69 73 6c 75
67 73 6c 75 67 6e 73 67 6e 73 6c 69 75 67 73 65
72 67 65 73 6e 72 67 75 69 72 6e 67 69 75 6c 6e
67 75 65 72 6e 67 6c 75 65 6e 20 77 72 67 72 69
75 6e 67 65 75 67 6e 72 65 77 67 6e 20 72 65 6c
69 75 67 6e 65 72 67 6e 65 72 3b 62 72 67 3b 65
69 6a 62 75 69 6c 72 6e 62 67 75 69 72 6e 67 6f

**********************

The files before and after encryption and decryption process are equal.
n_dimizas717@oscar:~/Serial_Program$
```

Figure 22: Baseline code execution 2

The first and the last part of both our implementations is exactly the same as with the serial C code and includes initialization of some parameters, the key creation(in a custom random way – not using the built-in random library), key expansion, as well as key destruction.

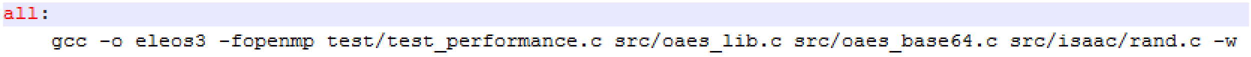
As far as the OpenMp is concerned, there is not much thought to take place. As we have mentioned earlier, OpenMP automates things for thread parallelism on a higher level so it requires minimal developer intervention. The use of “#pragma omp” parallel and “#pragma omp for” brackets divides the work of the for-loop into threads and automates the process by itself. It is worth noting that it takes full advantage of all available threads (in our case four).

For the transition to this approach, little intervention was needed, mainly including the OpenMP library header file to our .c source files, with this simple line of code:



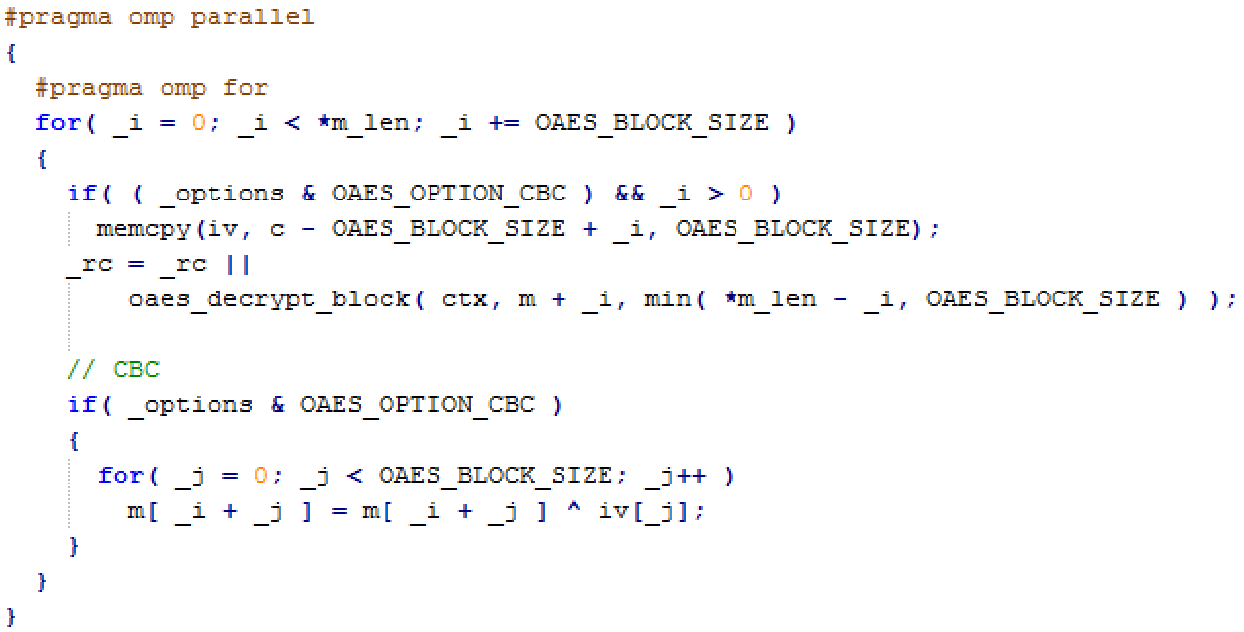
```
#include <stdlib.h>
#include <stddef.h>
#include <time.h>
#include <sys/timeb.h>
#include <string.h>
#include <omp.h>
```

Therefore, the compilation call in our makefile looks like the following:



```
all:
    gcc -o eleos3 -fopenmp test/test_performance.c src/oaes_lib.c src/oaes_base64.c src/isaac/rand.c -w
```

Other than that, the aforementioned #pragma brackets were basically the only addition to the base C serial code:



```
#pragma omp parallel
{
  #pragma omp for
  for( _i = 0; _i < *m_len; _i += OAES_BLOCK_SIZE )
  {
    if( ( _options & OAES_OPTION_CBC ) && _i > 0 )
      memcpy(iv, c - OAES_BLOCK_SIZE + _i, OAES_BLOCK_SIZE);
    _rc = _rc ||
        oaes_decrypt_block( ctx, m + _i, min( *m_len - _i, OAES_BLOCK_SIZE ) );

    // CBC
    if( _options & OAES_OPTION_CBC )
    {
      for( _j = 0; _j < OAES_BLOCK_SIZE; _j++ )
        m[ _i + _j ] = m[ _i + _j ] ^ iv[_j];
    }
  }
}
```

What actually takes place in this part of code is that each loop is assigned to one OpenMP thread, which means that –in a simple case- a 40-times loop would result in 4 threads performing 10 loops each. In our encryption example, each for loop represents the initialization and encryption of one block of the input file. Therefore, at any given moment a maximum of 4 blocks are handled concurrently by 4 threads of the CPU, which obviously accelerates the whole process.

We should also mention that we changed the timing used in the serial code, which was in seconds, to milliseconds for greater accuracy. That is also the case for both the OpenMP and CUDA approach.

On the other hand, the CUDA approach was significantly more complicated. We worked on two separate ways on using CUDA for the AES algorithm, which resulted in two executables, just like in the serial code.

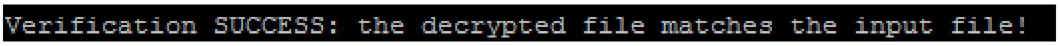
Both of these test executables, while differ on the input method as we mentioned, use the same cuda (.cu) file for the enctyption/decryption process, so the process can be explained in the same way for both of them.

After the initialization process, the input is analyzed to determine how many blocks it will be divided to. If the input size in chars (and in our case, bytes, as a character is converted to a uint8_t type variable which is an integer of 1byte) is divisible by 16 (the block size) then the quotient will be the number of CUDA threads that will be used. Each CUDA thread will encrypt its dedicated block. In case the size is not divisible by 16, we take the integer quotient and add one more block for the remainder, e.g if size=70, the number of threads needed will be integer(70/16) +1 = 4 + 1 = 5.

Next, there are some initializations to be done before we can launch the CUDA code (kernel). Most importantly, we have to allocate (*cudaMalloc*) GPU memory for the plaintext to be passed to the CUDA code, along with a struct (called *ctx*), which contains the key struct and some other parameters. After the allocation process, we have to copy the values of the data we need from the CPU memory to the GPU memory (*cudaMemCpy* host to device), so that it can be accessible from the kernel. It is noteworthy that some variables (mostly the ones that are read-only by the CUDA code) can be passed by value rather than copied implicitly to the GPU memory.

During the previous process, the whole input buffer is copied to the GPU memory and is accessible by all threads. Each thread will only deal with one block, though. After the kernel is called (__*global*__ function), the last thread is responsible to check if the last block of the plaintext must be padded. In case size is not a product of 16 in bytes, then the last block will be incomplete and the padding process serves in filling it. In general, the __*device*__ function which is responsible for the block encryption is called by each thread, using a different offset of the plaintext buffer as a function argument. In the end of the procedure, each thread writes back its block to the buffer (which still resides in the GPU memory) and the CUDA kernel terminates. The host code (C) is then responsible to copy the buffer and some other parameters (e.g return status) back to the CPU memory (*cudaMemCpy* device to host) so that it is accessible from the CPU. The host code now has access to the ciphertext. The decryption process works likewise, and when it is finished, the host code has received the final plaintext.

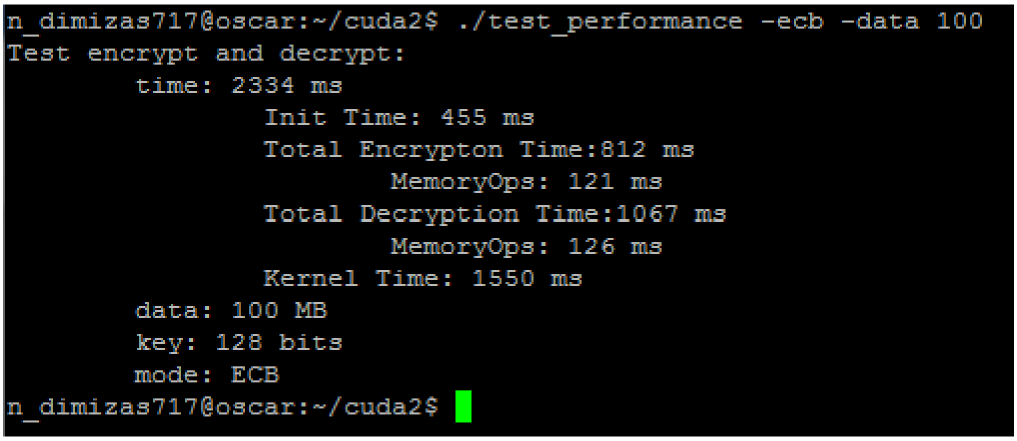
For the *test_encrypt* executable, the data is read from a text file, and in the end, as a part of an error-checking procedure, the program compares the final plaintext with the initial input and reports back to the user. A successful run should display that the two buffers are exactly the same:



```
Verification SUCCESS: the decrypted file matches the input file!
```

Figure 23: Verification Message

For the *test_performance* executable, the procedure explained above is repeated for a 1mb buffer in a loop. The times of the loop are determined by the user, using the "–data X" flag we mentioned earlier. For example, with a "-data 100" flag, a random 1mb buffer with data will be generated, and it will be encrypted and decrypted 100 times, displaying detailed timing results at the end of its execution:

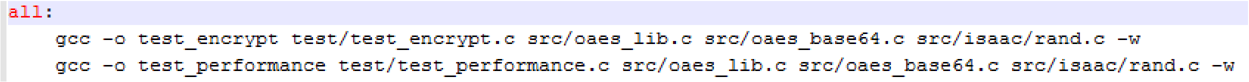


```
n_dimizas717@oscar:~/cuda2$ ./test_performance -ecb -data 100
Test encrypt and decrypt:
        time: 2334 ms
                Init Time: 455 ms
                Total Encrypton Time:812 ms
                        MemoryOps: 121 ms
                Total Decryption Time:1067 ms
                        MemoryOps: 126 ms
                Kernel Time: 1550 ms
        data: 100 MB
        key: 128 bits
        mode: ECB
n_dimizas717@oscar:~/cuda2$
```

Figure 24: Detailed timing results

Because of our limitation on CUDA blocks/threads that can run simultaneously, in our *test_encrypt* implementation, we have divided the initial file to 8000-char buffers that get encrypted/decrypted and then combined to form the final output. This way, the algorithm will run successfully no matter how large the input text (.txt) file is.

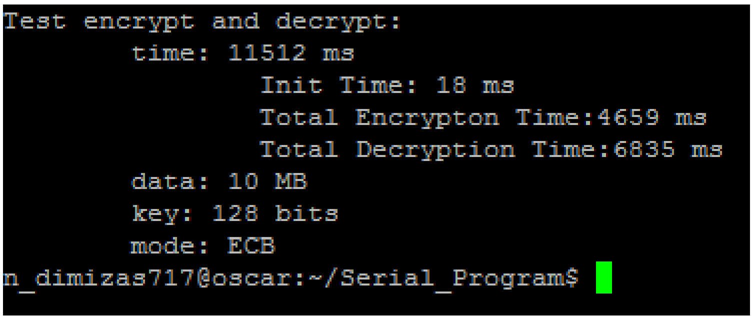
For compilation, we used the nvcc compiler (v6.5.12) and several .c and .cu source files. The content of our corresponding makefile is the following:



```
all:
	gcc -o test_encrypt test/test_encrypt.c src/oaes_lib.c src/oaes_base64.c src/isaac/rand.c -w
	gcc -o test_performance test/test_performance.c src/oaes_lib.c src/oaes_base64.c src/isaac/rand.c -w
```

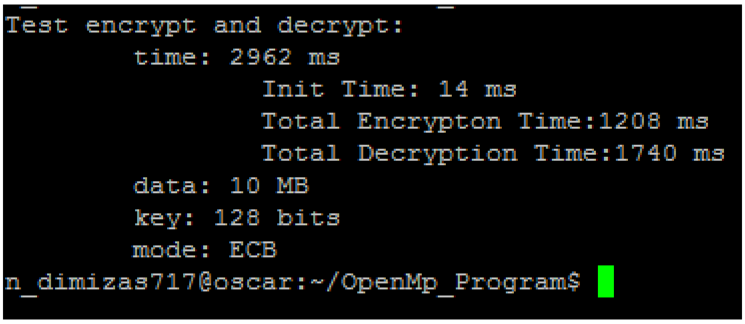
Test files were left almost intact through the transition from C to OpenMP/CUDA, with the only changes involving the way the input was received by our program, as we have already mentioned.

It is also worth noting that the CUDA implementation contains more detailed timing, as it is important to observe the timings for total execution, memory operations (cudaMalloc, cudafree, cudaMemCpy host-to-device and device-to-host), encryption/decryption timings, etc. This is shown below:



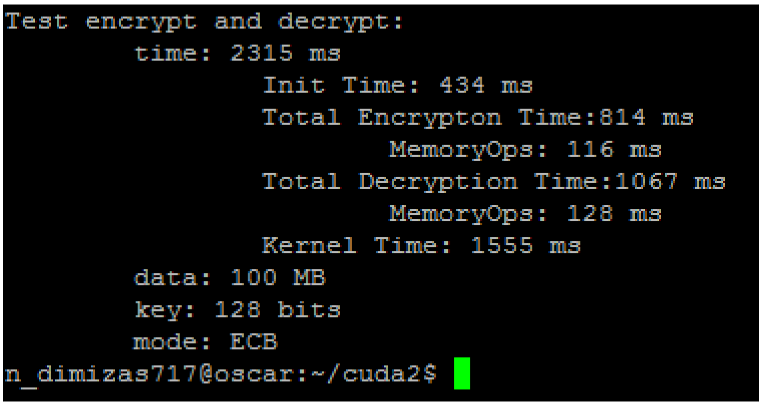
```
Test encrypt and decrypt:
        time: 11512 ms
                Init Time: 18 ms
                Total Encrypton Time:4659 ms
                Total Decryption Time:6835 ms
        data: 10 MB
        key: 128 bits
        mode: ECB
n_dimizas717@oscar:~/Serial_Program$
```

Figure 25:Serial code timings



```
Test encrypt and decrypt:
        time: 2962 ms
                Init Time: 14 ms
                Total Encrypton Time:1208 ms
                Total Decryption Time:1740 ms
        data: 10 MB
        key: 128 bits
        mode: ECB
n_dimizas717@oscar:~/OpenMp_Program$
```

Figure 26: OpenMP timings



```
Test encrypt and decrypt:
        time: 2315 ms
                Init Time: 434 ms
                Total Encrypton Time:814 ms
                        MemoryOps: 116 ms
                Total Decryption Time:1067 ms
                        MemoryOps: 128 ms
                Kernel Time: 1555 ms
        data: 100 MB
        key: 128 bits
        mode: ECB
n_dimizas717@oscar:~/cuda2$
```

Figure 27: CUDA timings

5.3 Code Optimization

After the developing of all the features that we intended to include in our code, and test cases using all available parameters (including different key sizes, smaller or larger files, etc.), we had to find ways to further optimize our code, given the algorithm we had chosen and the possibilities that were offered.

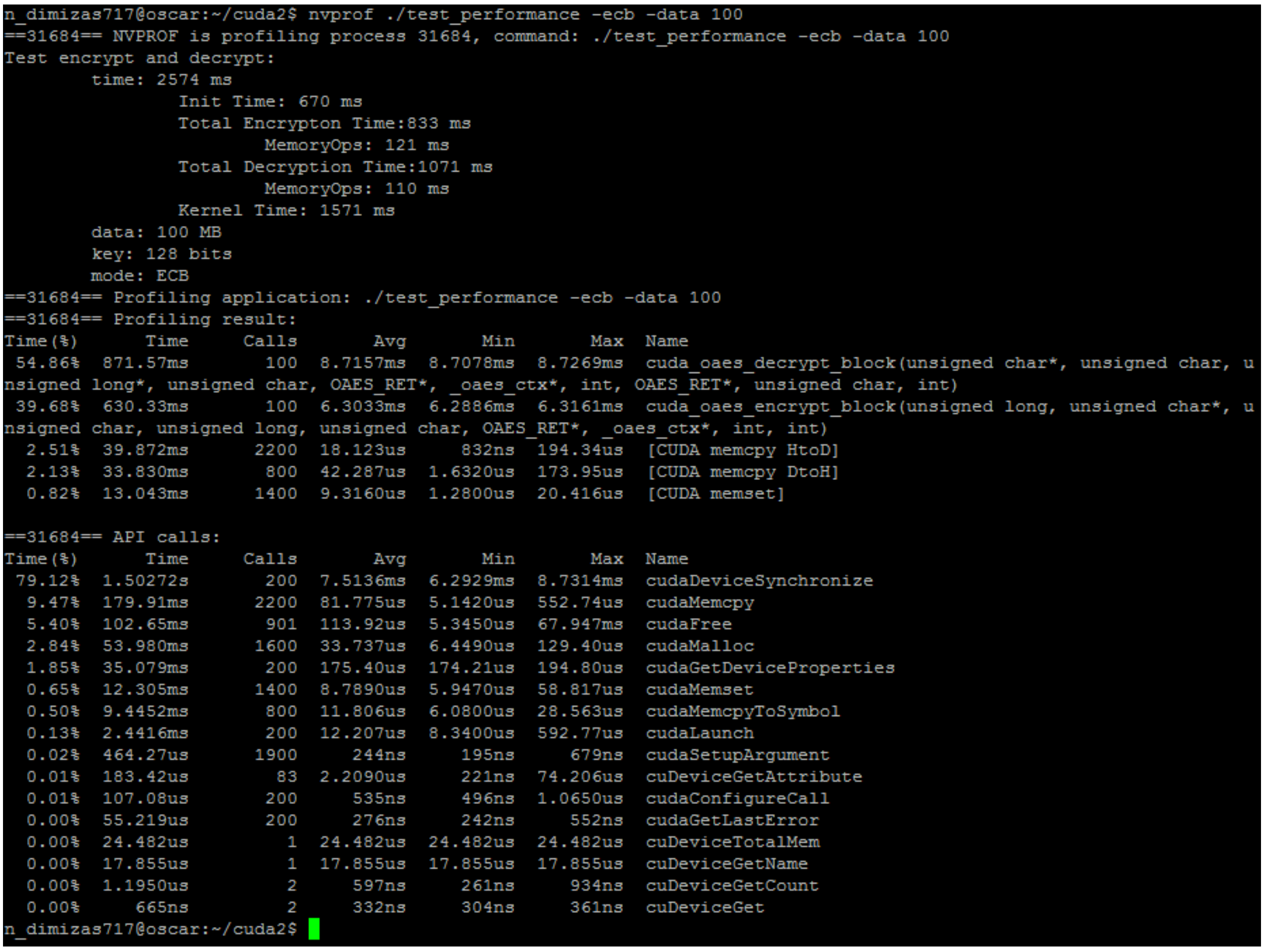
Firstly, as far as OpenMP is concerned, we trusted there were no further significant optimizations that could be included. Since OpenMP is a high level library and we were called to apply it to an existing piece of C code, there was no room for new ideas to reduce execution time or memory requirements.

On the other hand, CUDA works in quite a different way. In our first try, all needed parameters for encryption and decryption were copied to GPU memory using CudaMemCpy. As we moved deeper in CUDA programming we realized that certain values could be passed by value, avoiding the memory operations overhead which could prove to be quite costly, especially on large test cases.

Using the standard nvcc profiler (nvprof) we could take a deeper look at what functions are called, when and how many times, depending on the . The –print-gpu-trace includes all cuda API calls made as well as detailed info for each one of them.

It was obvious that the memory operations were a bit more than we wanted or expected. We decided to omit every possible memory allocation and copy process that was unnecessary, using by-value argument function calls when the data was read-only.

The improvement was visible, definitely not great, but the results got even better by a factor of 5% and we are convinced that we no longer include anything but the absolutely necessary operations in our code, as far as our knowledge goes.



```
n_dimizas717@oscar:~/cuda2$ nvprof ./test_performance -ecb -data 100
==31684== NVPROF is profiling process 31684, command: ./test_performance -ecb -data 100
Test encrypt and decrypt:
        time: 2574 ms
                Init Time: 670 ms
                Total Encrypton Time:833 ms
                        MemoryOps: 121 ms
                Total Decryption Time:1071 ms
                        MemoryOps: 110 ms
                Kernel Time: 1571 ms
        data: 100 MB
        key: 128 bits
        mode: ECB
==31684== Profiling application: ./test_performance -ecb -data 100
==31684== Profiling result:
Time(%)      Time     Calls       Avg       Min       Max  Name
 54.86%  871.57ms       100  8.7157ms  8.7078ms  8.7269ms  cuda_oaes_decrypt_block(unsigned char*, unsigned char, u
nsigned long*, unsigned char, OAES_RET*, _oaes_ctx*, int, OAES_RET*, unsigned char, int)
 39.68%  630.33ms       100  6.3033ms  6.2886ms  6.3161ms  cuda_oaes_encrypt_block(unsigned long, unsigned char*, u
nsigned char, unsigned long, unsigned char, OAES_RET*, _oaes_ctx*, int, int)
  2.51%  39.872ms      2200  18.123us     832ns  194.34us  [CUDA memcpy HtoD]
  2.13%  33.830ms       800  42.287us  1.6320us  173.95us  [CUDA memcpy DtoH]
  0.82%  13.043ms      1400  9.3160us  1.2800us  20.416us  [CUDA memset]

==31684== API calls:
Time(%)      Time     Calls       Avg       Min       Max  Name
 79.12%  1.50272s       200  7.5136ms  6.2929ms  8.7314ms  cudaDeviceSynchronize
  9.47%  179.91ms      2200  81.775us  5.1420us  552.74us  cudaMemcpy
  5.40%  102.65ms       901  113.92us  5.3450us  67.947ms  cudaFree
  2.84%  53.980ms      1600  33.737us  6.4490us  129.40us  cudaMalloc
  1.85%  35.079ms       200  175.40us  174.21us  194.80us  cudaGetDeviceProperties
  0.65%  12.305ms      1400  8.7890us  5.9470us  58.817us  cudaMemset
  0.50%  9.4452ms       800  11.806us  6.0800us  28.563us  cudaMemcpyToSymbol
  0.13%  2.4416ms       200  12.207us  8.3400us  592.77us  cudaLaunch
  0.02%  464.27us      1900     244ns     195ns     679ns  cudaSetupArgument
  0.01%  183.42us        83  2.2090us     221ns  74.206us  cuDeviceGetAttribute
  0.01%  107.08us       200     535ns     496ns  1.0650us  cudaConfigureCall
  0.00%  55.219us       200     276ns     242ns     552ns  cudaGetLastError
  0.00%  24.482us         1  24.482us  24.482us  24.482us  cuDeviceTotalMem
  0.00%  17.855us         1  17.855us  17.855us  17.855us  cuDeviceGetName
  0.00%  1.1950us         2     597ns     261ns     934ns  cuDeviceGetCount
  0.00%     665ns         2     332ns     304ns     361ns  cuDeviceGet
n_dimizas717@oscar:~/cuda2$
```

Figure 28: nvcc profiler example

5.4 Main Difficulties

Despite our acceptable level of knowledge on cryptography in general, the AES algorithm and a certain amount of experience and skill in CUDA programming, we came to find out that we needed to dive in a more extensive study of both areas. Our initial idea was solid enough and possible to realize, but certain details needed further investigation before we could proceed any further.

Despite the fact that the theoretical block encryption rounds and steps seem to be simple enough in a theoretical level, they proved to be somewhat harder to understand while written in C code. Luckily enough, the code was clean and straightforward, so step by step we were able to overcome all our difficulties and obtain an excellent grasp of what happens to the block on each round and step, as well as the usefulness of each step.

In addition, we were curious to find out the main idea behind the S-box and the other pre-configured tables used in each step. To clarify, even if it out of the general scope of this document, we wondered what makes a “good” S-box, how one can fill it in the best possible way, and what difference could any changes to it possible make. That was, admittedly, a more complex issue but we wanted to obtain as complete as possible knowledge on the subject of our thesis, so we invested a portion of our time on this matter.

Moreover, we studied the significance of a random key, the impact of the predictability of the key on the cipher’s security and what makes a random key generator better in general. We found out that if a key (or even a part of the key) is in any way predictable, it could lead to a breach of security on the cipher, especially on simpler block cipher codes, like ECB. That is because, due to the fact that the same key is used on each block, two same blocks encrypted with AES ECB will produce the same cipherblock (unlike chain ciphers like CBC where cipherblocks depend on previous cipherblocks), possibly leading to security flaws. Luckily the base C code we received had a custom random key generator that, from our research, was acceptably capable of producing random keys.

On the other hand, since CUDA is relatively new, documentation was almost our only advisor to developing. Several problems we had to face were not examined on any forum or site, leading us to face our most severe difficulties in the development part. The CUDA documentation is quite complete and easy to understand, but we thought some areas could have been explained better. One particular problem we had to deal with, was the CUDA kernels not running because of improper set of max-registers-per-thread, which is defined on compilation by the way. In several cases, kernels could not run at all, while on others only some of the threads were able to run. In other cases where many registers are being used can cause a low maximum occupancy and thus cause a number of processing cores to remain idle, which can impact performance.

After excessive research we chose a number of max-registers-per-thread that could achieve stability throughout the program while assuring that threads are not making unnecessary use of registers.

5.5 Comparison and Results

This is the part of the thesis we put our implementations to test in real world data and compare it to each other, as well as the serial C code. From what we have already analyzed, and given the true capabilities of threaded programming and most importantly a General Purpose GPU Computing (GPGPU computing) program, the results should lead us to positive conclusions. Let’s find out.

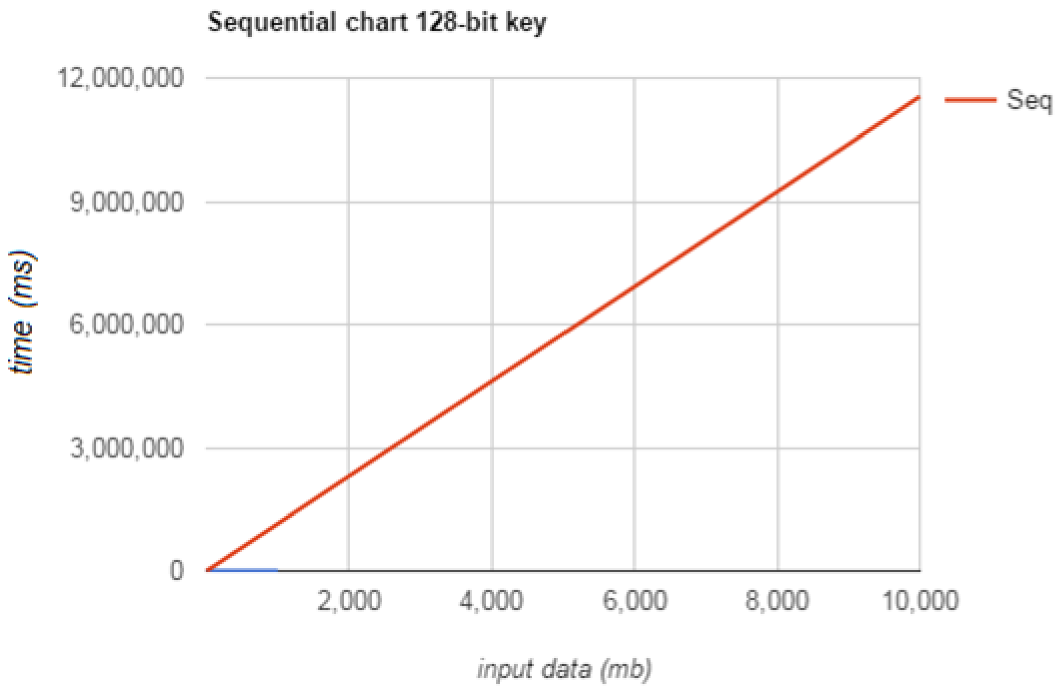
During the tests, we used the *test_performance* executable, which, as we mentioned, uses user defined size but random generated data for its test purposes and then displays the results. The data parameters we used in all Serial, OpenMP and CUDA executables in megabytes are 1, 5, 10, 50, 100, 1000, 10000 and were chosen carefully in order to better depict the impact of parallelism in a wide range of data sizes. We can safely assume that in even larger datasets, the results will be similar to the 10000mb execution.

Below we present a series of tables and charts, which we trusted that better depict our results in each case. We compare the execution time in each implementation whilst changing the key size, compute the speedup, explain the impact of the key size on each one and of course, compare all those implementations, which was our main cause to start with.

5.5.1 Sequential Results

Firstly, we present the info we gathered from the sequential/serial program executions. These serve only as a comparative to any of the other parallel implementations we are discussing.

Sequential 128-bit key results:



Graph 1: Sequential 128-bit key chart

What could help better explain the previous graph, is the throughput that the serial code is able to handle on a certain amount of time, for example per second. The following table demonstrates how the throughput varies with the input size.

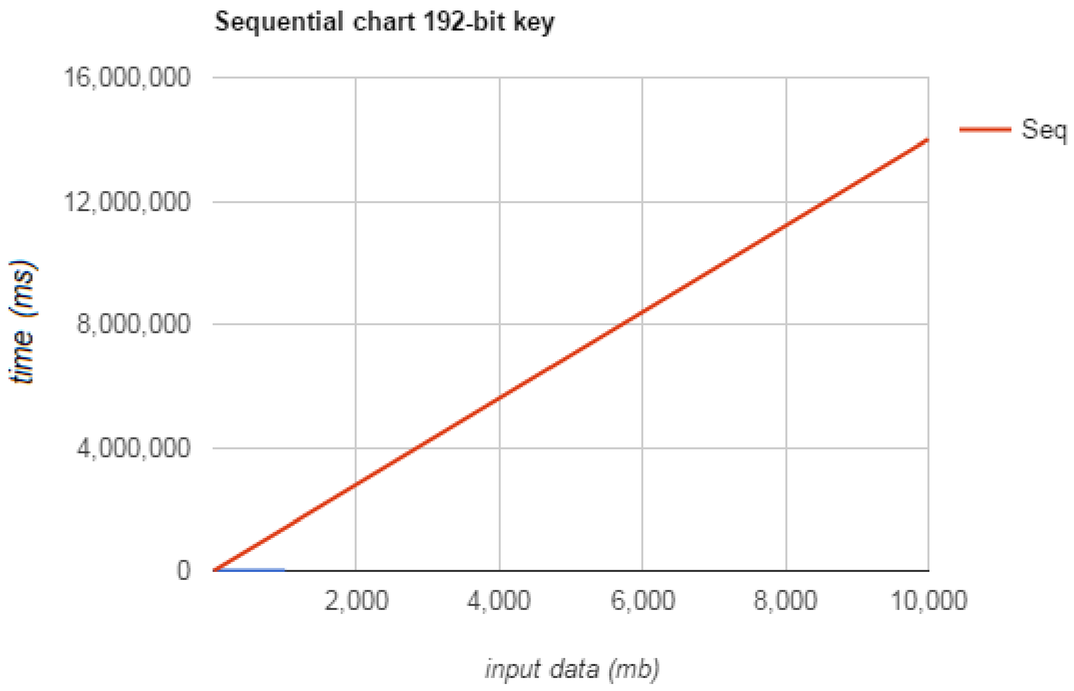
Size (MB)	1	5	10	50	100	1000	10000
Throughput (MB/s)	0.85	0.86	0.86	0.86	0.87	0.87	0.87

Table 3: Sequential throughput 128-bit key

As we can see, there is not much variation, which is to be expected, because of the nature of the serial program. The same argument stands for the other two key sizes, of course, and will be verified shortly. The code is able to process an average of 0.86mb of data input per second during our 128-bit key tests.

We can expect a smaller average throughput on our 192-bit and 256-bit key tests, given the fact that the 128-bit test runs 10 rounds of encryption on each block.

Sequential 192-bit key results:



Graph 2: Sequential 192-bit key chart

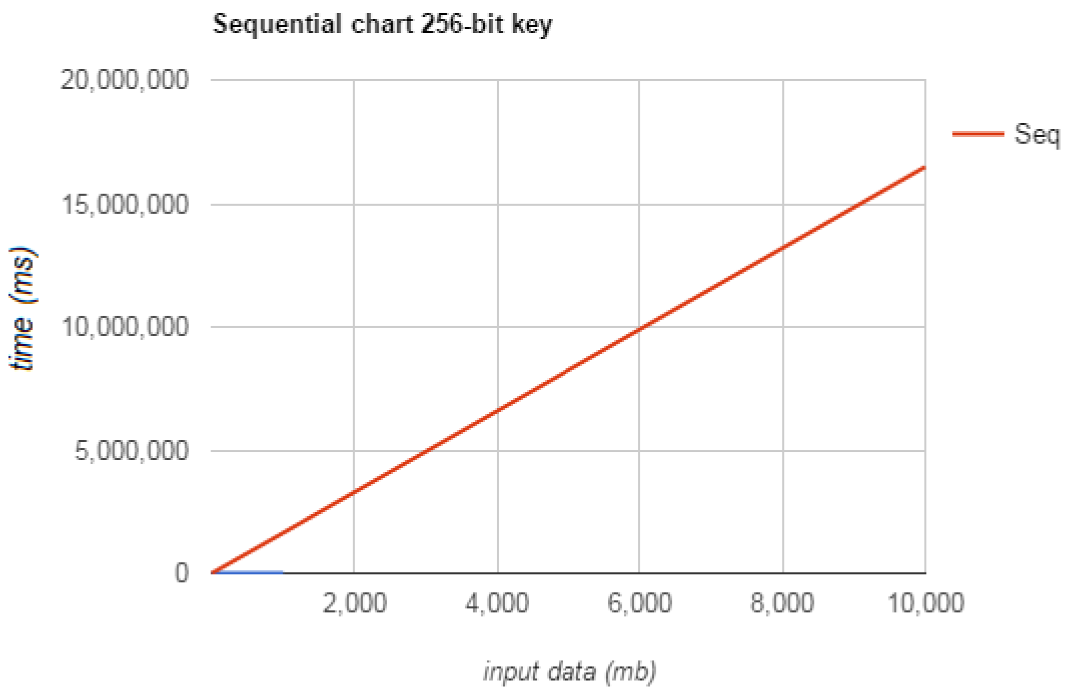
From the chart above, we will now proceed to present our throughput results:

Size (MB)	1	5	10	50	100	1000	10000
Throughput (MB/s)	0.71	0.71	0.72	0.72	0.72	0.72	0.72

Table 4: Sequential throughput 192-bit key

The average throughput is 0.72mb/s, which is less than our previous tests with the 128-bit key, naturally. That verifies our previous expectations. Each block goes through 2 additional rounds of encryption (12 in total) when a 192-bit key is used which explains why the program cannot keep up with our 128-bit key results.

Sequential 256-bit key results:



Graph 3: Sequential 256-bit key chart

The throughput results are as follows:

Size (MB)	1	5	10	50	100	1000	10000
Throughput (MB/s)	0.6	0.61	0.62	0.61	0.61	0.61	0.61

Table 5: Sequential throughput 256-bit key

For the same reasons as described above, the 256-bit key test throughput falls behind both the 128-bit and 192-bit key throughputs, because it involves 14 rounds of encryption on each block (4 more than the 128-bit key test and 2 more than the 192-bit key test respectively). It averages on 0.61mb/s.

The first thing that pops to the eye is that the key size affects real world timings in the execution. This is totally natural and can be easily explained due to the fact that the size of the key in the AES algorithm defines the number of rounds that each block of the file will go through in the process of both the encryption and the decryption. It is a part of the AES algorithm, as standardized. As a matter of fact, the 128bit key sets the number of rounds (Nr) to 10, the 192bit key to 12 and the 256bit key to 14. Therefore, this is something expected, and should affect not only the serial, but the OpenMP and CUDA implementations as well.

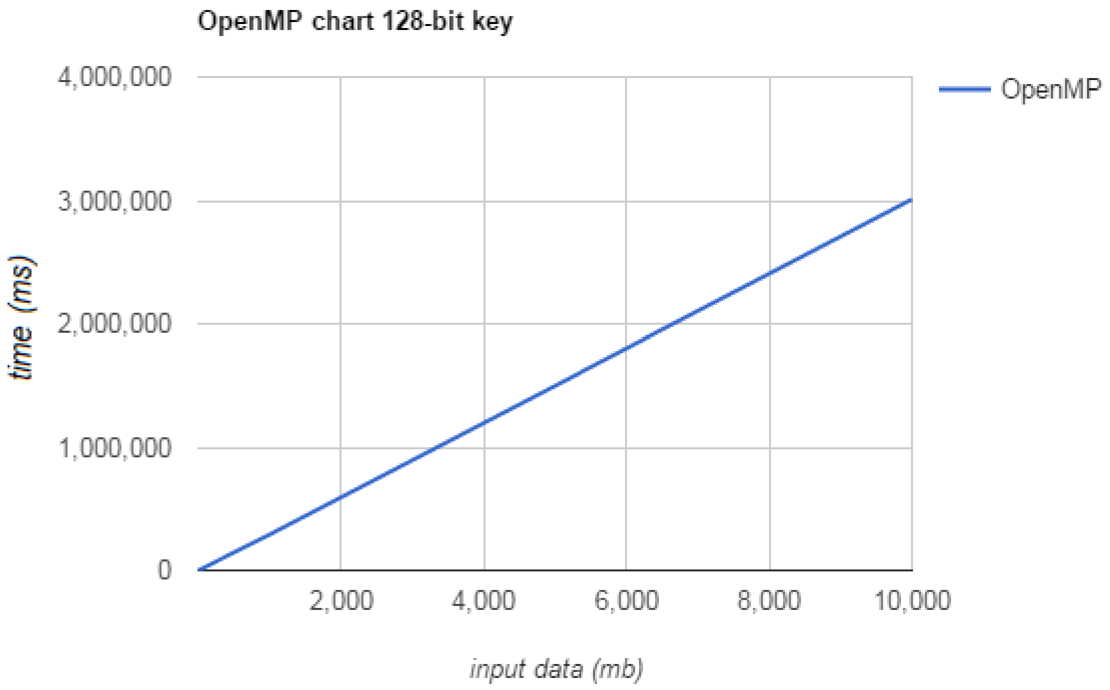
How much does the size of the key affect real world time, though? For comparison purposes we present the following table, for 1gb of data (though in this implementation, the percentages should be roughly the same regardless of the data size we compare against different key sizes):

Key size	Performance compared to 128-bit key execution time
128-bit	100%
192-bit	119%
256-bit	141%

Table 6: Sequential key size - performance

5.5.2 OpenMP Results

Now that we set our comparative base, let’s take a look at the OpenMP results:



Graph 4: OpenMP 128-bit key chart

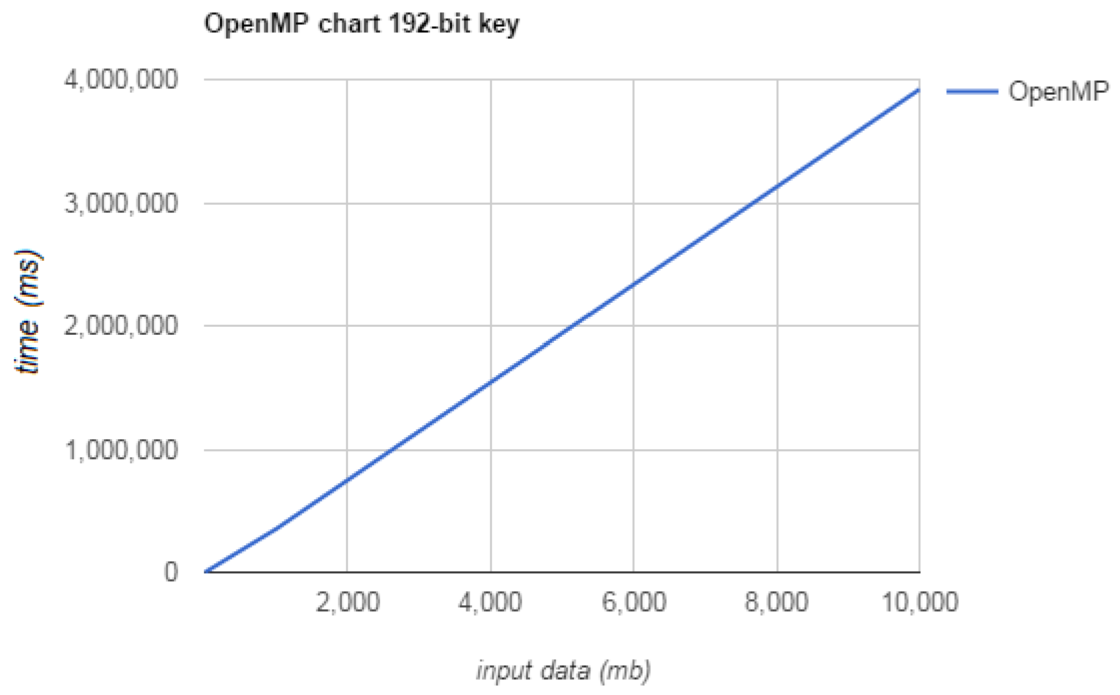
The corresponding throughput table is the following

Size (MB)	1	5	10	50	100	1000	10000
Throughput (MB/s)	3.1	3.4	3.4	3.4	3.4	3.4	3.3

Table 7: OpenMP throughput 128-bit key

It may seem strange that on 10gb of data input, the throughput drops significantly. That is probably because of cache memory misses and the time that is requires to replace dirty blocks on the cache. The cache is a very fast type of memory that is embedded in the CPU and is used to accelerate calculations. Basically the CPU often copies blocks of memory from the RAM to the cache in order to allow it to have much faster access to the data later. In large data sets, the cache may get full and it may be necessary to replace some of these entries, and that explains the slightly elevated timing levels on the 10gb test.

OpenMP 192-bit key results:



Graph 5: OpenMP 192-bit key chart

The corresponding throughput table is the following

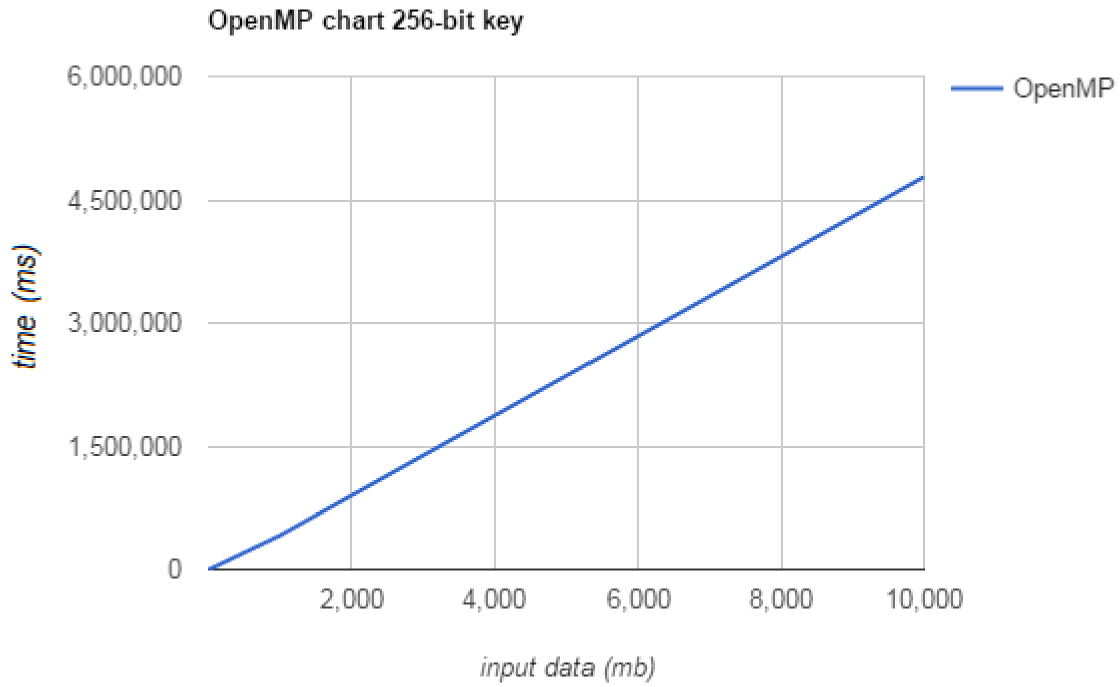
Size (MB)	1	5	10	50	100	1000	10000
Throughput (MB/s)	2.56	2.8	2.8	2.8	2.8	2.8	2.55

Table 8: OpenMP throughput 192-bit key

Once again we can observe that the throughput drops on the 10gb data test. This is due to the same reasons as in the 128-bit key tests, and we can expect it on the 256-key test that follows, as well.

The data throughput capabilities of the OpenMP program is naturally reduced on each test compared to the 128-bit, for the same reasons we explained on the sequential program. The 2 additional rounds of the encryption and decryption processes slow down the whole process, resulting in less efficiency and thus throughput capability.

OpenMP 256-bit key results:



Graph 6: OpenMP 256-bit key chart

The corresponding throughput table is the following

Size (MB)	1	5	10	50	100	1000	10000
Throughput (MB/s)	2.3	2.3	2.3	2.3	2.4	2.4	2.1

Table 9: OpenMP throughput 256-bit key

Once again, both the 10GB test throughput is reduced due to cache memory reasons, and each individual throughput is reduced even more due to the 14 rounds of encryption and decryption. Our expectations are verified.

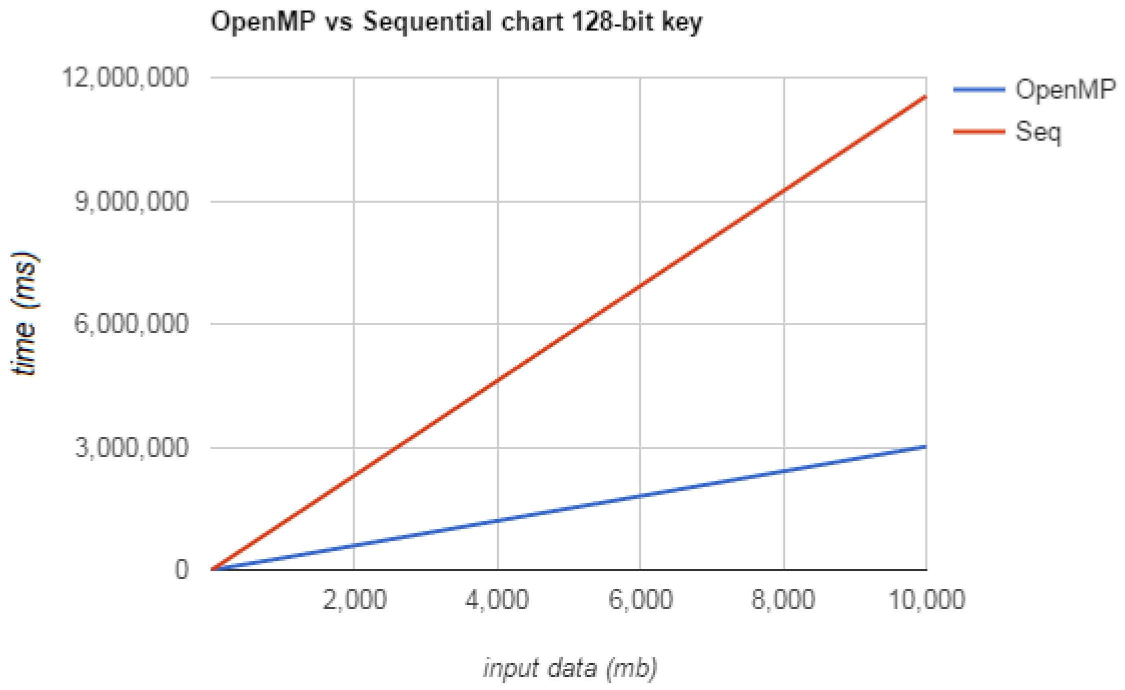
How much does the key affect the real-time timings though? It can be easily presented through this table:

Key size	Performance compared to 128-bit key execution time
128-bit	100%
192-bit	121%
256-bit	141%

Table 10: OpenMP key size-performance

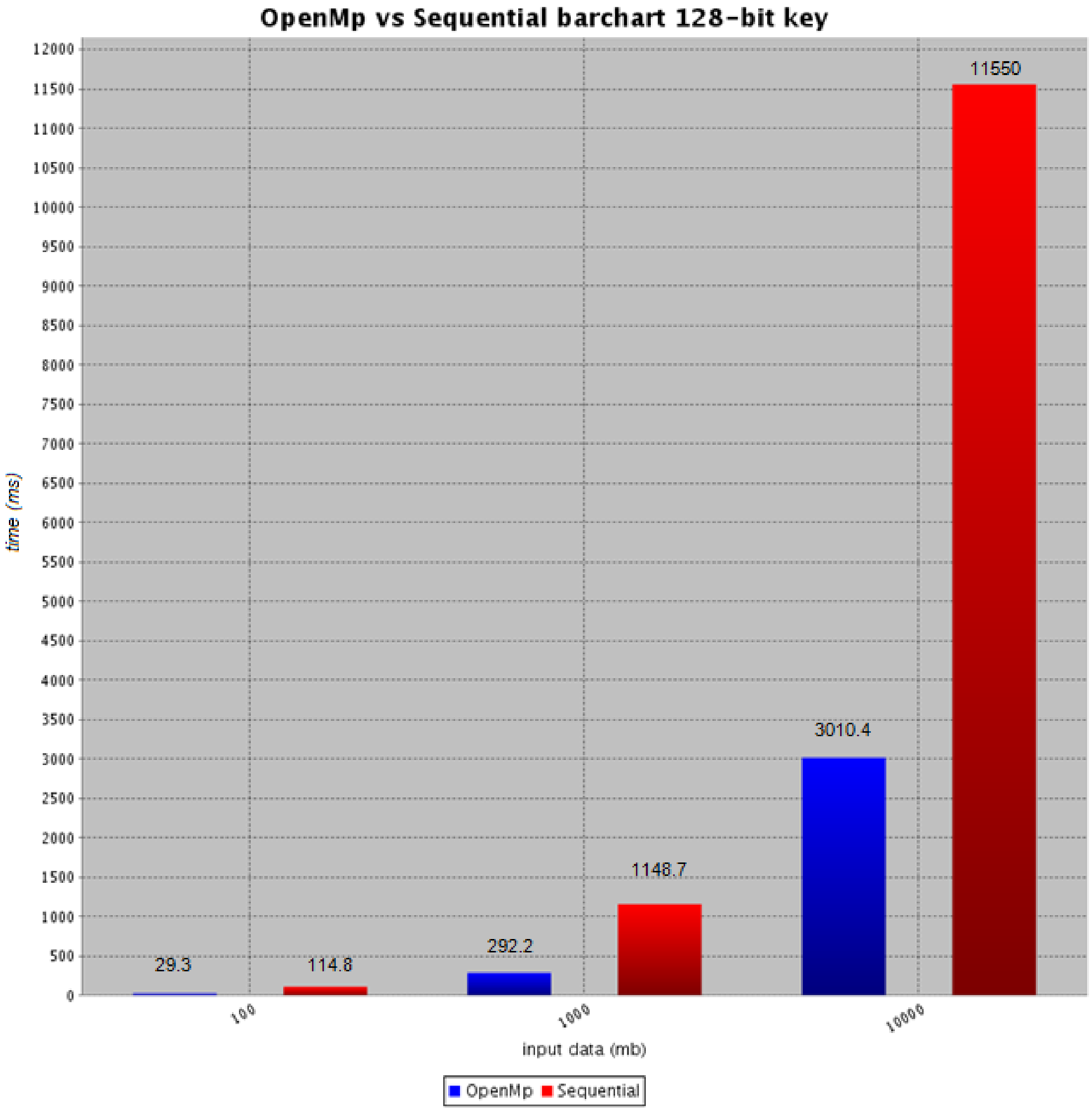
We can see that key size affects our 1GB test results in a very similar factor to the sequential executions. The 192-bit key takes 121% of the time versus the serial's 119% percentage, and the 256-bit key takes 141% for both serial and OpenMP.

The real question, however, is how does OpenMP code compare to the sequential code? These bar charts can paint the main picture:



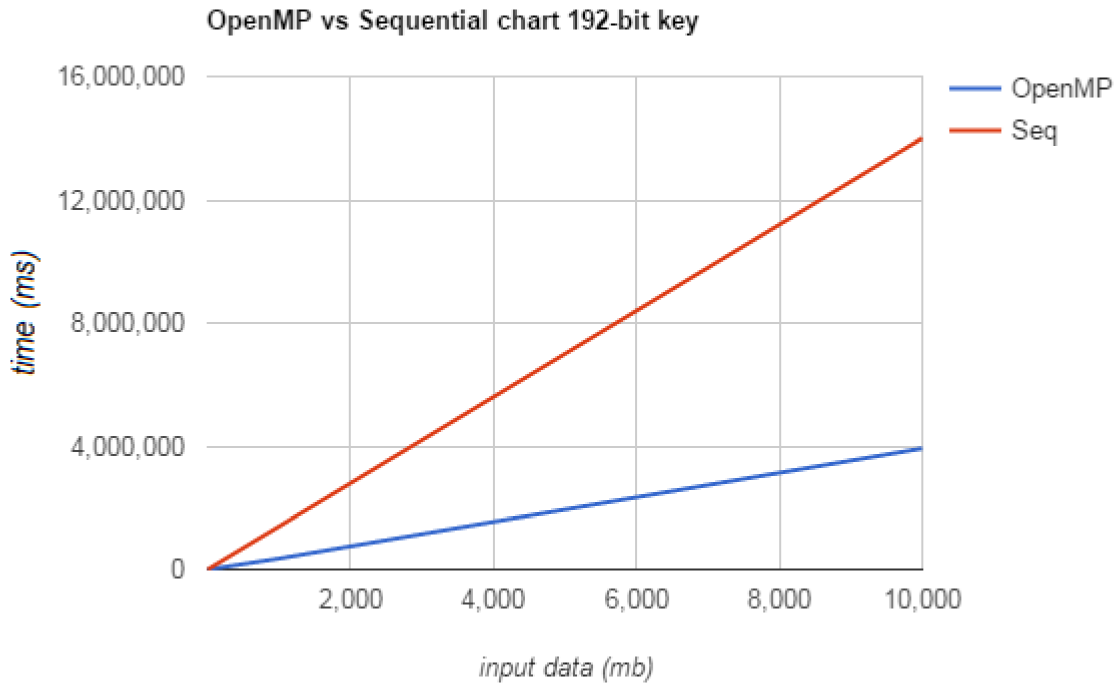
Graph 7: Sequential vs OpenMP 128-bit key chart

Optically the results seem satisfying. The chart above contains info about tests from 1mb to 10gb though and that is why we are going to provide an additional barchart for each key size to better depict our results as far as the 3 largest tests are concerned.

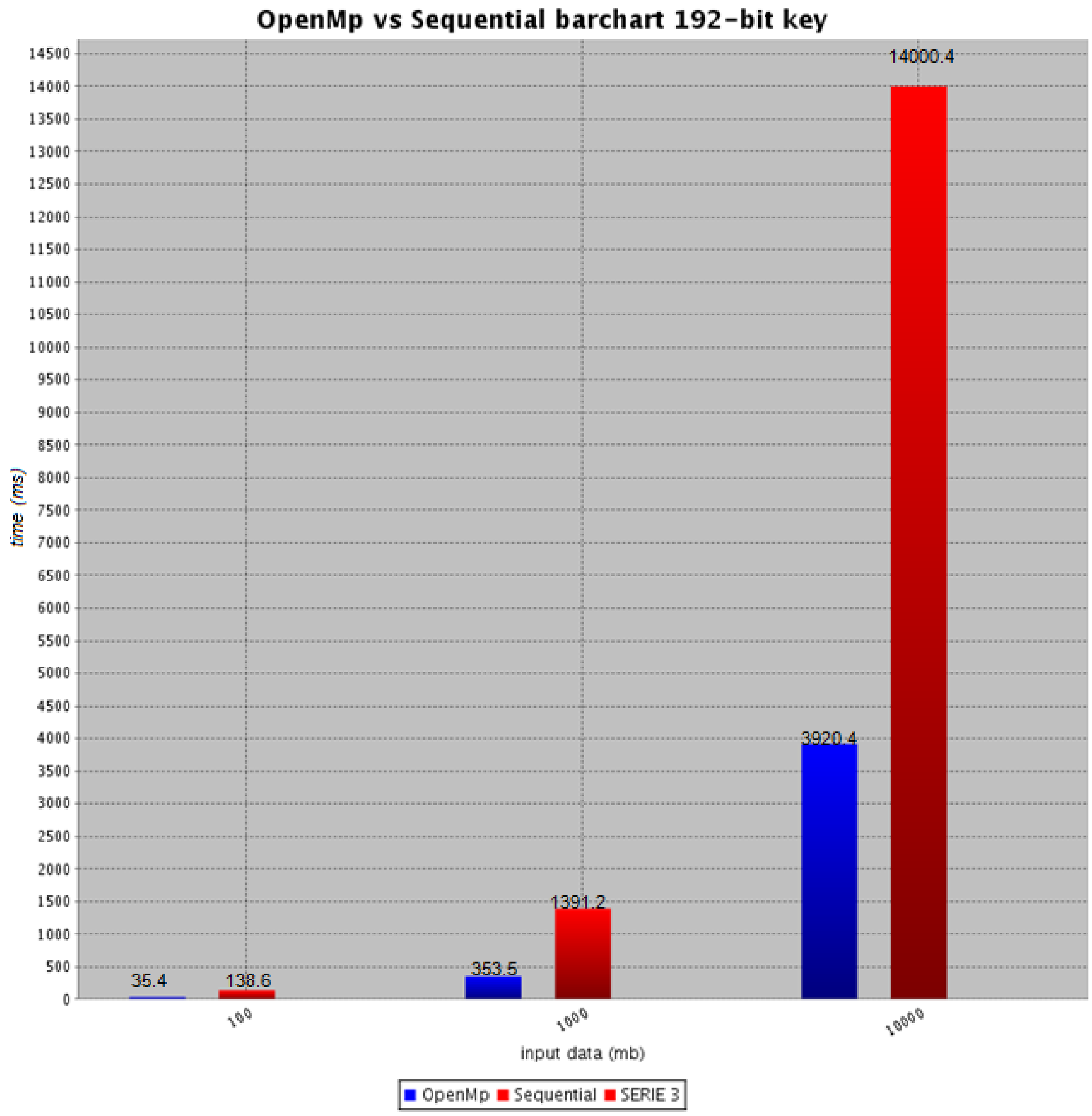


Graph 8: Sequential vs OpenMP 128-bit key barchart

In addition, the 192-bit key results are shown below:

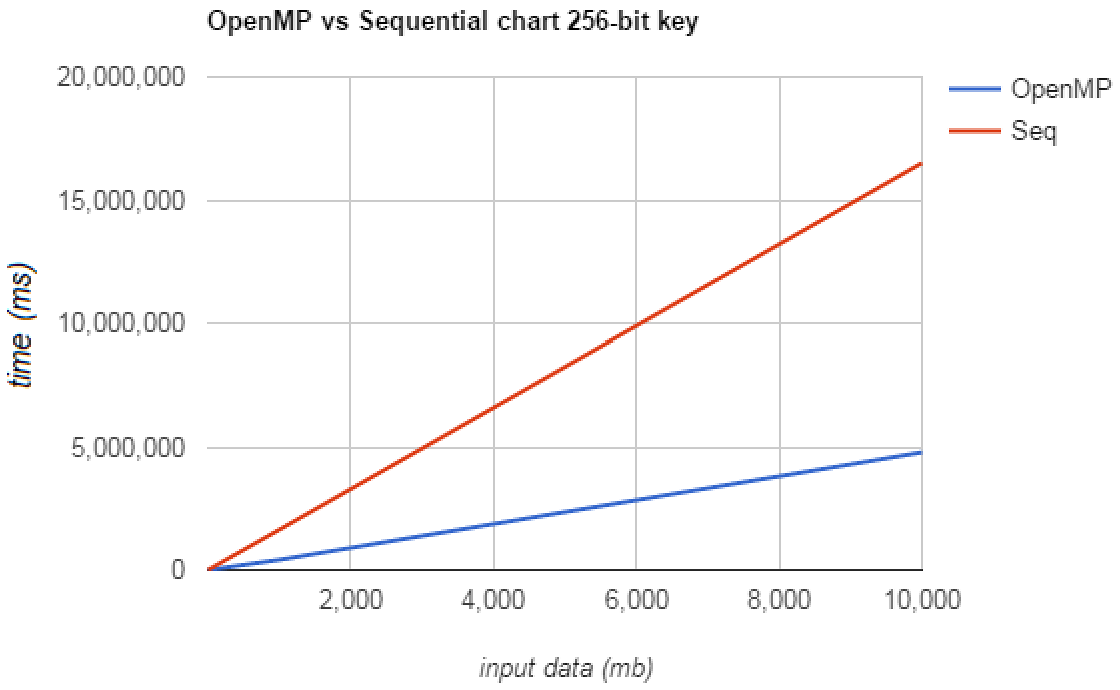


Graph 9: Sequential vs OpenMP 192-bit key chart

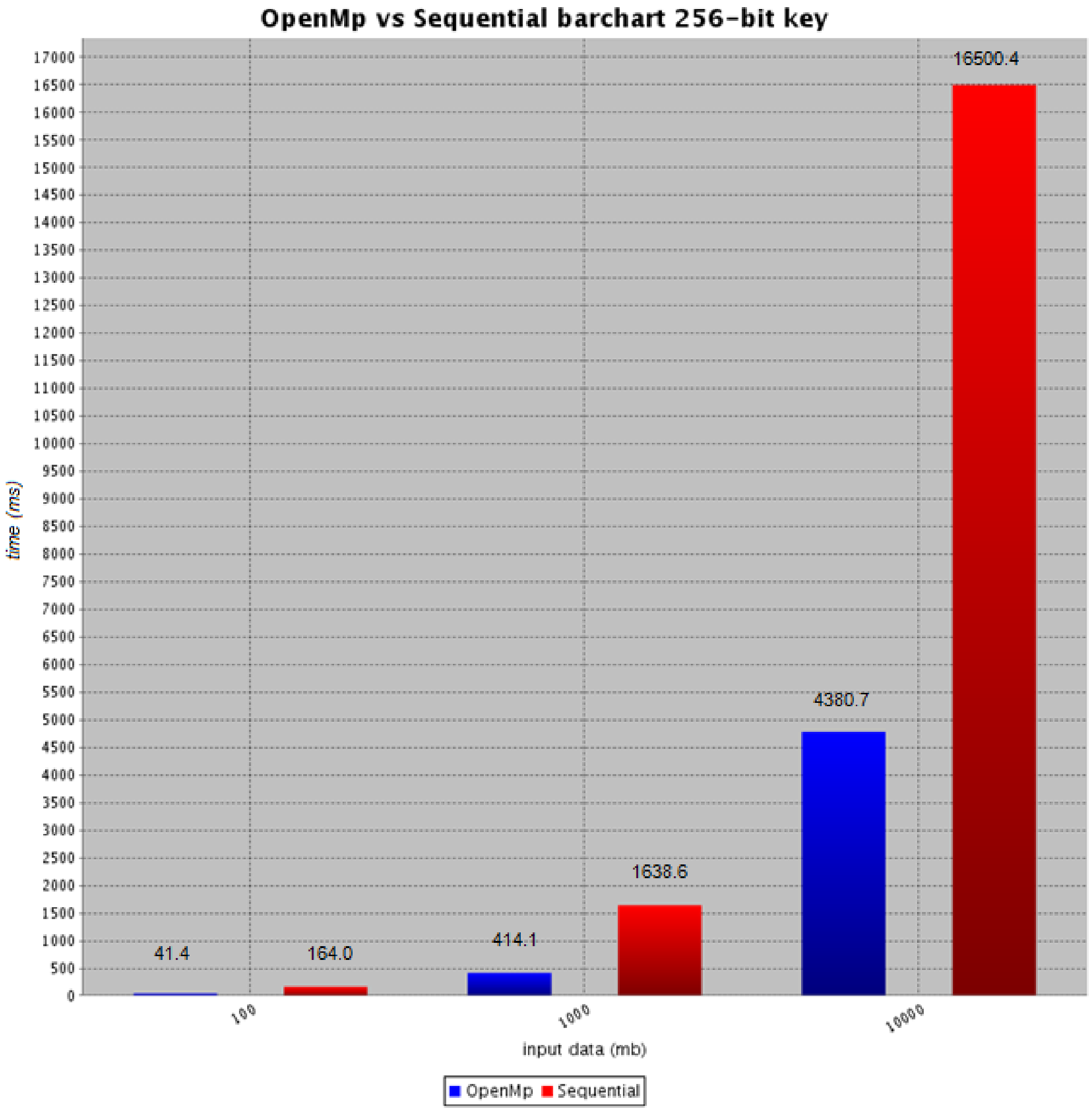


Graph 10: Sequential vs OpenMP 192-bit key barchart

Finally, the 256-bit key comparison is depicted in the following charts:



Graph 11: Sequential vs OpenMP 256-bit key chart



Graph 12: Sequential vs OpenMP 256-bit key barchart

Even though these bar charts can give the main impression of acceleration in the execution of the algorithm, our main concern here is the actual speedup in terms of percentages. But to better understand the concept of speedup in parallel computing we should take a brief look at Amdahl's law.

Amdahl's law **[29]** is a model for the expected speedup and the relationship between parallelized implementations of an algorithm and its sequential implementations, under the assumption that the problem size remains the same when parallelized. The main variables it involves can be shown in the equation below:

- n ϵ N, the number of threads of execution
- B ϵ [0,1], the fraction of the algorithm that is strictly serial

The time T(n) an algorithm takes to finish when being executed on n thread(s) of execution corresponds to:

$$T(N) = T(1)\left(B + \frac{1}{n}(1 - B)\right)$$

Therefore, the theoretical speedup S(n) that can be had by executing a given algorithm on a system capable of executing n threads of execution is:

$$S(n) = \frac{T(1)}{T(n)} = \frac{T(1)}{T(1)\left(B + \frac{1}{n}(1 - B)\right)} = \frac{1}{B + \frac{1}{n}(1 - B)}$$

So, theoretically, if a problem is 100% parallelizable (though it never is), and we have 4 available threads of execution (just like our test machine), then B=0, n=4 and S(n)=4. It is foolish to expect such a speedup though, since no algorithms are 100% parallelizable, and the AES ECB is no exception to that rule.

It is more convenient to use the S(n)=T(1)/T(n) equation for our results, since we already have the timings of the executions, whilst lacking the B parameter. Another way to calculate the speedup would be to divide the corresponding throughputs. So:

128-bit key:

Data size (MB)	OpenMP vs. Sequential (speedup)
1	3.65
10	3.89
100	3.89
1000	3.93
10000	3.93

Table 11: OpenMP vs Sequential 128-bit key speedup

192-bit key

Data size (MB)	OpenMP vs Sequential (speedup)
1	3.78
10	3.88
100	3.89
1000	3.94
10000	3.94

Table 12: OpenMP vs Sequential 192-bit key speedup

256-bit key

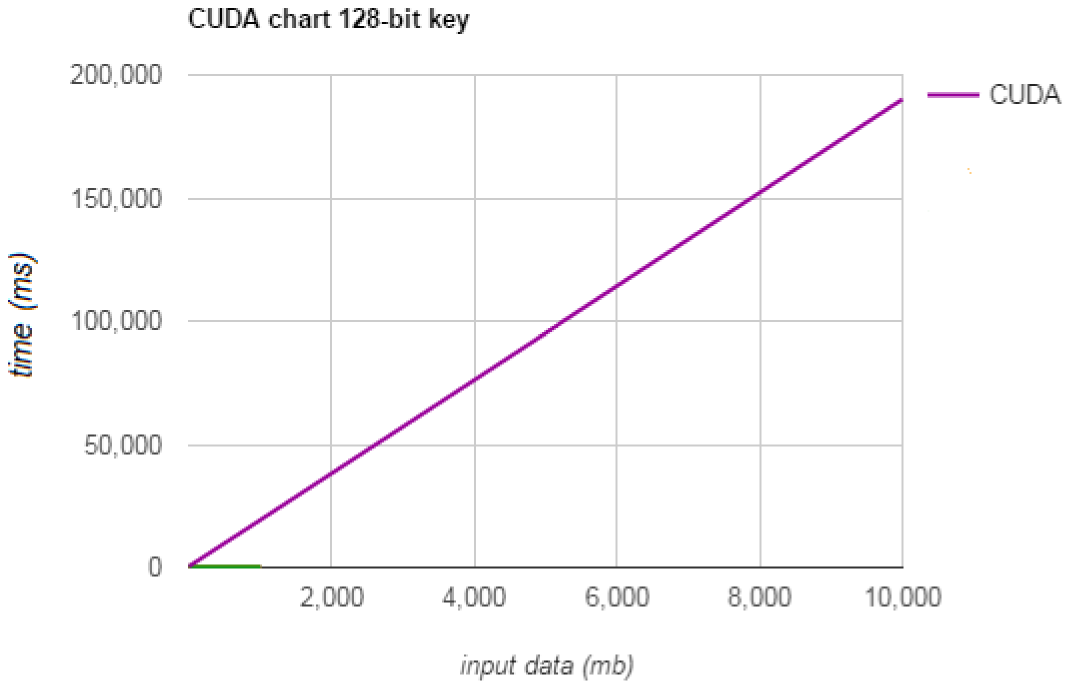
Data size (MB)	OpenMP vs Sequential (speedup)
1	3.77
10	3.95
100	3.96
1000	3.96
10000	3.96

Table 13: OpenMP vs Sequential 256-bit key speedup

As we can see, the speedup in each case is very close to 4 which would be the perfect speedup, if the algorithm was 100% parallelizable. That leads us to two conclusions. First, only a very small part of the algorithm is strictly sequential, and second, parallel programming is working very well in our case with OpenMP threads.

5.5.3 CUDA Results

Finally, we present the core of our testing procedure, which is the presentation of the results of the CUDA approach to the AES algorithm.



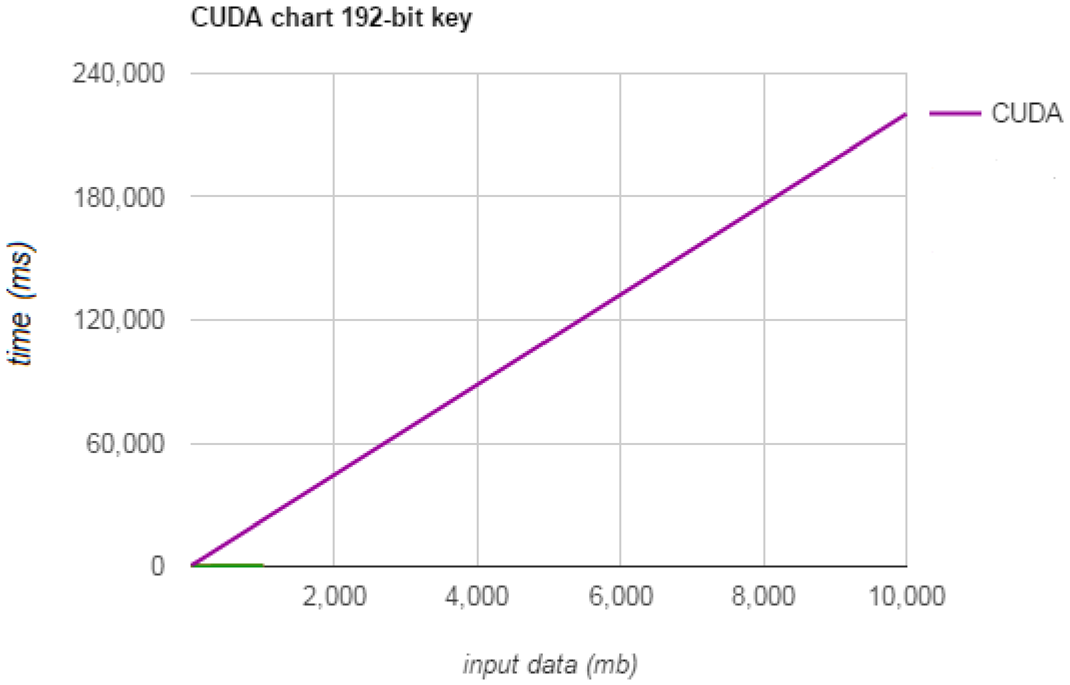
Graph 13: CUDA 128-bit key chart

The deriving throughput table is the following:

Size (MB)	1	5	10	50	100	1000	10000
Throughput (MB/s)	2.1	9.1	15.6	35.7	43.5	52	52.6

Table 14: CUDA throughput 128-bit key

CUDA 192-bit key results:



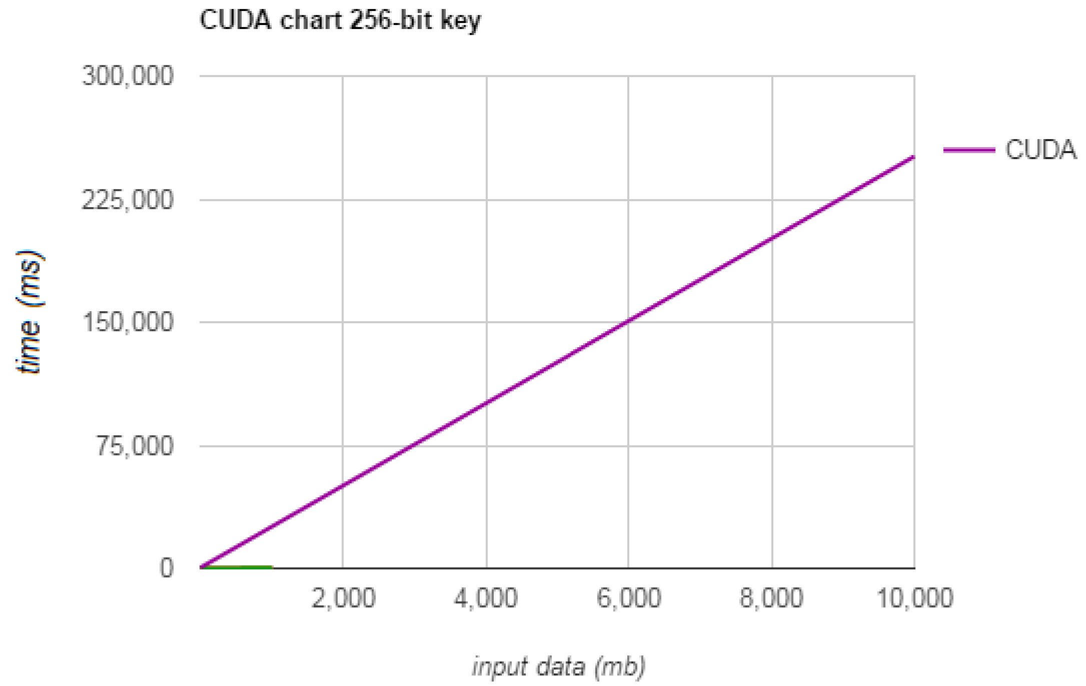
Graph 14: CUDA 192-bit key chart

Again, we present the average throughput on each of our tests:

Size (MB)	1	5	10	50	100	1000	10000
Throughput (MB/s)	2	9.1	12.5	33.3	38.4	44.6	45.4

Table 15: CUDA throughput 192-bit key

CUDA 256-bit key results:



Graph 15: CUDA 256-bit key chart

The final throughput table:

Size (MB)	1	5	10	50	100	1000	10000
Throughput (MB/s)	2.1	8.3	11.3	29.4	34.5	40	40

Table 16: CUDA throughput 256-bit key

Naturally, key size also affects the CUDA implementation. Below we present the table with the comparison of the timings of 1gb data input, using all 3 key sizes.

Key size	Performance compared to 128-bit key execution time
128-bit	100%
192-bit	116%
256-bit	132%

Table 17: CUDA key size-performance

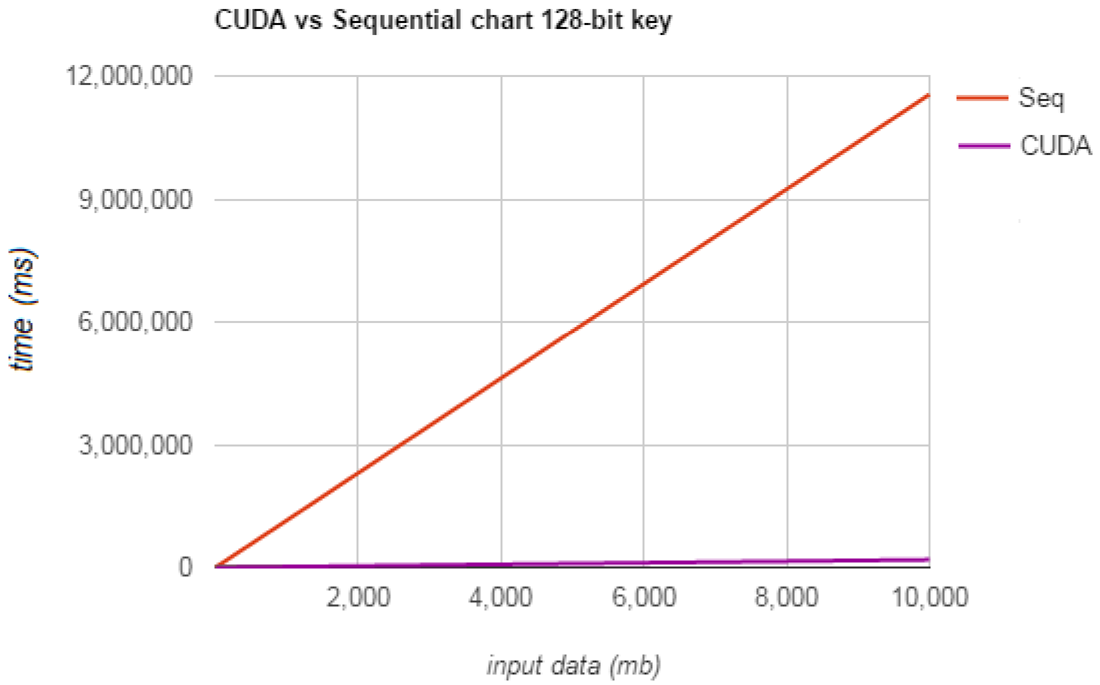
What is really interesting is that the key size affects are results but to a significantly minor extent in comparison with the sequential and OpenMP approaches, which were in fact quite the same. The final and complete table of 1gb input is the following:

	Sequential	OpenMP	CUDA
128-bit	100%	100%	100%
192-bit	119%	121%	116%
256-bit	141%	141%	132%

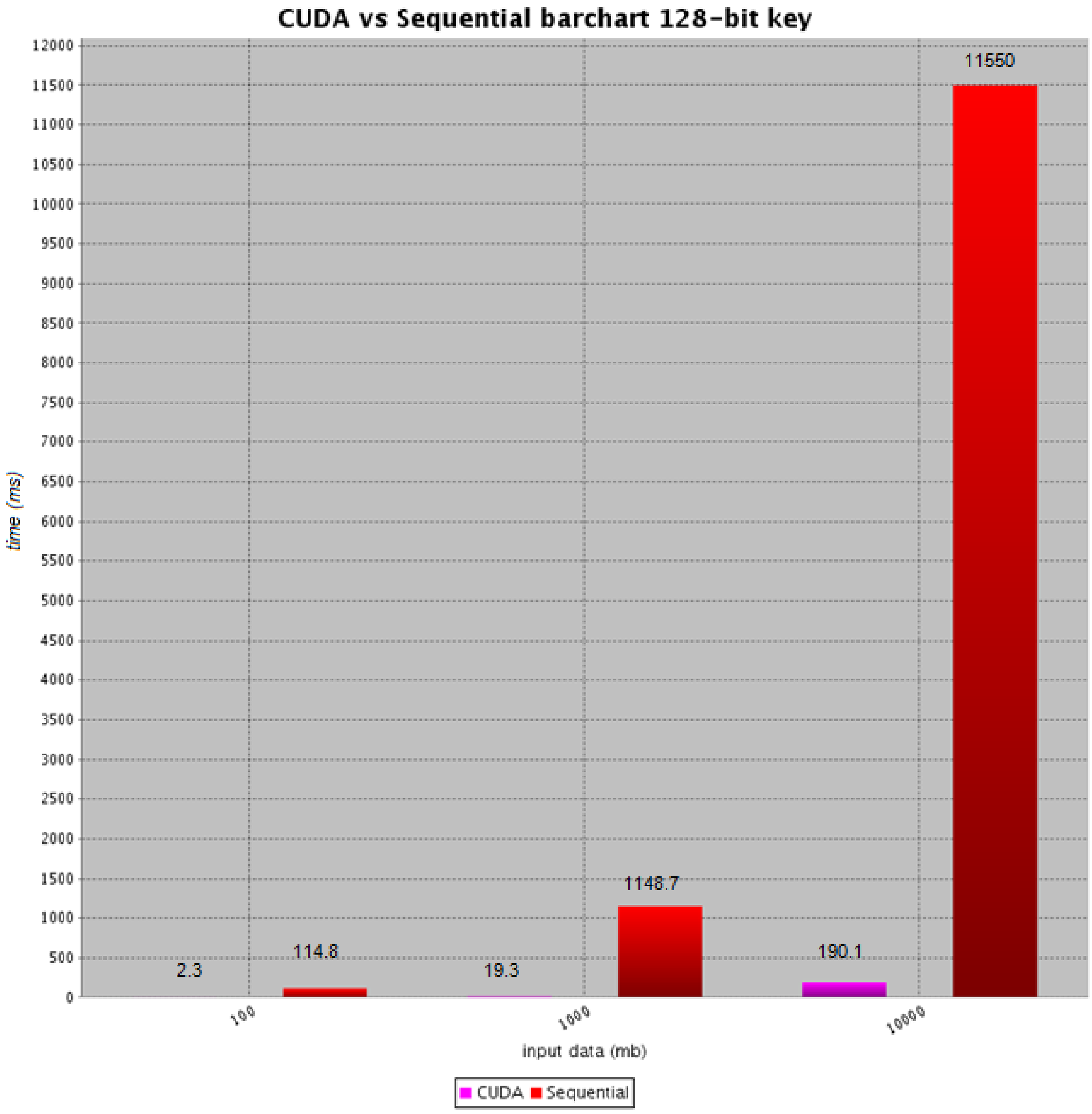
Table 18: All implementations key size-performance

Pure numbers don’t tell the whole story, though. Our main purpose is to compare CUDA to Sequential and OpenMP code. First let’s take a look at the bar charts.

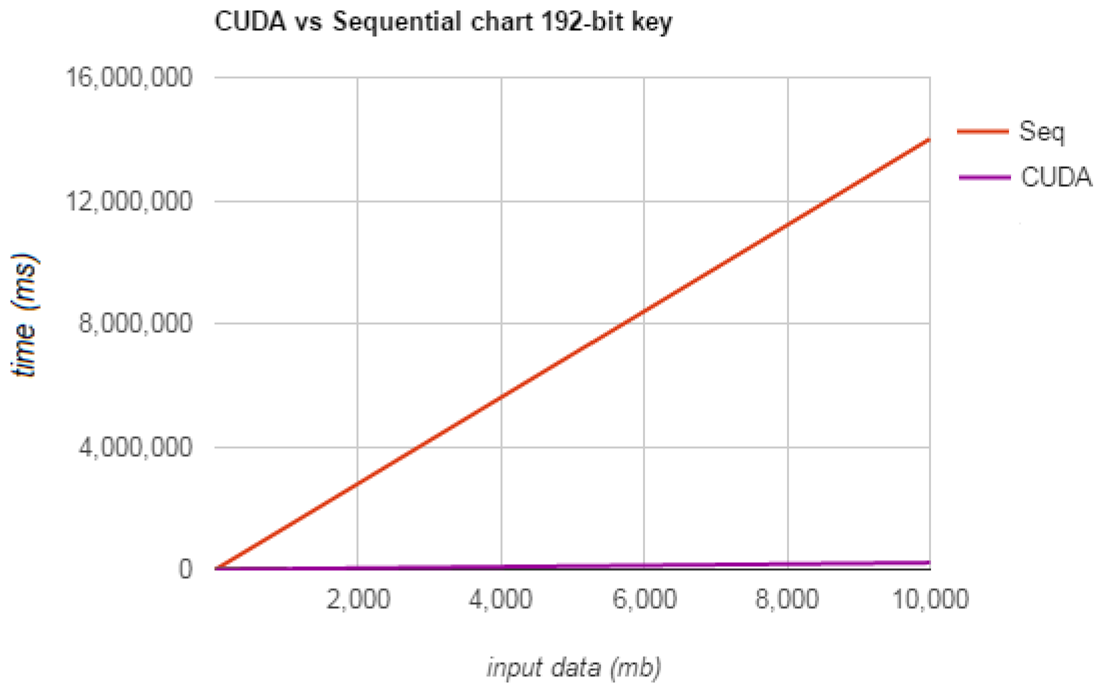
CUDA vs Sequential code



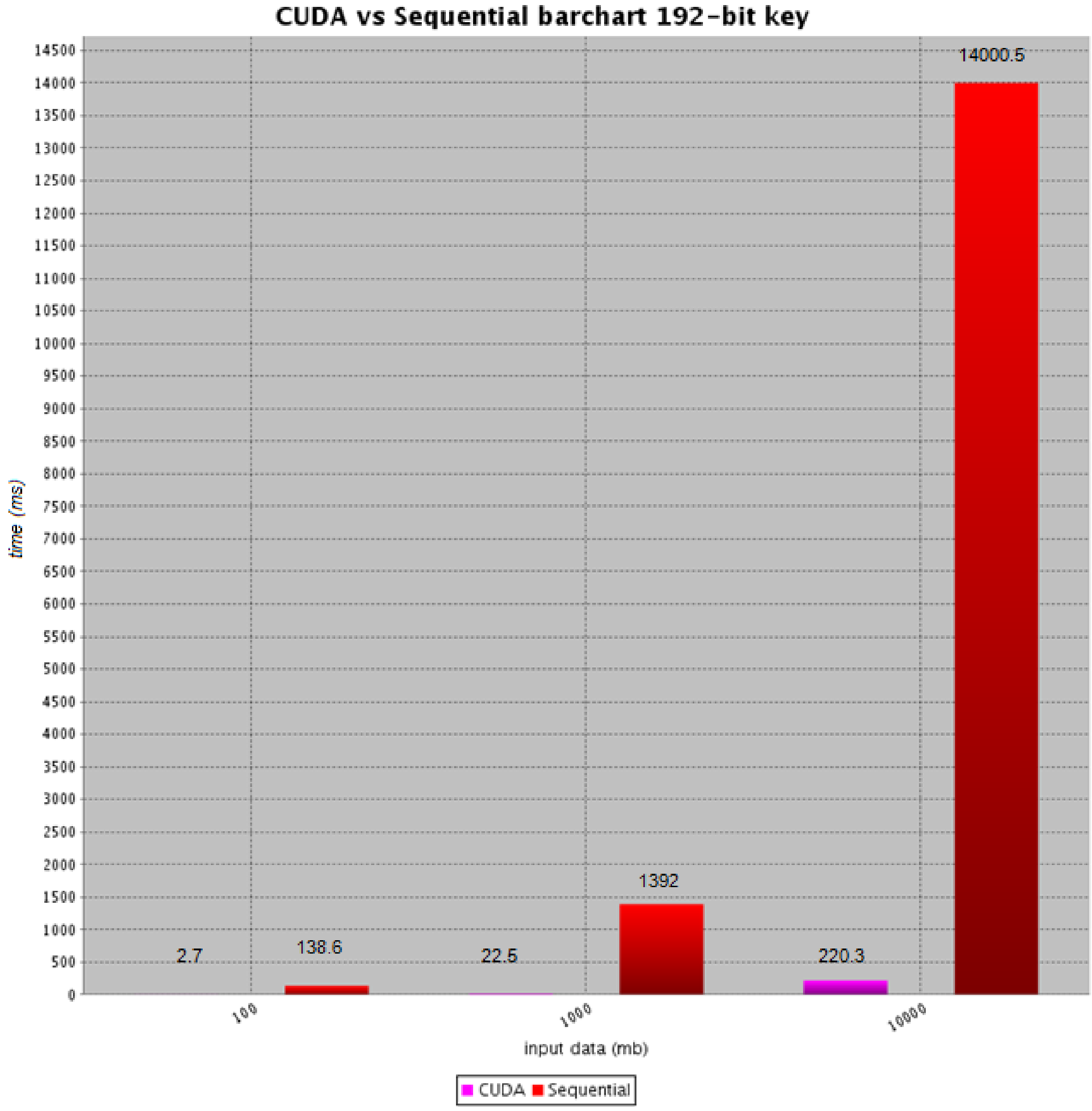
Graph 16: CUDA vs Sequential 128-bit key chart



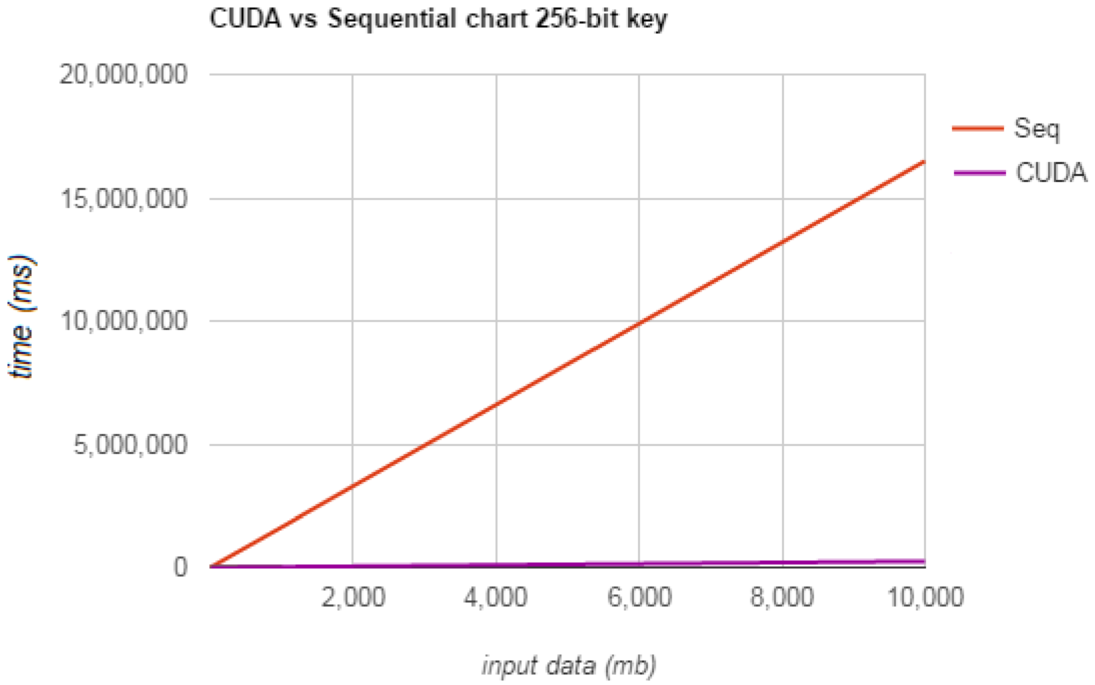
Graph 17: CUDA vs Sequential 128-bit key barchart



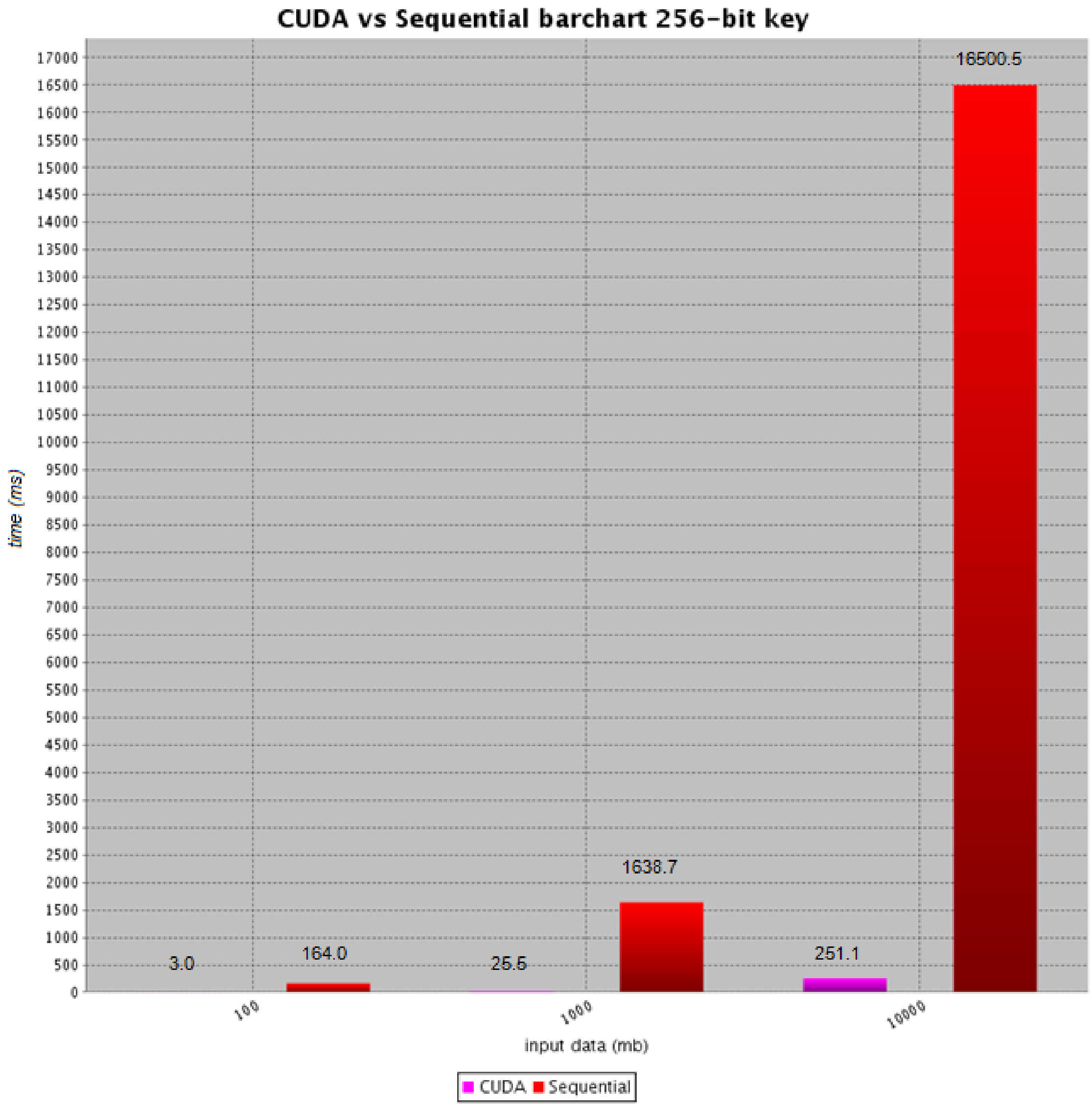
Graph 18: CUDA vs Sequential 192-bit key chart



Graph 19: CUDA vs Sequential 192-bit key barchart

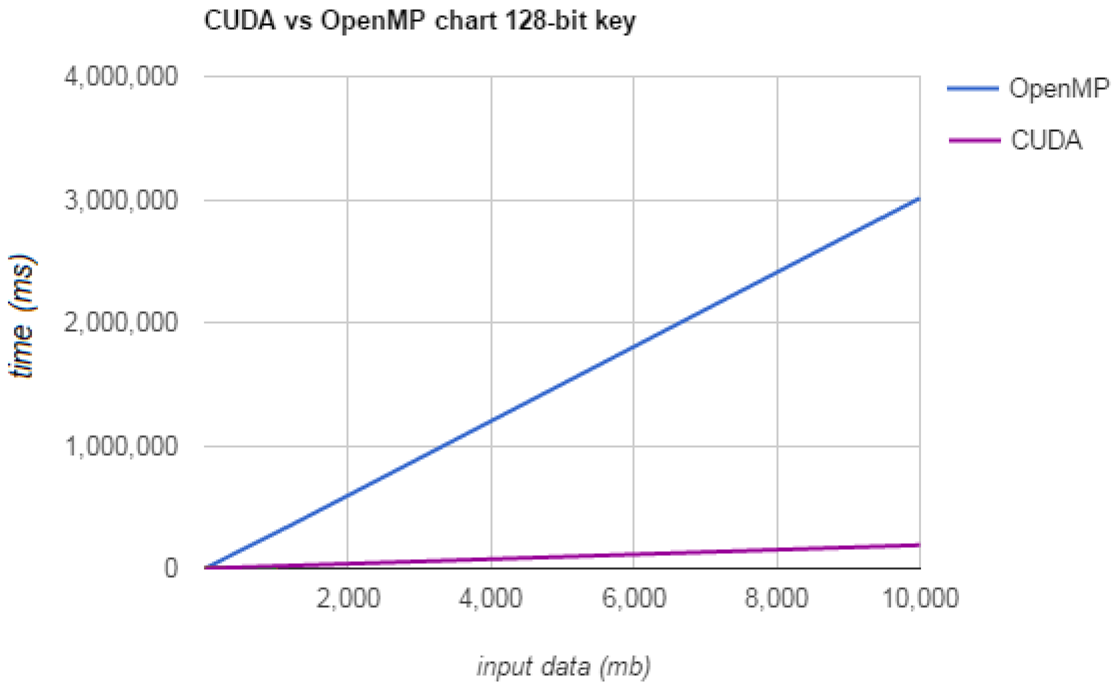


Graph 20: CUDA vs Sequential 256-bit key chart

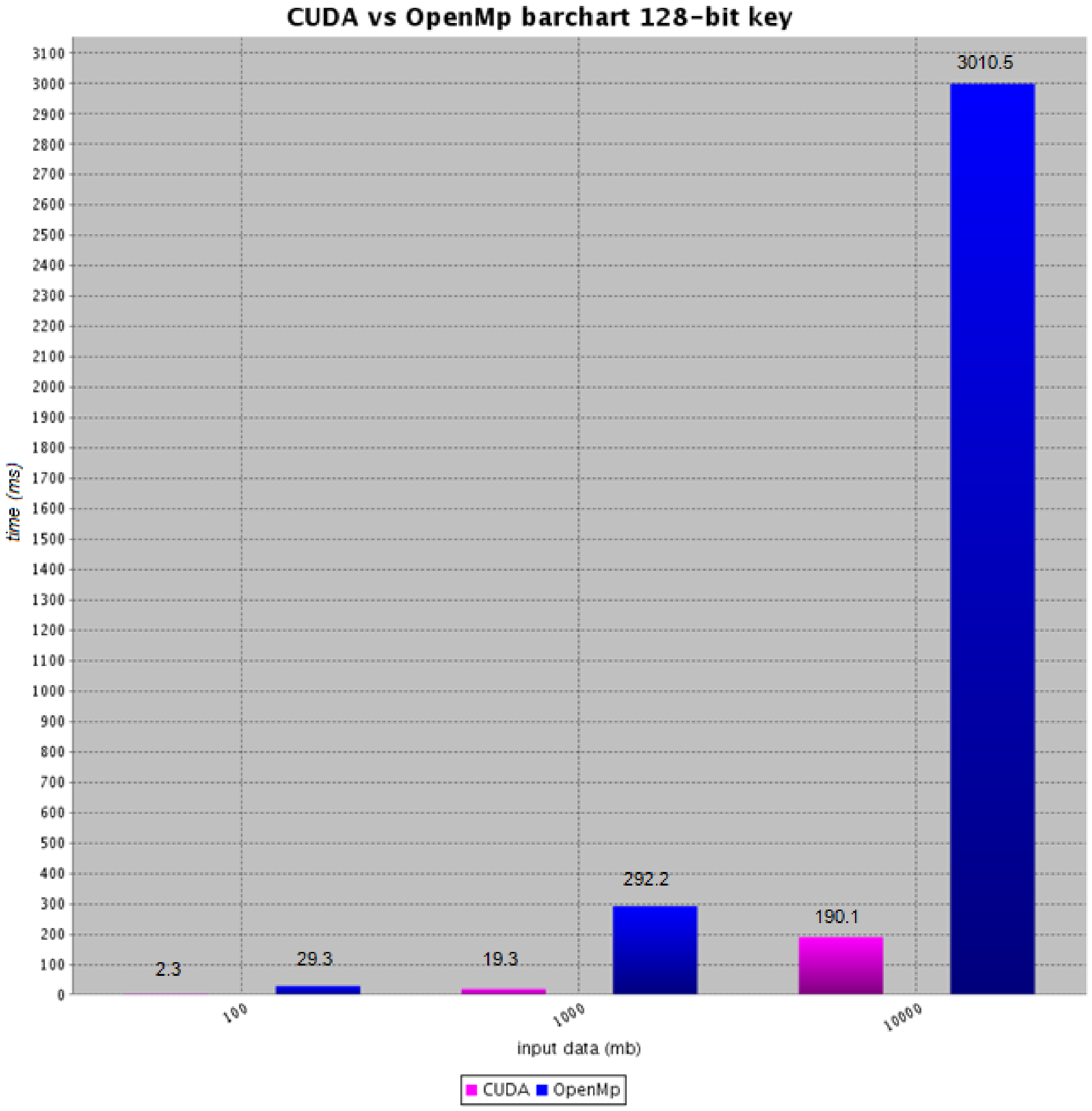


Graph 21: CUDA vs Sequential 256-bit key barchart

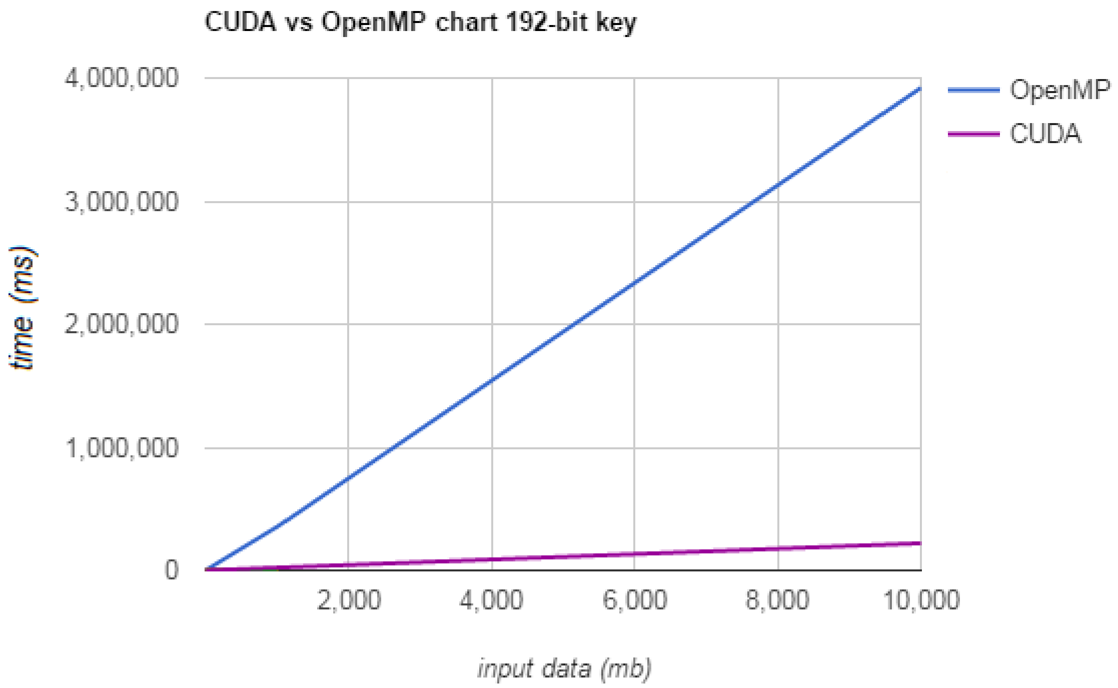
CUDA vs OpenMP



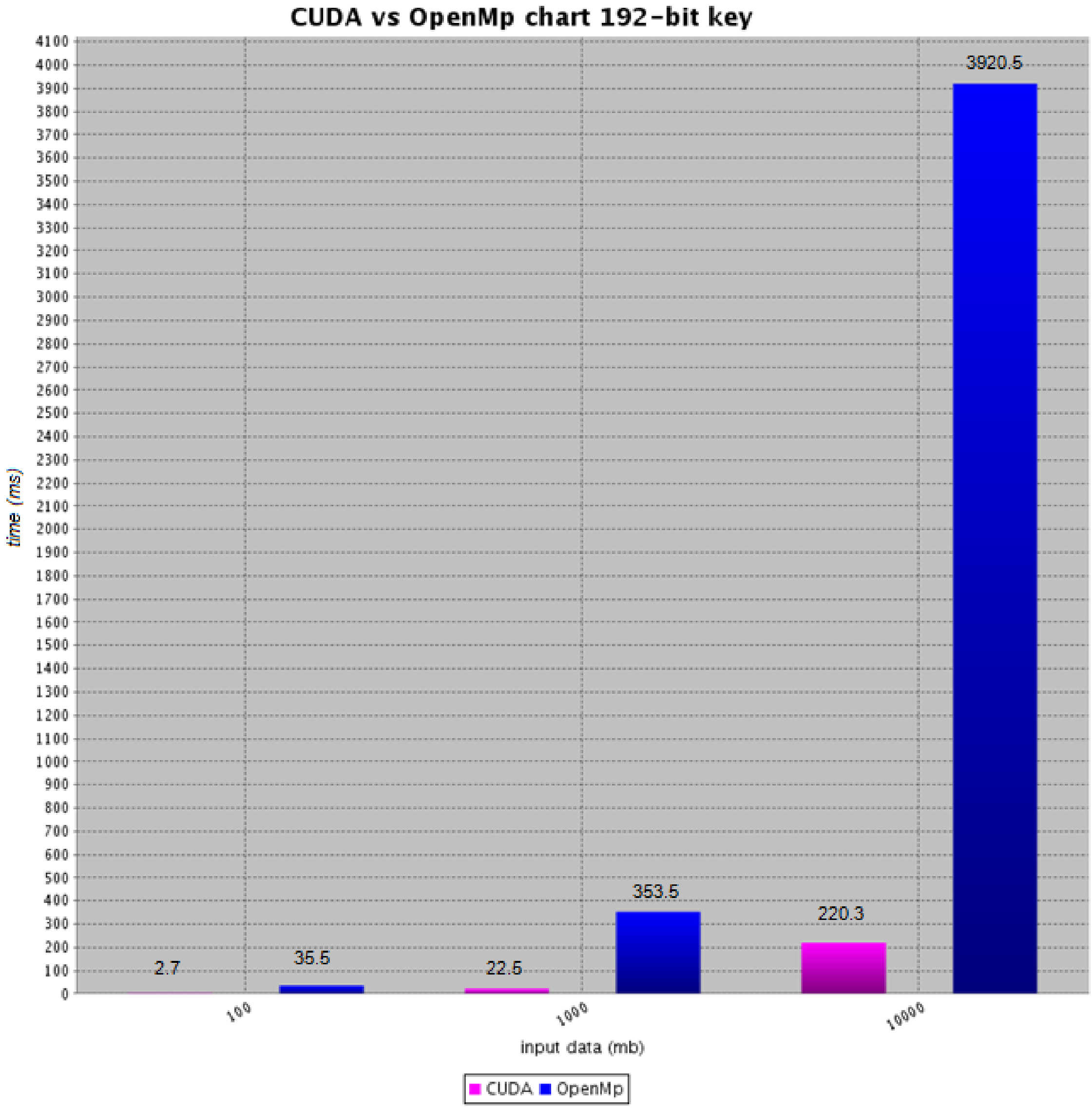
Graph 22: CUDA vs OpenMP 128-bit key chart



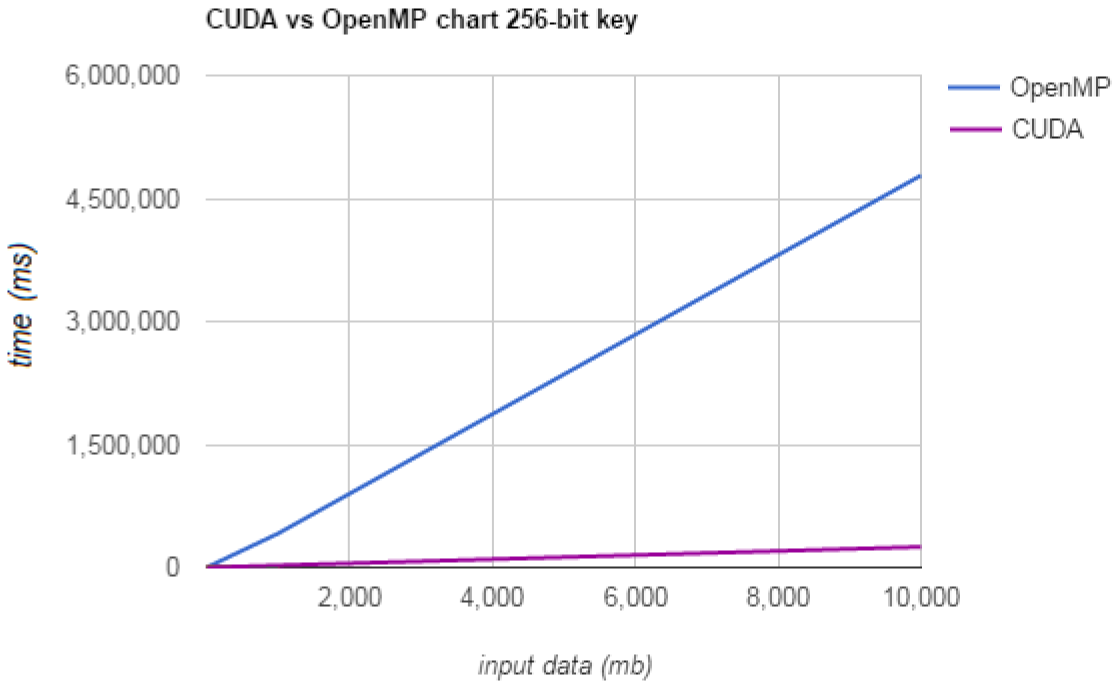
Graph 23: CUDA vs OpenMP 128-bit key barchart



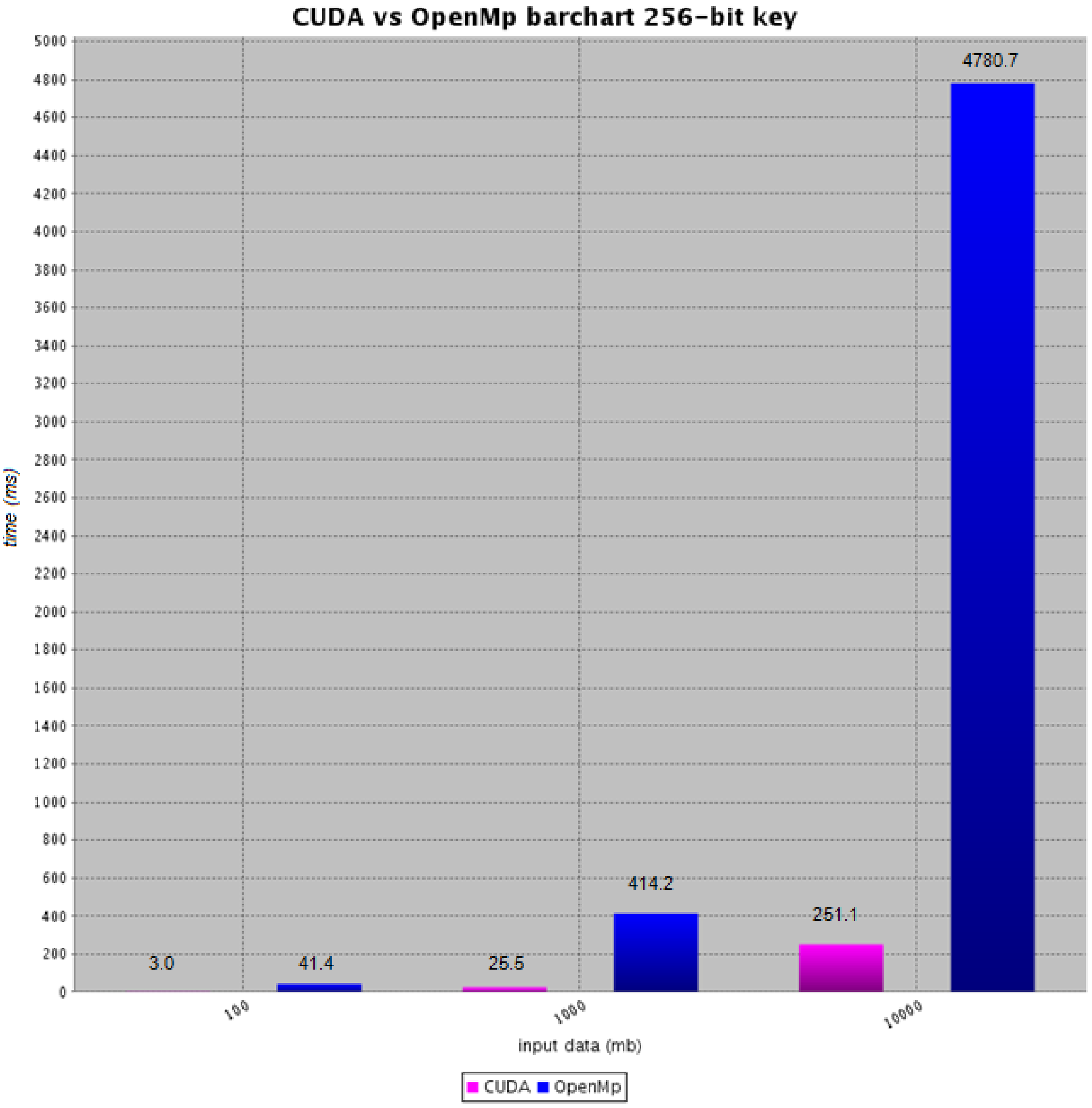
Graph 24: CUDA vs OpenMP 192-bit key chart



Graph 25: CUDA vs OpenMP 192-bit key barchart



Graph 26: CUDA vs OpenMP 256-bit key chart



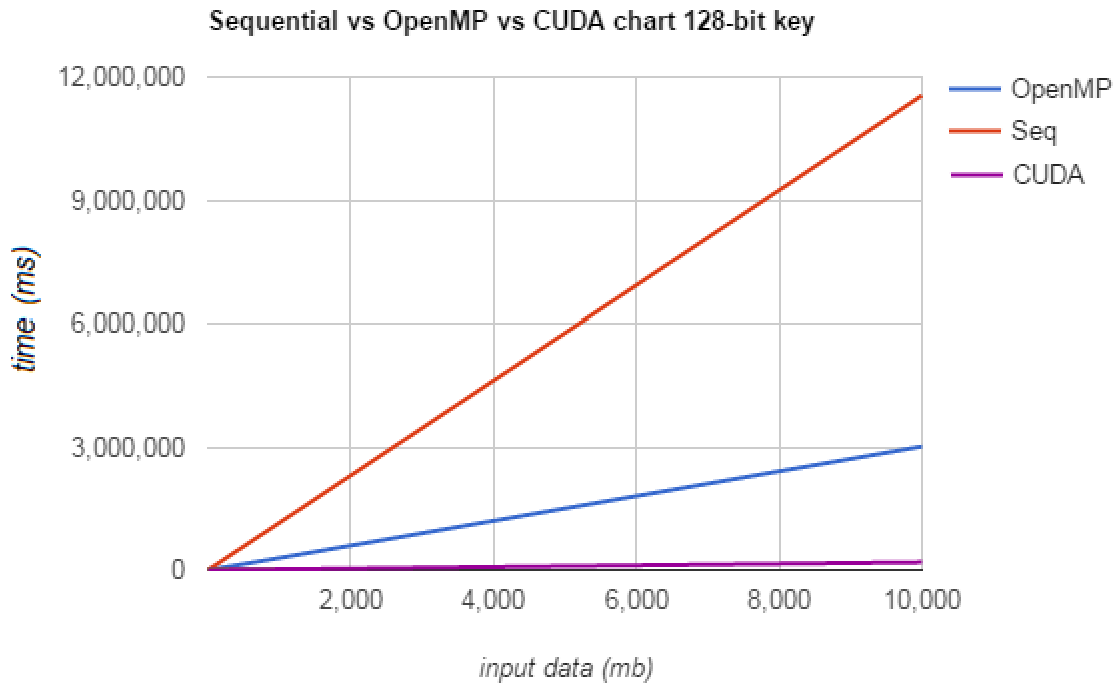
Graph 27: CUDA vs OpenMP 256-bit key barchart

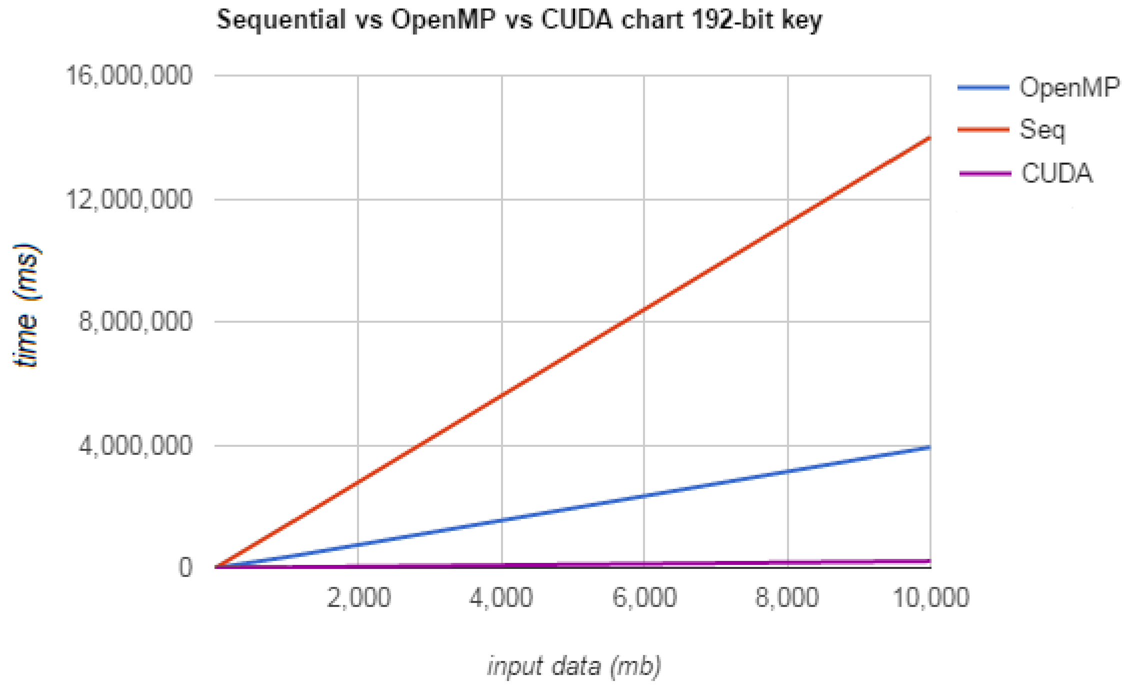
Needless to say the diagrams above show clearly in an optical way that CUDA is able to run the algorithm in parallel in a much faster pace, and to be precise, handling much more throughput for the same time. The results are indeed impressive and verify our initial expectations about CUDA code and its performance boost potential.

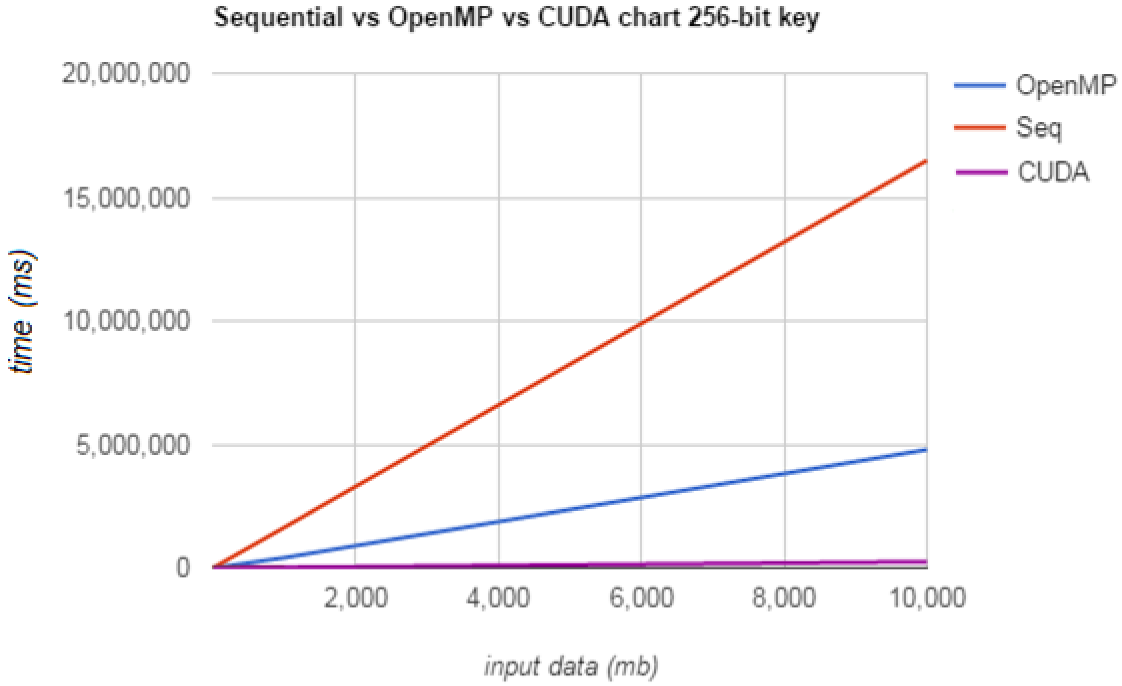
Before we present the speedup results, we should clarify that just because our test machine and in particular the GPU can run 448 threads in parallel, it is wrong to assume that the perfect speedup would be 448 (again, in case of a 100% parallelizable algorithm). This assumption would be true only if the CPU threads on our AMD processor were exactly the same in all aspects, which is very far from true. GPU and CPU threads have many differences that start from architecture reasons to many other reasons we referred to many times in this thesis.

Given the previous charts, it is more than obvious that the speedup is going to be huge, especially compared to the sequential code, but also in comparison to OpenMp.

Sequential vs OpenMP vs CUDA






Sequential vs OpenMP vs CUDA chart 256-bit key
20,000,000
15,000,000
10,000,000
5,000,000
0
time (ms)
2,000
4,000
6,000
8,000
10,000
input data (mb)
OpenMP
Seq
CUDA

CUDA Speedup

CUDA speedup vs Sequential

128-bit key

Data size (MB)	CUDA vs Sequential (speedup)
1	2.43
10	18
100	49.3
1000	59.5
10000	60.5

Table 19: CUDA vs Sequential 128-bit key speedup

192-bit key

Data size (MB)	CUDA vs Sequential (speedup)
1	2.84
10	17.3
100	53.3
1000	62
10000	63.6

Table 20: CUDA vs Sequential 192-bit key speedup

256-bit key

Data size (MB)	CUDA vs Sequential (speedup)
1	3.5
10	23.1
100	54.6
1000	65.5
10000	65.7

Table 21: CUDA vs Sequential 256-bit key speedup

At first glance, we notice three things. First, the speedup is increasing significantly when using larger data sets as an input, and secondly, the larger the key, the higher the speedup given the same input data. It is clear that using CUDA in the AES ECB algorithm makes perfect sense in most cases performance-wise, but the stats show that CUDA is ultra-efficient when using bigger keys and large input sizes.

Last but not least, since the speedup is lower for smaller inputs, one could wonder if there is a threshold to that speedup at some point. There comes the question whether the speedup is always >1 (and therefore it makes sense to use CUDA over serial code), or in small inputs the sequential runs faster. The answer unsurprisingly is that on small input data, the serial code could run faster. The real question, is how small.

As a matter of fact, we found out that encryption runs faster on serial code if the input ranges from some bytes to several kilobytes. But that changes drastically once we pick larger inputs for encryption/decryption. Even on our smallest data set (1mb) using the smallest possible key (128-bit), the least speedup is 2.46 which is too much to neglect. Whether the same point stands in a CUDA vs OpenMP comparison, is something we will examine right away.

CUDA speedup vs OpenMP

128-bit key

Data size (in mb)	CUDA vs OpenMP (speedup)
1	0.67
10	4.4
100	12.6
1000	15.2
10000	15.8

Table 22: CUDA vs OpenMP 128-bit key speedup

192-bit key

Data size (in mb)	CUDA vs OpenMP (speedup)
1	0.76
10	4.6
100	13.4
1000	15.7
10000	17.8

Table 23: CUDA vs OpenMP 192-bit key speedup

256-bit key

Data size (in mb)	CUDA vs OpenMP (speedup)
1	0.94
10	5.86
100	14
1000	16.2
10000	19

Table 24: CUDA vs OpenMP 256-bit key speedup

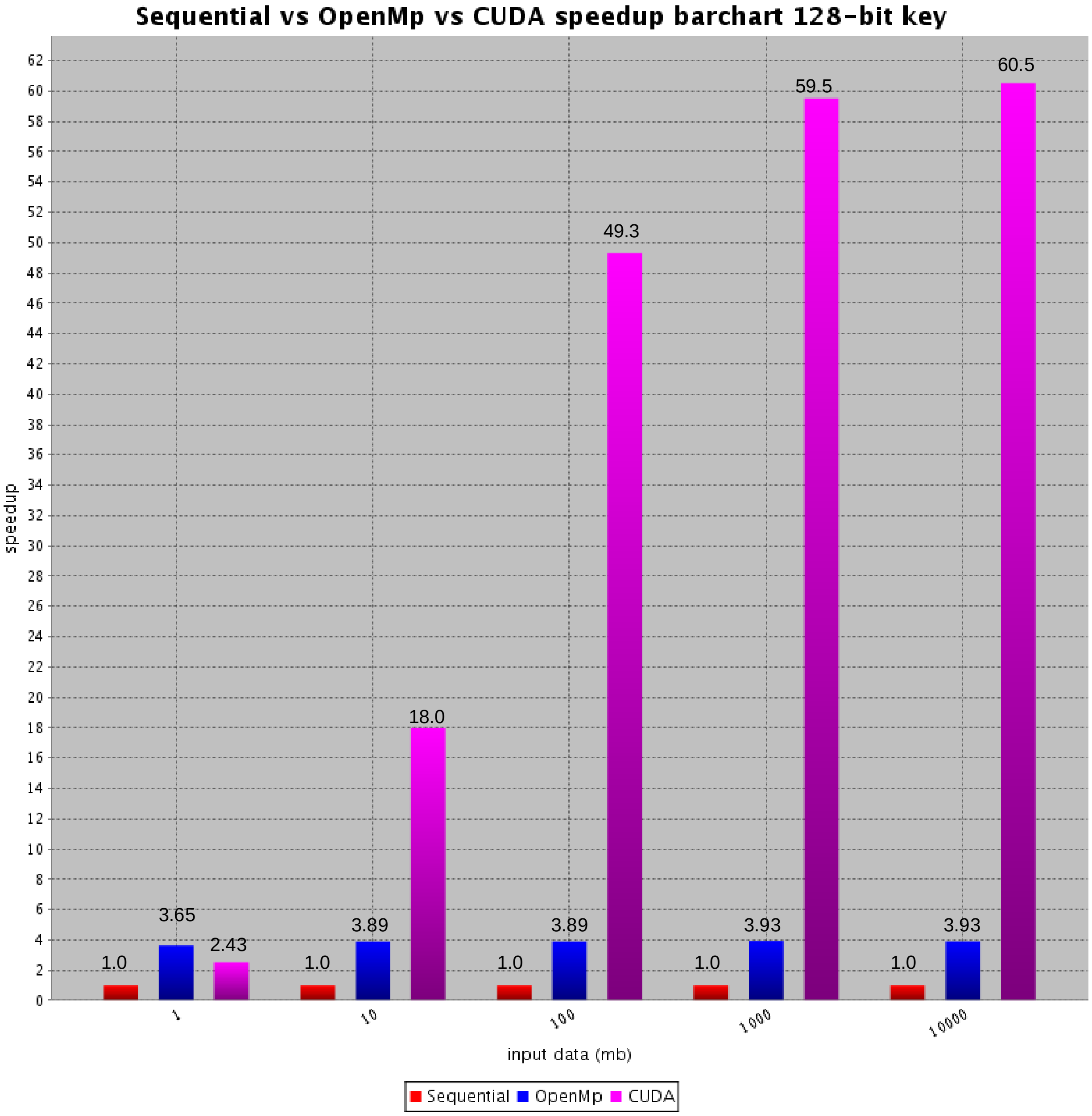
Sure, the results are not as impressive as on the previous comparison, but first of all that was to be expected since OpenMP is still a parallel implementation which means its faster than the sequential one, and secondly some of these numbers still remain very impressive performance-wise.

What stands out here, is the fact that OpenMP is faster than CUDA on our first sample input of 1MB, on all key sizes. That leads us to the conclusion that for small inputs of several MBs it is more efficient to use CPU parallelism instead of CUDA. That fact only stands for these cases, however. We can see that CUDA beats OpenMP by far in our second (10MB) test input, and the difference is getting even wider when the input sizes get even larger. In addition, the observation we made on the CUDA vs Sequential comparison still stands. CUDA is faster on both larger data sets and bigger key sizes, even against OpenMP.

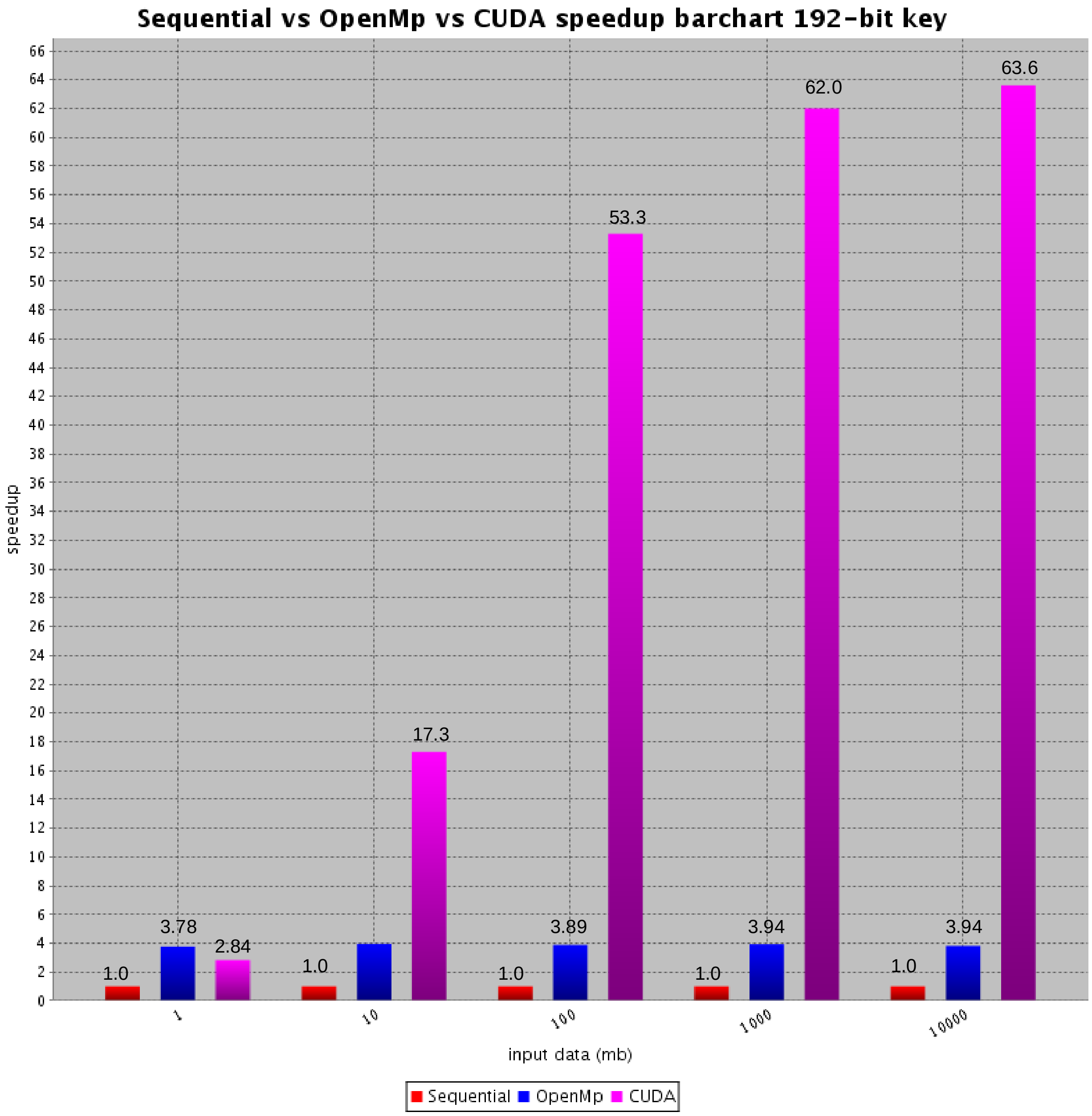
To summarize, we proved the superiority in performance that CUDA shows over both sequential C code and OpenMP CPU-parallel implementations, throughout a series of tests using different input and key sizes. It may now be clear that it is not always the best option, referring to small input sizes, but when the input reaches sizes over 5-7MB the performance gains are massive. To simplify, CUDA may not be the most appropriate solution in cryptography for text format messages in an Instant Messaging application (which should have a maximum size of several KBs), but it surely excels in terms of performance in file cryptography.

5.6 Speedup Tables

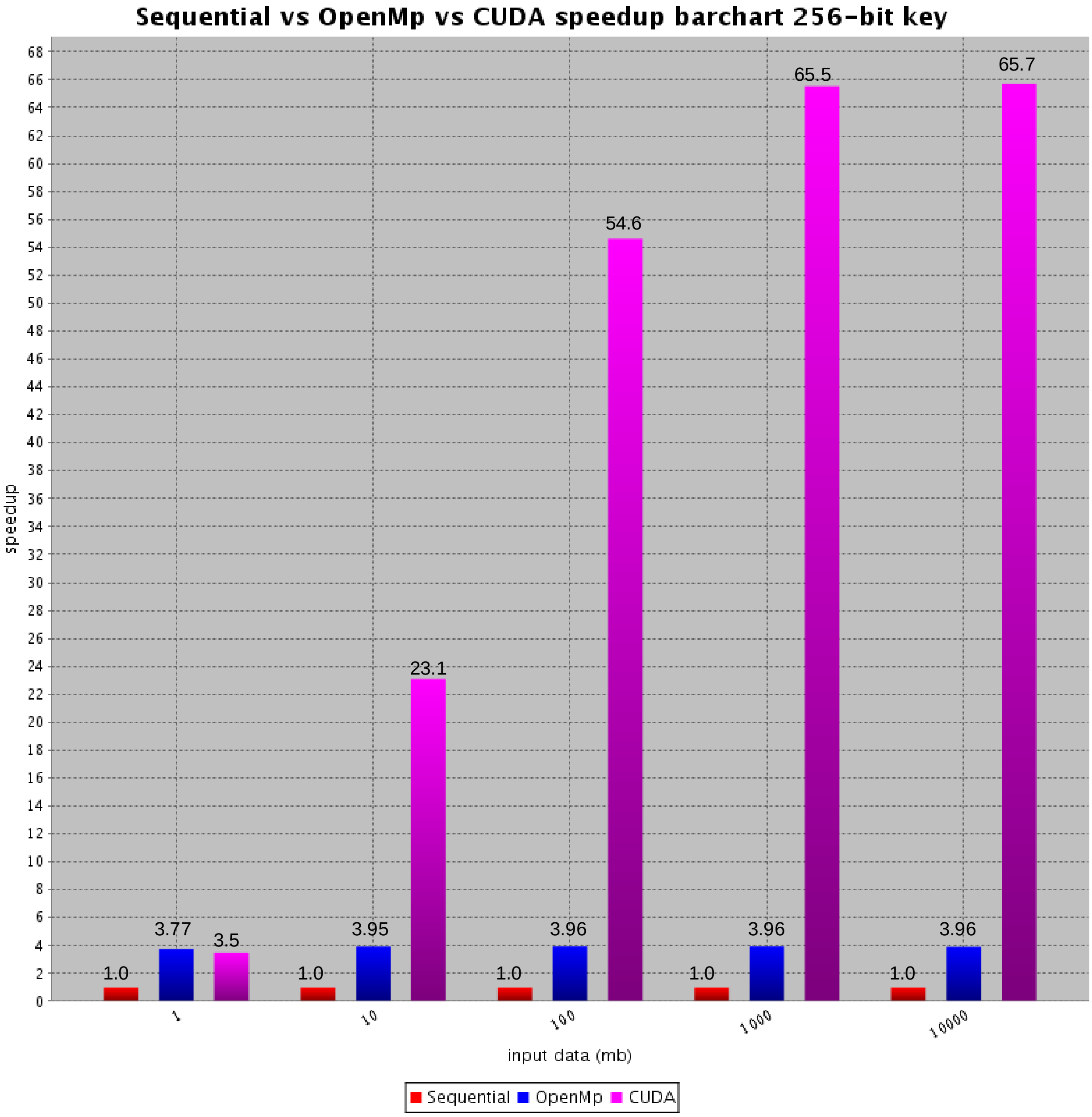
Below we present the final speedup tables that show the results of our testing as a whole in a graphical way. These bar charts paint the bigger picture and summarize the point of this thesis.



Graph 28: Sequential vs OpenMP vs CUDA 128-bit key speedup



Graph 29: Sequential vs OpenMP vs CUDA 192-bit key speedup



Graph 30: Sequential vs OpenMP vs CUDA 256-bit key speedup

6. TABLE OF RESULTS

Lastly, we present the tables with all timing values of the sequential, OpenMP and CUDA implementations, which helped to draw the results we discussed on this chapter and create the diagrams, charts and secondary tables we used to better present and further explain our results. Some of the values we presented in our charts were slightly approximated in order to provide rounder percentages.

These tables were filled with the values we drew from executing the *test_performance* executable on each implementation, using different data sizes and different keys. As far as the sequential implementation is concerned, the 10GB input took over an hour to complete and, while we have the actual timings in milliseconds (and of course used it in the charts and speedup calculations), we didn't include it. Furthermore it is natural to expect a timing which is around 10x the time of the 1gb timing.

What really stands out in the CUDA parts of these tables is the initialization time. A brief description of this stage of the algorithm would be that it includes reading the command line arguments and setting up the program accordingly (such as block cipher mode, key size and input data), initialization of variables such as the key and ctx structs, as well as block buffers to be used, plus one more costly procedure. This procedure is the CUDA initialization (driver initialization and CUDA context creation) and is a necessary overhead for the CUDA code to run. According to NVIDIA's forums, there are compilation modes that are supposed to instruct the CUDA driver to behave in such a way that no such initialization is needed every time a piece of CUDA code runs, but we believed that this overhead should be included in our tests anyway. To be precise, in a 475ms initialization time, this very delay would take up to 465ms.

This process happens once, in the first CUDA API call so we forced called cudaFree(0) in the beginning of our code to provoke it implicitly.

On that note, we can observe that the initialization time remains somewhat stable, regardless of the key size or input data. That is to be expected, as the only thing that changes throughout the whole process is the creation of a smaller/larger key, which should make minor difference in timing.

One other thing that may seem somewhat strange at first glance is the fact that decryption seems to take longer than the encryption process. One would think that since all stages of encryption are reversible, it would take approximately the same time for both the encryption and decryption, but from our research this is not true. Some stages in the decryption process apparently take more time than their corresponding encryption stages. It is also worth noting that in CUDA, decryption requires some more memory operations (CudaMemCopy) in comparison to the encryption process, which also slows down the program a little bit.

Timing Using 128-bit key (ms)

Input size	Serial CPU implementation				OpenMP implementation				CUDA Implementation			
	Entire Program Time	Initialization Time	Total Encryption Time	Total Decryption Time	Entire Program Time	Initialization Time	Total Encryption Time	Total Decryption Time	Entire Program Time	Initialization Time	Total Encryption Time	Total Decryption Time
											Cuda viriables initialization	Cuda viriables initialization
1	1178	10	480	688	322	17	130	175	478	458	9	11
											1	2
5	5755	10	2330	3416	1489	20	604	864	555	460	41	54
											5	6
10	11562	10	4695	6858	2963	17	1205	1741	641	452	82	107
											12	13
50	57862	10	23510	34342	14689	17	5983	8694	1417	475	408	534
											59	64
100	114811	10	46648	68154	29280	20	11902	17358	2327	445	810	1072
											122	126
1000	1148744	10	466215	682517	292216	17	118727	173473	19298	465	8170	10663
											1239	1257
10000	11550000	10	-	-	3010467	22	1256903	1753542	190126	552	824438	107134
											12389	13007

Table 25: All timings 128-bit key

Timing Using 192-bit key (ms)

Input size	Serial CPU implementation				OpenMP implementation				CUDA implementation			
	Entire Program Time	Initialization Time	Total Encryption Time	Total Decryption Time	Entire Program Time	Initialization Time	Total Encryption Time	Total Decryption Time	Entire Program Time	Initialization Time	Total Encryption Time Cuda viriables initialization	Total Decryption Time Cuda viriables initialization
1	1407	10	564	834	379	15	154	211	494	470	11 2	13 2
5	6977	10	2823	4143	1808	18	732	1058	556	445	49 5	62 6
10	13985	10	5685	8290	3565	18	1441	2106	804	584	94 11	126 12
50	69492	10	28124	41358	17368	20	7163	10454	1532	437	469 59	626 63
100	138636	10	56213	82413	35498	20	14402	21075	2655	464	949 118	1242 124
1000	1391841	10	563830	828000	353548	18	142711	210819	22480	540	9421 1203	12519 1296
10000	14000450	10	-	-	3920472	22	1757249	2163201	220292	463	94689 12293	125138 12759

Table 26: All timings 192-bit key

Timing Using 256-bit key (ms)

Input size	Serial CPU implementation				OpenMP implementation				CUDA implementation			
	Entire Program Time	Initialization Time	Total Encryption Time	Total Decryption Time	Entire Program Time	Initialization Time	Total Encryption Time	Total Decryption Time	Entire Program Time	Initialization Time	Total Encryption Time	Total Decryption Time
											Cuda viriables initialization	Cuda viriables initialization
1	1663	10	672	981	442	15	179	250	469	443	12	14
											1	1
5	8196	10	3316	4871	2171	20	858	1292	593	467	53	73
											6	7
10	16262	10	6578	9674	4181	20	1693	2470	713	462	111	140
											12	13
50	81388	10	32932	48447	20675	23	8324	12326	1721	472	535	714
											57	60
100	164010	10	66090	97911	41401	20	16736	24646	2958	461	1065	1432
											121	131
1000	1638680	10	660901	977764	414173	20	167159	246996	25479	472	10775	14232
											1309	1328
10000	16500450	10	-	-	4780723	18	1946792	2833913	251076	462	107484	143128
											12782	13118

Table 27: All timings 256-bit key

ABBREVIATIONS - ACRONYMS

CUDA	Compute Unified Device Architecture
CPU	Central Processing Unit
GPU	Graphics Processing Unit
DES	Data Encryption Standard
AES	Advanced Encryption Standard
RSA	Rivest, Shamir, and Adelman
H2D	Host to Device
D2H	Device to Host

BIBLIOGRAPHY-REFERENCES

1. David A. Patterson, John L. Hennessy "Computer Organization and Design 4th Edition" Volume 1, greek translation by Dimitris Gizopoulos, 2010
2. Maurice Herlihy, Nir Shavit "The Art of Multiprocessor Programming"
3. David Culler, J. P. Singh, Anoop Gupta "Parallel Computer Architecture"
4. OpenAES C implementation, http://nalramli.com/OpenAES/
5. Skype FAQ, https://support.skype.com/en/faq/FA31/does-skype-use-encryption
6. Blaise Barney, Lawrence Livermore National Laboratory "Introduction to Parallel Computing", https://computing.llnl.gov/tutorials/parallel_comp/#Who
7. David C. Brock "Understanding Moore's Law: Four Decades of Innovation", Philadelphia, Chemical Heritage Press 2006,

   https://www.kth.se/social/upload/507d1d3af276540519000002/Moore's%20law.pdf
8. Parallel Programming Languages

   https://en.wikipedia.org/wiki/Parallel_computing#Parallel_programming_languages
9. Margaret Rouse "Advanced Encryption Standard (AES) Defintion", Nov. 2014 http://searchsecurity.techtarget.com/definition/Advanced-Encryption-Standard
10. Sri Vllabh, Aida Janciragic, Sashidhar Reddy, "Security Issues in Networks with Internet Acess", www.utdallas.edu/~ravip/cs6390/slides/Slides_10.ppt
11. Description of Symmetric and Asymmetric Encryption https://support.microsoft.com/en-us/kb/246071
12. Block Ciphers, https://en.wikipedia.org/wiki/Block_cipher
13. Dr. Bill Young, Department of Computer Sciences, University of Texas at Austin, "Stream and Block Encryption" https://www.cs.utexas.edu/~byoung/cs361/lecture45.pdf

14. Michelle Larson, “Differences Between DES and AES Encryption”, Sept. 2014 http://web.townsendsecurity.com/bid/72450/What-are-the-Differences-Between-DES-and-AES-Encryption
15. Block Cipher mode of operation https://en.wikipedia.org/wiki/Block_cipher_mode_of_operation
16. Avi Kak, “AES: The Advanced Encryption Standard”, May 2015, https://engineering.purdue.edu/kak/compsec/NewLectures/Lecture8.pdf
17. Threading Programming Guide https://developer.apple.com/library/mac/documentation/Cocoa/Conceptual/Multithreading/Introduction/Introduction.html
18. Blaise Barney, Lawrence Livermore National Laboratory, “POSIX Threads Programming”, https://computing.llnl.gov/tutorials/pthreads/#Pthread
19. Classification of Parallel Computers https://computing.llnl.gov/tutorials/parallel_comp/parallelClassifications.pdf
20. Shared Memory Computers, https://cvw.cac.cornell.edu/parallel/shared
21. General Purpose Computing on graphics processing units, https://en.wikipedia.org/wiki/General-purpose_computing_on_graphics_processing_units
22. OpenCL, https://en.wikipedia.org/wiki/OpenCL
23. CUDA, https://en.wikipedia.org/wiki/CUDA
24. Margaret Rouse, “Race Condition definition”, http://searchstorage.techtarget.com/definition/race-condition
25. Synchronization (Computer Science) https://en.wikipedia.org/wiki/Synchronization_(computer_science)
26. Nvidia GTX TITAN Z specifications, http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-titan-z/specifications
27. Intel Xeon E7-8870 specifications, http://ark.intel.com/products/53580/Intel-Xeon-Processor-E7-8870-30M-Cache-2_40-GHz-6_40-GTs-Intel-QPI

28. NTUA, “Description of Known Answer Tests and Monte Carlo Tests for Advanced Encryption Standard (AES) Candidate Algorithm Submissions”, January 1998, http://www.ntua.gr/cryptix/old/cryptix/aes/docs/katmct.html

29. Jakob Jenkov, “Amdahl’s Law”, http://tutorials.jenkov.com/java-concurrency/amdahls-law.html

30. Sabbir Mahmud, "A study on parallel implementation of Advanced Encryption Standard (AES)", M.Sc. thesis, Independent University, Bangladesh, 2004

    http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.134.9382&rep=rep1&type=pdf

31. Difference Between AES and 3DES http://www.differencebetween.net/technology/difference-between-aes-and-3des/

32. Brute Force Attack https://www.techopedia.com/definition/18091/brute-force-attack

33. CUDA threads, blocks, grids, warps https://llpanorama.wordpress.com/2008/06/11/threads-and-blocks-and-grids-oh-my/

34. Edgar Danielyan, “Goodbye DES, Welcome AES” http://www.cisco.com/web/about/ac123/ac147/archived_issues/ipj_4-2/goodbye_des.html