OPTIMIZATION OF ADVANCED ENCRYPTION STANDARD (AES) ON CUDA

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF INFORMATICS OF
MIDDLE EAST TECHNICAL UNIVERSITY
BY

BURAK ÇELİK

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE
DEGREE OF
MASTER OF SCIENCE
IN
THE DEPARTMENT OF CYBER SECURITY

SEPTEMBER 2019

Approval of the thesis:

OPTIMIZATION OF ADVANCED ENCRYPTION STANDARD (AES) ON CUDA

Submitted by **BURAK ÇELİK** in partial fulfilment of the requirements for the degree of **Master of Science in Cyber Security Department, Middle East Technical University** by,

Prof. Dr. Deniz Zeyrek Bozşahin
Dean, **Graduate School of Informatics** _____

Assoc. Prof. Dr. Aysu Betin Can
Head of Department, **Cyber Security** _____

Assist. Prof. Dr. Cihangir Tezcan
Supervisor, **Cyber Security, METU** _____

Examining Committee Members:

Prof. Dr. Ali Aydın Selçuk
Computer Engineering Department, TOBB _____

Assist. Prof. Dr. Cihangir Tezcan
Cyber Security, METU _____

Assoc. Prof. Dr. Banu Günel Kılıç
Information Systems, METU _____

Date: _____

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

Name, Last name : BURAK ÇELİK

Signature : _____

ABSTRACT

OPTIMIZATION OF ADVANCED ENCRYPTION STANDARD (AES) ON CUDA

ÇELİK, BURAK
MSc, Department of Cyber Security
Supervisor: Assist. Prof. Dr. Cihangir Tezcan

September 2019, 89 pages

This thesis presents several optimization techniques of AES implementations on CUDA. 6 different CUDA kernels are implemented for AES-128 exhaustive search with different software designs and they are compared with each other using Nsight experiment results. Outcome of these results are used for finding the best CUDA implementation and from it, AES-128, AES-192 and AES-256 versions are created for exhaustive search, on the fly CTR and file encryption. They are compared with CPU implementations in order to decide whether GPU or CPU is the fastest considering these topics. For this comparison, two different type of CPU implementations are created which are AES-NI, using new instruction set of Intel, and basic C++. 1, 2, 4 and 8 threads versions of these implementations are compared with CUDA and results are shared. According to them, CUDA is 21, 19 and 18 times faster than the best CPU implementations for exhaustive search with respect to key length. These ratios are 4 times for CTR implementations in which 37.52 GBs of data can be encrypted each second while using CUDA. File encryption for CUDA is 22, 19 and 17 times faster than the best CPU implementations. CUDA can encrypt 31.24 GBs of data per second in this regard without considering I/O operations.

Keywords: AES, GPU, CUDA, AES-NI

ÖZ

CUDA KULLANARAK GELİŞMİŞ ŞİFRELEME STANDARDI (AES) OPTİMİZASYONU

ÇELİK, BURAK
Yüksek Lisans, Siber Güvenlik Bölümü
Tez Yöneticisi: Dr. Öğr. Üyesi Cihangir Tezcan

Eylül 2019, 89 sayfa

Bu tez, CUDA üzerinde yazılan AES uygulamaları ile ilgili optimizasyon tekniklerini sunmaktadır. Farklı yazılım tasarımlarına sahip AES-128 için yazılmış 6 farklı CUDA kapsamlı arama uygulaması geliştirilmiş ve Nsight deney sonuçları kullanılarak birbirleriyle karşılaştırılmıştır. Sonuçlar, en iyi CUDA uygulamasını bulmak için kullanılmıştır. AES-128, AES-192 ve AES-256 sürümleri için kapsamlı arama, CTR ve dosya şifreleme uygulamaları bulunan en iyi CUDA sürümü üzerinden geliştirilmiştir. Bu konular göz önünde bulundurularak, GPU ve CPU uygulamaları karşılaştırılmıştır. Bu karşılaştırma için yeni Intel komut seti (AES-NI) ve temel C++ olmak üzere iki farklı CPU uygulama seti geliştirilmiştir. Bu uygulamaların 1, 2, 4 ve 8 iş parçacıklı versiyonları CUDA ile karşılaştırılıp sonuçlar paylaşılmıştır. Elde edilen sonuçlara göre, CUDA, anahtar uzunluğuna göre kapsamlı arama için en iyi CPU uygulamalarından 21, 19 ve 18 kat daha hızlı olarak bulunmuştur. Bu oranlar CTR modu ile şifreleme için 4 kata düşmüş olarak hesaplanmıştır. Bu modda, CUDA saniyede 37.52 GB veri şifreleyebilmektedir. Dosya şifreleme kısmında ise CUDA 22, 19 ve 17 kat daha hızlı olarak ölçülmüştür. CUDA, bu konuda ise saniyede 31.24 GB veriyi şifreleyebilir.

Anahtar Kelimeler: AES, Ekran kartı, CUDA, AES-NI

ACKNOWLEDGMENTS

I would like to thank Dr. Tezcan for his perpetual support and wisdom from the inception of this thesis to its conclusion for he spared none while working with me. Without him, this research would not be as meticulously detailed as it is. He always encouraged me to set sail in uncharted waters and steered me into the proper course when I faced a rub along the way.

I also wish to extend my gratitude to my father, my mother and my brother. They are the ones who raised me and made me the person who I am today. Without them, I would have never been involved in computer software and this thesis would not exist.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

AES	Advance Encryption Standard
AES-NI	AES New Instructions
AMD	Advanced Micro Devices
CBC	Cipher Block Chaining Mode
CFB	Cipher Feedback Mode
CPU	Central Processing Unit
CTR	Counter Mode
CUDA	Compute Unified Device Architecture
DES	Data Encryption Standard
ECB	Electronic Codebook Mode
FIPS	Federal Information Processing Standard
GCN	Graphics Core Next
GPU	Graphics Processing Unit
IDE	Integrated Development Environment
IEC	International Electrotechnical Commission
IPC	Instruction Per Clock
IPW	Instructions Per Warp
ISO	International Organization for Standardization
IV	Initialization Vector
MD5	Message Digest Algorithm 5
NIST	The National Institute of Standards and Technology
PCIe	Peripheral Component Interconnect Express
RAM	Random Access Memory
S-box	Substitution box
SHA-1	Secure Hash Algorithm 1
SHA-2	Secure Hash Algorithm 2
SHA-3	Secure Hash Algorithm 3
SM	Streaming Multiprocessor
SP	Streaming Processor
SPN	Substitution Permutation Network
TDEA	Triple Data Encryption Algorithm
VRAM	Video Random Access Memory

CHAPTER 1

INTRODUCTION

1.1. Cryptography

Since the beginning of human interactions, securing valuable messages or concealing the communication data has been one of the most important complications of civilizations. Due to this need of protecting sensitive information, cryptography has been developed. Its initial traces go back to the Egyptians 4000 years ago [1]. Since then, it has been a crucial part of human civilizations. It played a critical role in both world wars. In the time of the World War II, cryptographic mechanical cipher machines were used widely. The most notable one of them is called the enigma machine used by Nazi Germany to transmit messages. Its cryptanalysis played a vital role on the course of the war.

The basis of cryptography is built on concealing valuable data from unwanted eyes. By using various cryptographic techniques, the communication should look like rubbish to eavesdroppers. However, the right person, whom the message is meant for, could decode the gibberish communication into meaningful message.

Before the computer era, cryptography was solely based on linguistic patterns. With the emergence of computers which made the old ciphers easy targets, cryptographic algorithms have been designed based on some mathematical theories, making these algorithms hard to break in practice. Ergo, newfound ciphers began to expand their complexity in order to be more secure. Thus, modern cryptography was born.

Modern cryptographic algorithms consist of mainly three topics. These are called:

- **Secret Key Cryptography:** Uses a single key for both encrypting and decrypting the data, also called as symmetric encryption. Dating back to Ancient Rome where it is used as Caesar cipher to communicate in secret. Some examples are: AES, DES, Blowfish, Serpent and A5/1

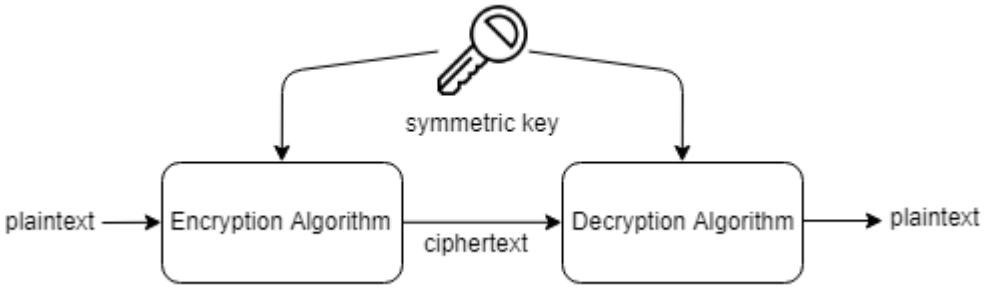


Figure 1.1: Secret Key Cryptography

- **Public Key Cryptography:** Uses one key for encryption and one for decryption purposes, also called as asymmetric encryption. Also used for authentication and key exchange for symmetric encryption.

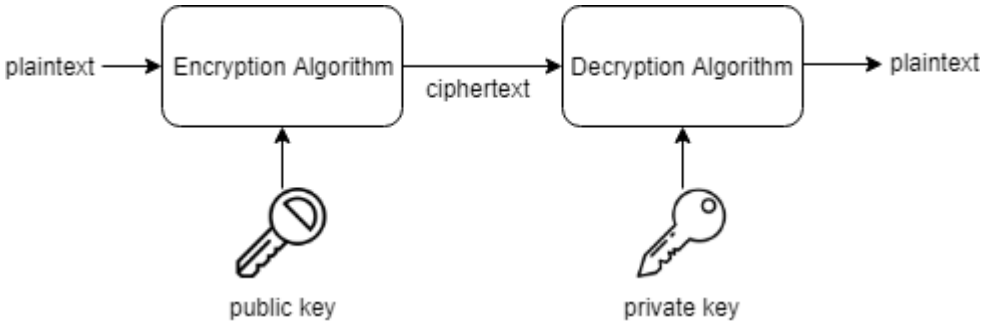


Figure 1.2: Public Key Cryptography

- **Hash Functions:** Uses some transformations to irreversibly transform the data so that the output becomes the fingerprint of the plaintext. Mostly used for message integrity concerns. Some examples are: MD5, SHA-1, SHA-2 and SHA-3

Figure 1.3: Hash Functions

1.2. Block Ciphers

Block ciphers are deterministic cryptographic algorithms, operating on fixed length of data. They belong to secret key cryptography section since they use one key for encryption and decryption purposes. Most of the modern designs are based on iterated ciphers in which each round of cipher algorithm is iterated over some predetermined

number. Usually, each round takes different round keys produced from the secret key via key scheduling algorithm of the related cipher. The purpose of this key scheduling process is to create different sets of keys in order to be used in round functions. In this way, each round has a completely different key.

There are several design paradigms for block ciphers. One of them is called Feistel Networks. In this pattern, the block to be encrypted is divided into two equal sized halves. A dedicated round function is applied to one half using the round key while the other half is XORed with the result of the round function half. Then, the two halves are swapped with each other. DES is the quintessential Feistel block cipher adopted in 1976 as a FIPS for the United States of America, which is illustrated in Figure 1.4. It operates on 64 bits of blocks with 16 rounds while having the same key size length [2]. However, it only provides 56 bits of security since the least significant bits of each byte is a parity bit which are used for error detection. Due to its short key size, security of the basic form of DES is compromised [3]. Moreover, NIST publicly withdrew DES in 2005 while proposing a newfound secure version of the same cipher, TDEA [3].

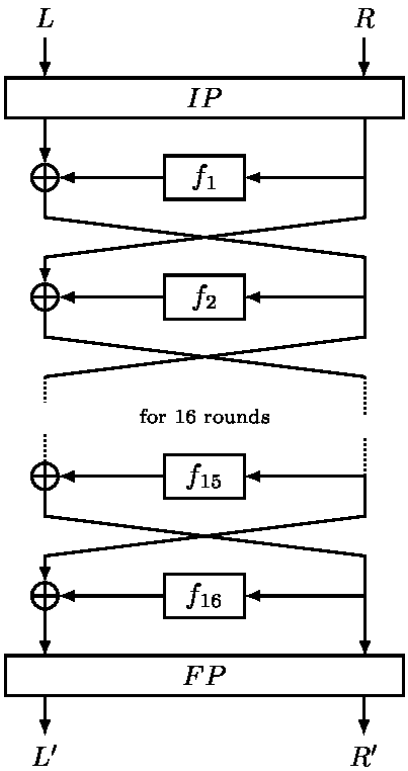


Figure 1.4: DES Feistel Network block cipher

TDEA, also known as Triple DES, is basically three DES iterations where each iteration uses different keys. Thus, total of three DES keys are used for one TDEA iteration. NIST only allows using TDEA with three different keys since other options

decrease the security of the cipher [4]. Encryption works like series of encrypt-decrypt-encrypt blocks while decryption is achieved through decrypt-encrypt-decrypt operations while using different keys for each DES iteration. Each triple process encrypts one block of 64 bits of data just like DES. Nevertheless, OpenSSL believed in 2016 that the security of TDEA was considered low and removed from the default cipher options. In order to use it, *"enable-weak-ssl-ciphers"* config must be given [5].

The other most known paradigm for block ciphers is called SPN. It is the direct implementation of confusion-diffusion paradigm proposed by Claude Shannon who is considered as the father of information theory [3]. Basically, in each round, confusion and diffusion functions operate on input block. Besides that, round keys are added to ciphertext in each round. Confusion paradigm makes the resulting ciphertext depend on the input plaintext in such a way that it should not be exploitable. This increases the vagueness of the ciphertext while making the relation between plaintext and ciphertext as complicated as possible. Substitution part of SPNs, mostly utilizes on S-boxes, provides confusion on the ciphertext. On the other hand, diffusion makes sure that each piece of the input should influence many of the different ones in the output. In this way, the redundancy of the plaintext is increased so that it should be obscure enough to prevent attempts to deduce the key. In SPNs, permutation layers make sure the diffusion paradigm is applied to the block cipher. The most notable SPN block cipher is AES, also known as Rijndael.

1.2.1. Mode of Operations

Since block ciphers only work on fixed lengths of block sizes, operation modes are needed for transforming encryption of a single block of data into a complete encryption algorithm. Basically, a mode of operation gives instructions on how to connect blocks in order to form a large data block. There are various modes of operations for different purposes like ECB, CBC, CFB or CTR. Most of these modes of operations are published by NIST with the correct implementation options and test vectors [6].

Some of the operations use IV for each encryption or decryption. IV must be random and non-repeating in order to ensure of producing distinct ciphertexts for each block when using the same key. This is considered a common problem for ECB. In ECB, every block is encrypted with the same key, thus, resulting in the same ciphertext for the blocks with the same plaintext. This creates a huge disadvantage since the encryption of whole block is compromised due to lack of diffusion. NIST encourages not to use this mode if this property is not desirable [6].

In this research, CTR mode (Figure 1.5) is selected since it allows parallelization of both encryption and decryption. Parallelization is needed for CUDA implementations

as well as CPU ones. CTR mode uses successive values of a predefined counter. These counter values are encrypted as a sequence of output blocks and they are XORed with the plaintext to produce ciphertext, and vice versa. Each sequence counter must be unique in order to create diffusion among input blocks. This is recommended by NIST [6]. There are several methods for generating counter numbers but in this research, the standard incrementing function proposed by NIST [6] is used. Basically, given an initial counter block for a message, successive counter blocks are derived by applying an incrementing function.

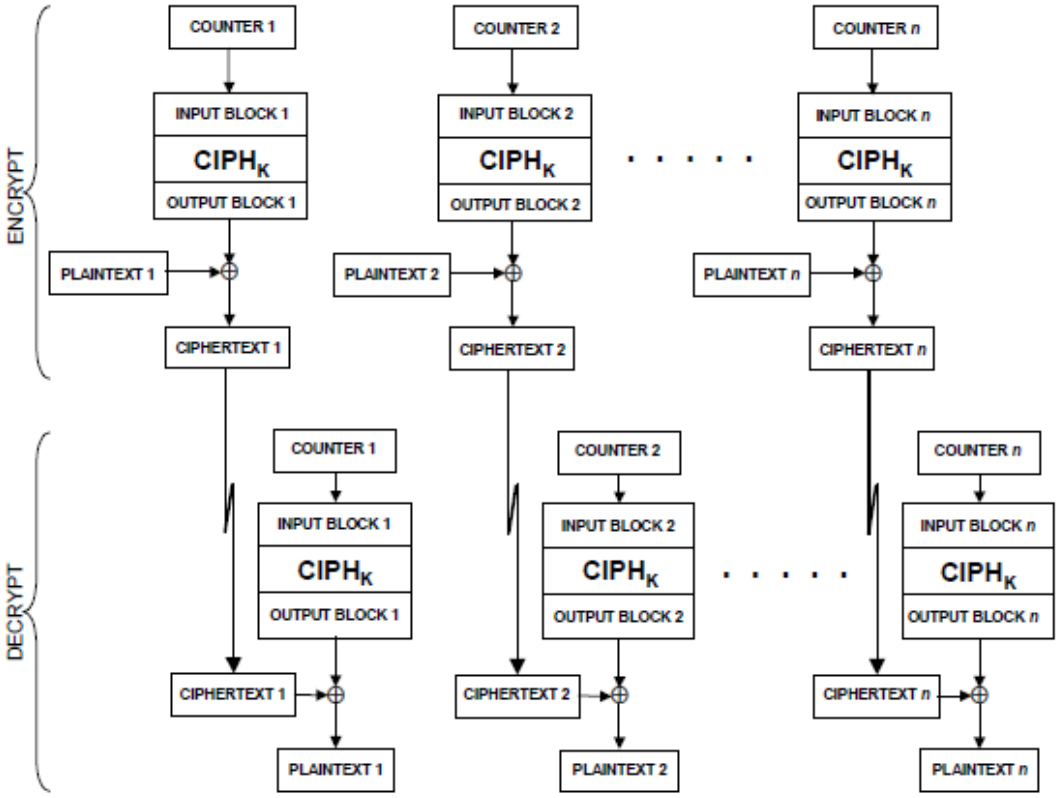


Figure 1.5: The CTR mode [6]

1.3. AES

Advanced Encryption Standard is a symmetric block cipher established by NIST in 2001. It's the subset of the Rijndael block cipher which is developed by two cryptographers from Belgium, Vincent Rijmen and Joan Daemen. AES has been adopted by the U.S government and used worldwide. It became a U.S. federal government standard [7] in 2002 and is included in the ISO/IEC 18033-3. AES is approved by the NIST for protecting sensitive (unclassified) information that requires cryptographic protection [8].

When NIST announced a competition for choosing a successor to Data Encryption Standard (DES) which was starting to become vulnerable to brute-force attacks, the international cryptographic community took the interest and sent several algorithms to NIST in order to be evaluated. NIST expected all new algorithms to be blocks ciphers with block size of 128 bits and key sizes of 128, 192 and 256 bits. After the announcement in 1997, NIST accepted several symmetric key algorithms submitted by the world cryptographic community.

In 1999, NIST selected five algorithms from the contestants for extensive analysis. Implementations were tested for these five block ciphers considering speed, reliability and resistance to various attacks both in hardware and software. After substantial discussions, Rijndael cipher was selected as the proposed algorithm in 2000 [9] and published by NIST as an encryption standard [7]. In 2003, the U.S. government announced that AES could be used to protect sensitive information and after that it became a U.S. encryption standard for data encryption.

After its successful use by the U.S. government, usage of AES has been immensely increased in private sector. Since NIST used transparent selection process for the election of candidates, security and cryptography experts put high level of confidence in AES. This has allowed AES to become the most popular block cipher since its release. Today, it is used in many protocols such as Secure Sockets Layer (SSL), Transport Layer Security (TLS) and Hyper Text Transfer Protocol Secure (HTTPS).

1.4. AES-NI

Since AES is widely used for cryptographic purposes and considered the current dominant block cipher, CPU manufacturers such as Intel and AMD proposed an extension in 2008 to the x86 instruction set architecture, known as Advanced Encryption Standard New Instructions Set, shortly AES-NI. These new instructions enable fast and secure data encryption and decryption using AES defined by FIPS publication number 197 [7]. Since the encryption and decryption processes happens on the hardware level inside CPU, AES-NI has significant performance benefits considering its software implemented rivals. AES-NI supports every standard key length of AES using the standard block size.

Intel suggests that AES-NI increases performance by more than an order of magnitude for parallel modes of operation such as CTR and provides roughly 2-3 fold gains for non-parallelizable modes like CBC [10]. It provides six new instructions for AES that offer full hardware support. These instructions consist of processing each round and key scheduling. In the White Paper document [10] provided by Intel Corporation, examples implemented in C programming language can be found for each key size.

Besides improving performance, AES-NI provides some security benefits such as preventing side channel and cache-based attacks since it does not rely on table-based lookups for each AES round processing. In addition to that, it is easier to implement AES-NI version of the cipher concerning the software implementation due to prepared instructions, reducing the overall code size.

1.5. GPU

GPU is a piece of electronic circuit that is specialized in fast graphics rendering especially in computer games and image processing. GPUs are widely used in embedded systems, personal computers, mobile phones and game consoles. They are originally used solely for gaming. However, the processing power residing inside these cards unveiled a new approach. Modern GPUs have advanced capabilities which are being harnessed more broadly for accelerating computational workloads. Their highly parallel structure makes them more efficient than CPUs for processing large blocks of data in parallel.

In terms of architecture, GPU is composed of hundreds of cores that can handle thousands of threads simultaneously. On the other hand, CPU has just a few cores with lots of cache memory that can handle a few software threads at a time [11]. Basically, CPU has extremely intelligent sparse cores while GPU has surplus of cores with shortage in cleverness in terms of instructions.

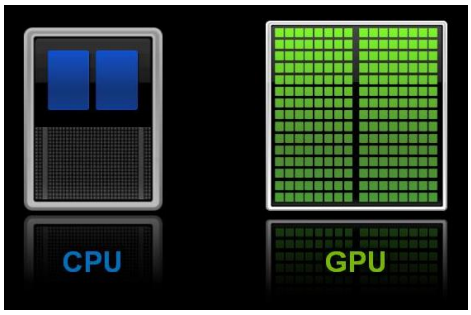


Figure 1.6: The difference between a CPU and a GPU [11]

GPUs have been becoming more and more popular nowadays. The reason for that can be interrelated with several topics. One of them is crypto mining. GPUs offer efficient cryptocurrency mining due to its architecture considering CPU. When cryptocurrency algorithms were launched around 2014, the demand for GPUs on the market skyrocketed. This also increased the price tags of GPUs.

Another topic can be considered as deep learning. Deep learning is a subset of AI and machine learning that uses multi layered artificial neural networks to deliver improvised accuracy in tasks such as object detection, speech recognition etc. Since

the training part of neural networks take too much time, using GPU comes in handy on this topic. Considering CPU, GPU offers a huge performance difference on the training of deep learning algorithms. For instance, training times for AlexNet, a convolutional neural network designed by Alex Krizhevsky and the winner of 2012 ImageNet contest by a huge margin due to the utilization of GPUs during training [12], can be found in Figure 1.7 which shows that using a high-end GPU results in being almost 15 times faster than using a high-end CPU [13]. This just shows how much power GPUs have in store.

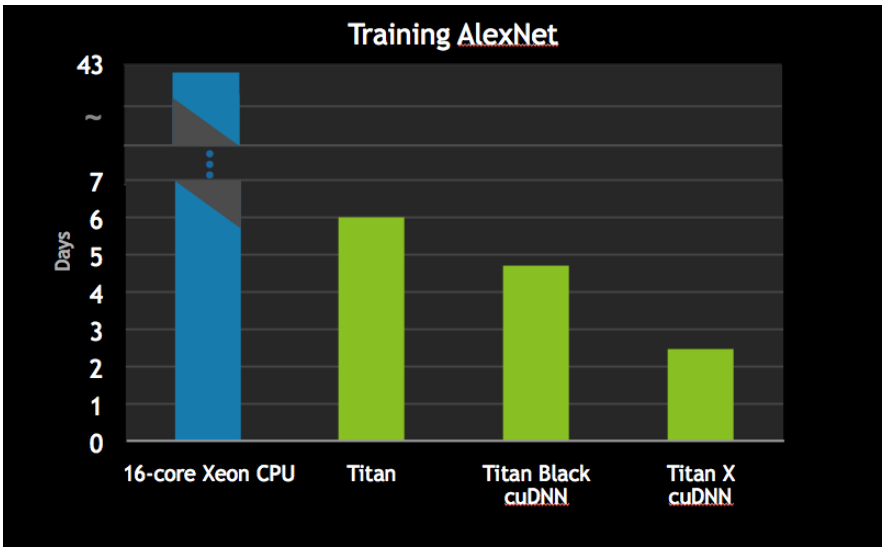


Figure 1.7: Training days for AlexNet [13]

GPUs are widely used for cryptographic purposes since most of the modern cryptographic algorithms are embarrassingly parallel and can easily be implemented for multithreaded architectures. In this regard, the power that can be extracted from GPUs can greatly enhance the output performance as they offer greater efficiency in terms of parallelism with respect to CPUs. For instance, the first SHA-1 collision was found with the help of GPUs by combining many special cryptanalytic techniques. The computational effort spent in this attack was equivalent to $2^{63.1}$ SHA-1 compressions [14]. According to their calculations, this would take approximately 6,500 CPU years or 100 GPU years. GPU shortens the total time spent on the attack by 65 times. Moreover, another research proved that optimizations can even further improve the efficiency that GPUs offer as 9 days of brute force attack on CRYPTO1 stream cipher was able to be decreased to less than 5 hours with various techniques [15]. This shows that GPUs provide great performance output and it can even be extended with special implementation techniques. There is also a research concerning chosen-prefix collision attack on SHA-1 and MD5 [16]. While the computation cost of this attack is somewhat practical, it still requires huge computational power which means thousands of GPUs in order to obtain the chosen-prefix collision in a reasonable

time. Research also shows that collision attack on MD5 has a complexity of 2^{40} which is very feasible to do for a GPU. Thus, GPUs are very useful in terms of cryptographic algorithms and offer a lot of power.

Table 1.1: *GPU Market share changes* [17]

GPU Supplier	Market share in 2017	Market share in Q3'18	Market share in Q4'18
NVIDIA	66.3%	74.3%	81.2%
AMD	33.7%	25.7%	18.8%

There are two big GPU manufacturers in the world: NVIDIA and AMD. GPU market is fairly divided to these two corporations. However, according to Jon Peddie Research; NVIDIA has the overwhelming share with 81.2% in Q4 2018 while AMD only has 18.8% [17]. Moreover, Steam, a popular digital distribution platform developed by Valve Corporation for purchasing video games, provides user statistics on GPU usage. According to this usage statistics, NVIDIA GPUs dominate the gaming market as well with 75.02% share as of February of 2019 [18]. Due to this dominance among the GPU market, in this research, CUDA developed by NVIDIA is selected for parallel processing platform.

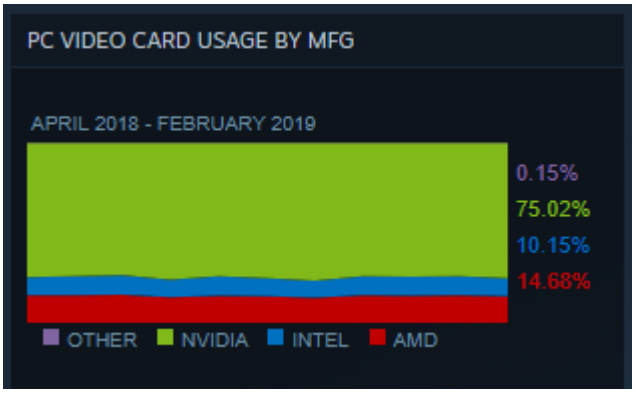


Figure 1.8: Steam GPU usage statistics [18]

1.6. CUDA

CUDA is a parallel computing platform and programming model developed by NVIDIA for GPUs, introduced in 2006. Its main purpose is to solve complex computational problems in a more efficient way than a CPU can. GPU accelerated applications developed by using CUDA are programmed with popular languages like C, C++, Fortran, Python and MATLAB. CUDA enables developers to speed up the computation by using the impressive floating-point performance of GPUs. Since GPUs have a lot of compute intensive cores, it allows CUDA to harness the highly

parallelizable data processing potential. General power of CUDA comes from parallel computations where the same program is executed on many data elements.

CUDA utilizes threads on GPU cores so that large data sets can be processed at the same time. Basically, it allows developing applications which scales its parallelism to leverage the increasing number of GPU cores. Without CUDA, it would take lots of time to code something onto the GPU. So, it can be considered as an efficient framework for writing complex programs for GPU architecture.

1.7. Overview

In this research, CPU and GPU implementations of AES are compared with each other and their performance is evaluated. Several GPU implementations are developed in order to reach the highest throughput and their performance values are also assessed with each other to find the best GPU implementation among them. Several topics like occupancy, cache hit rate and bank conflicts related to CUDA are discussed for these GPU implementations. Besides that, CPU implementations are implemented on C++. They can be divided to two pieces: base C++ implementation and AES-NI version on C++.

Implementations are developed on a 64-bit Operating-System, Microsoft Windows 10 Home Edition, with 16 GB RAM and x64-based processor, Intel Core i7-4770K CPU with the base processor frequency of 3.50 GHz. This CPU can increase the frequency of its cores with Intel Turbo Boost Technology to 3.90 GHz. It is fairly an old CPU launched on Q2'13. Moreover, it has Intel AES New Instructions deployed which enables to encrypt or decrypt using just CPU instructions.

For GPU implementations, NVIDIA GTX 970 graphics card is used. This card has the compute capability version of 5.2 and 1664 CUDA cores with the base clock frequency of 1,050 MHz. It has 4 GB GDDR5 memory, VRAM, with 256-bit memory interface [19]. CUDA version 9.2 is used with this card. This GPU comes with a software called GPU Tweak which provides an intuitive interface to access serious functionalities of GPU. From this software, GPU clock frequency is boosted to 1,400 MHz which increases the overall performance output of the card. Every implementation in this research is run with this specification. According to our calculations, this boost creates approximately 16% performance increase for each CUDA implementation. Furthermore, CPU can also be overclocked like this case but the performance difference between GPU and CPU is so much that even with overclocking the CPU, its output cannot reach the performance level of CUDA.

While developing the implementations, Microsoft Visual Studio 2017 IDE Community Edition is used. C++ and AES-NI implementations are compiled with Visual C++ 2017 compiler. AES-NI development is prepared with the help of AES-NI White Paper published by Intel Corporation [10]. For the whole process of development, a private GitHub repository is used for the version control of the code. CUDA part of the implementation is developed using C++ with Visual C++ 2015 compiler. The reason for selecting a different compiler for CUDA considering CPU implementations is that CUDA version 9.2 was not supported by the newly Visual C++ 2017 compiler. Therefore, Visual C++ 2015 compiler is selected for CUDA implementations.

Three types of implementations are developed for AES-128, AES-192 and AES-256 individually. First of them is exhaustive search in which the application tries to find the correct key for given a plaintext and a ciphertext. Secondly, on-the-fly encryption using CTR mode is developed. After that, file encryption using this CTR implementation is developed. The performance difference between them is discussed.

For GPU implementations, several distinct optimization techniques are applied on exhaustive search implementation of AES-128. These different techniques are examined, and their advantages and disadvantages are discussed. From the obtained results, AES-192 and AES-256 implementations are developed for CUDA.

1.8. Related Work

GPUs are widely used for several researches in order to show that they are the better choice when it comes to cryptographic algorithms. In the paper named as "Cryptanalysis of the Full AES Using GPU-Like Special-Purpose Hardware", researchers investigated the feasibility of a hypothetical supercomputer consisting of 3.10^{10} GPU-like AES processors which can reach a throughout of up to 10^{12} AES operations per seconds [20]. Although the speed is very promising, they concluded that the cost of building such hardware is so high it would be roughly 1.5 trillion US$ in 2009. Another research, "Fast software AES encryption", showed that GPU has the best performance among other processors like 8-bit and 32-bit microcontrollers [21]. NVIDIA 8800 GTX card is used in that research which has 575 CUDA cores. Its pre-computed key scheduling encryption, like CTR case, produces throughput of 23.3 Gb/sec on AES-128, which is approximately 2.91 GB/sec. Since they did not release source codes, it is not feasible to exactly compare our results with theirs. However, our CUDA implementation can reach 37.52 GB/sec. Considering their GPU has 575 CUDA cores and ours has 1664 CUDA cores, almost 3 times of improvement is expected considering normal cases. However, advancement margin is huge, and it is not only because we use a more up-to-date card, but because of our better optimization

techniques. Moreover, that research shows throughput of 12.9 GB/sec when key schedule is not pre-computed just like exhaustive search case. This equals to approximately 1.61 GB/sec while our best exhaustive search implementation can reach 26.3 GB/sec throughput.

"Acceleration of AES encryption on CUDA GPU" paper investigated GPU capability of AES encryption on CUDA [22]. They were able to achieve 35.2 Gbps AES encoding throughput with NVIDIA GeForce GTX 285 (240 CUDA cores) which strongly shows the great potential of CUDA for cryptographic accelerator in that research. This ratio equals to 4.4 GB/sec and it is nowhere near to our results on CUDA AES-128 CTR mode which is 37.52 GB/sec. They also suggested the best case for storing look-up tables and round keys is shared memory which is the case of our research.

Another research was conducted on the power efficiency evaluations of block ciphers on Kepler and AMD GCN [23]. According to the experiment results which are done on GTX 680 (1536 CUDA cores) and Radeon HD 7970 (2048 cores), HD 7970 consumed approximately 30% more power than GTX 680 on all variants of AES. However, Radeon HD 7970 produced more throughput as expected since it has many more CUDA cores than GTX 680. The search is concluded that Kepler architecture is designed to be effective and less power consuming while GCN is focused on the throughputs of arithmetic and logical instructions.

Researchers from AMD investigated the computing resources of GPU, AMD HD 2900 XT (320 cores), in "Symmetric Key Cryptography on Modern Graphics Hardware" paper [24]. According to them, this single graphics card is able to achieve data rates of 3-30 Gbps. It also speeds up symmetric key implementations by 6-60 times with respect to high performance CPUs. This indicates the power difference between GPU and CPU. However, their CUDA result which is up to 3.75 GB/sec is still not better than our findings. Another research is done on GPU acceleration of block ciphers (AES, Blowfish, IDEA, DES) in OpenSSL library [25]. Results show that GPU brings about 8-10 times improvement in speed considering CPU. Therefore, they were able to accelerate symmetric block ciphers using GPUs compared to traditional CPU implementations. Considering full disk encryption, a research named as "Fast disk encryption through GPGPU acceleration" is done on XTS mode of operation applied together with Twofish algorithm within TrueCrypt suite [26]. NVIDIA GTX 260 (192 CUDA cores) is used in that research and it is found that GPU outperforms a four core CPU by 67%. They stated that using GPU as co-processor for disk encryption is proved to be an effective accelerator for both single users and network area storage systems.

Intel also published a paper "Breakthrough AES Performance with Intel AES New Instructions" concerning the speeds of AES-NI implementations on CPU [27]. In that

research, authors showed the excellent performance of AES algorithm on Intel Core i7-980X Extreme Edition CPU using AES-NI. This CPU has 6 cores and 12 threads. They achieved approximately 0.24 cycles per byte on 6 cores for AES-128 on parallel modes. This CPU has max turbo frequency of 3.60 GHz. If we take this into account as 3.60 x 10^9 cycles per second, according to equation (used for every cycle per byte calculation in this research):

$$cycles\ per\ byte = \frac{\text{cycles per second}}{\text{bytes per second}}$$

0.24 cycles per byte is equivalent to 13.96 GB/sec throughput on this CPU. This is an extreme edition CPU with a very high price tag and our best CUDA implementation on AES-128 CTR mode is better than their result by a big margin with using almost 5 times cheaper GPU. When our GPU clock frequency is considered as 1.4 GHz, the same equations shows 0.03 for our cycle per byte result which is equivalent to 37.52 GB/sec. It must be noted that our CUDA implementation is run on 1664 CUDA cores. This basically shows that each CUDA core has approximately 57.83 cycle per byte value while each CPU cores of Intel White Paper has around 1.44 cycle per byte. This also proves that CPU cores are much more advanced and efficient than CUDA cores. However, when large amounts of CUDA cores are gathered together to work, they can outperform CPU cores like the case of this research.

1.9. Our Contribution

In this research, we present AES implementation techniques for CUDA on exhaustive search, on the fly CTR encryption and file encryption using CTR mode. Several different techniques for AES implementation on are described and they are compared with each other using Nsight results of AES-128 exhaustive search. The best kernel among them is selected and compared with CPU implementations, which consist of AES-NI and C++. Other AES key lengths are also implemented in this research. According to our results, CUDA appears to be the superior option. It outperforms even AES-NI which is the hardware implementation. CUDA shows it superior power difference on exhaustive search and file encryption in which it is faster around 20 times than the best CPU implementation. Our results show that CUDA is still the fastest option on CTR implementation, but it is only 4 times faster than AES-NI.

AES is used all around the world. Findings of this thesis are important since using GPU can significantly improve the performance of AES. For instance, theoretical experiments on AES can be verified on GPU easier than CPU. Another topic that CUDA provides performance difference is full disk encryption. By using CUDA, applications which use full disk encryption should take lesser time. Moreover, this research also shows that by using CUDA, AES based cryptographic algorithms like

SHA-3, RECTANGLE etc. can be greatly improved in performance if they are implemented on GPU. Cryptocurrency mining is another topic in which GPUs outperform CPUs. Moreover, algorithms used on these cryptocurrencies are being improved each day in terms of efficiency. Optimizations done on these algorithms increase the capability of cryptocurrency mining which in return results in greater cryptocurrency gains. So, in this regard, keenly optimizing the code is also important considering the efficiency of the results.

This thesis consists of several chapters. Chapter 2 explains the basic structure of AES and its key schedule. It also provides information on how AES can be implemented as software. Chapter 3 is about CUDA and its architecture. It is important to understand what kind of environment that CUDA provides in order to optimize the code. For this reason, aspects related to GPU programming like thread and memory hierarchy are explained in a detailed manner. Chapter 4 presents the implementation results of CUDA, AES-NI and C++. 6 different kernels are meticulously compared with each other using Nsight results of AES-128 exhaustive search. After that, the best kernel is selected and compared with CPU implementations of AES-NI and C++. This process is done for 128, 192 and 256 bits of AES keys. Moreover, the same comparison between GPU and CPU is done for on the fly CTR and file encryption implementations. The results are presented in Chapter 5.

CHAPTER 2

AES

2.1. Design Specifications

AES processes 128 bits data blocks for encryption while using cipher keys with length of 128, 192 or 256 bits. It uses a rectangular structure to hold these bits. In this rectangular structure, basic processing unit is a byte which consists of consecutive 8 bits. Ergo, AES has the option to work on 16, 24 and 32 bytes of keys respectively. According to these key sizes, number of rounds is changed for each encrypted block; 10, 12 and 14 respectively (Table 2.1). Internal rectangle structure of AES can be considered as a two-dimensional array of bytes which is called the state. Every state consists of four rows of bytes. At the beginning of the encryption, the input block consist of 16 bytes is distributed to this rectangle structure column-wise (Figure 2.1).

Table 2.1: AES Key-Block-Round Combinations

	Block Size (byte)	**Key Length** (byte)	**Number of Rounds**
AES-128	16	16	10
AES-192	16	24	12
AES-256	16	32	14

p_0	p_4	p_8	p_{12}
p_1	p_5	p_9	p_{13}
p_2	p_6	p_{10}	p_{14}
p_3	p_7	p_{11}	p_{15}

k_0	k_4	k_8	k_{12}	k_{16}	k_{20}
k_1	k_5	k_9	k_{13}	k_{17}	k_{21}
k_2	k_6	k_{10}	k_{14}	k_{18}	k_{22}
k_3	k_7	k_{11}	k_{15}	k_{19}	k_{23}

Figure 2.1: State and cipher key layout for block size of 128 bits and key of 192 bits [9]

AES is implemented on four round transformations. These individual transformations; SubBytes, ShiftRows, MixColumns and AddRoundKey, are responsible for changing the state with respect to their purposes. These four transformations compose the round transformation which can be explained in the pseudo code below:

```
Round (State, RoundKey) {
    SubBytes (State)
    ShiftRows (State)
    MixColumns (State)
    AddRound (State, RoundKey)
}
```

However, the final round of the cipher is slightly different as MixColumns transformation is removed.

```
FinalRound (State, RoundKey) {
    SubBytes (State)
    ShiftRows (State)
    AddRound (State, RoundKey)
}
```

Before the round transformations, initial cipher key is used in AddRound transformation. After that, for n-1 rounds (Table 2.1), Round function is processed. In the last round, FinalRound function is used. Overall AES encryption structure can be demonstrated as:

```
AddRound (State, RoundKey)
// For n-1 rounds
Round (State, RoundKey) {
    SubBytes (State)
    ShiftRows (State)
    MixColumns (State)
    AddRound (State, RoundKey)
}
// For final round
FinalRound (State, RoundKey) {
    SubBytes (State)
    ShiftRows (State)
    AddRound (State, RoundKey)
}
```

2.1.1. SubBytes Transformation

SubBytes transformation is responsible for non-linear byte substitution that operates independently on each byte of the state using a predetermined substitution table called the S-box (Figure 2.2). So, in this stage of the encryption, every individual byte of the

state is changed according to the S-box and the new value is overwritten onto the current one. S-box operates on 4 bits of two index structure of a byte. For instance, if the byte representation of $state_{1,1}$ is {47}, then the substitution value can be found by the intersection of the row with index 4 and the column with index 7. This would result in {47} having the value of {a0} (

Table *2.2*).

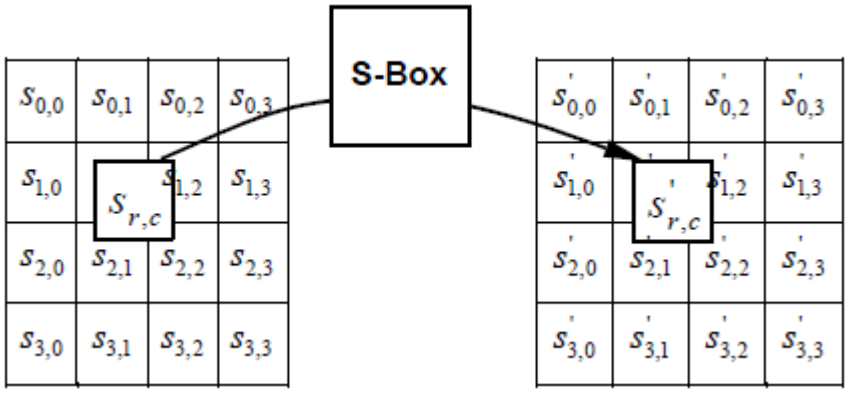


Figure 2.2: SubBytes applies the S-box to each byte of the state.

Table 2.2: AES S-Box: Substitution values for bytes (in hexadecimal format)

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**
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	d1	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

2.1.2. ShiftRows Transformation

ShiftRows transformation is all about shifting the bytes of the last three rows of the state. The first row is left unchanged. Each byte of the second row is shifted one to the left. Similarly, the third and fourth rows are shifted two and three times to the left respectively (Figure 2.3).

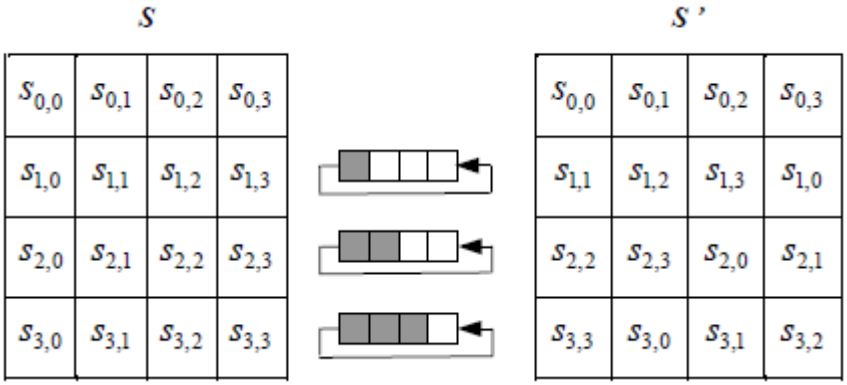


Figure 2.3: ShiftRows cyclically shifts the last three rows of the state

2.1.3. MixColumns Transformation

MixColumns transformation operates on the state column by column (Figure 2.4). This operation is all about matrix multiplication of the related columns. Inside MixColumns function, the resulting column is obtained by a matrix multiplication which can be considered as multiplication in Galois Field, GF(2^8) [9].

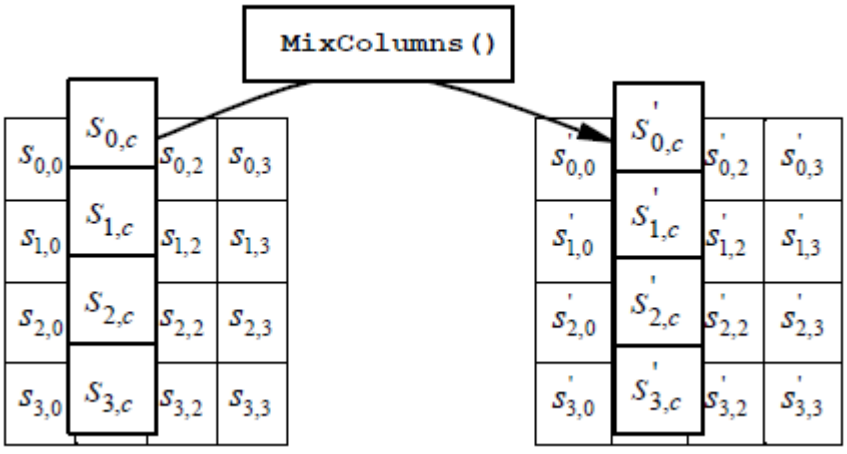


Figure 2.4: MixColumns operates on the state column by column

$$\begin{bmatrix} s'_{0,c} \\ s'_{1,c} \\ s'_{2,c} \\ s'_{3,c} \end{bmatrix} = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \begin{bmatrix} s_{0,c} \\ s_{1,c} \\ s_{2,c} \\ s_{3,c} \end{bmatrix}$$

Figure 2.5: Matrix for multiplication in MixColumns transformation

2.1.4. AddRoundKey Transformation

In this transformation, each byte of the round key which is generated in key expansion section of the cipher is added to the state by simple bitwise XOR operation.

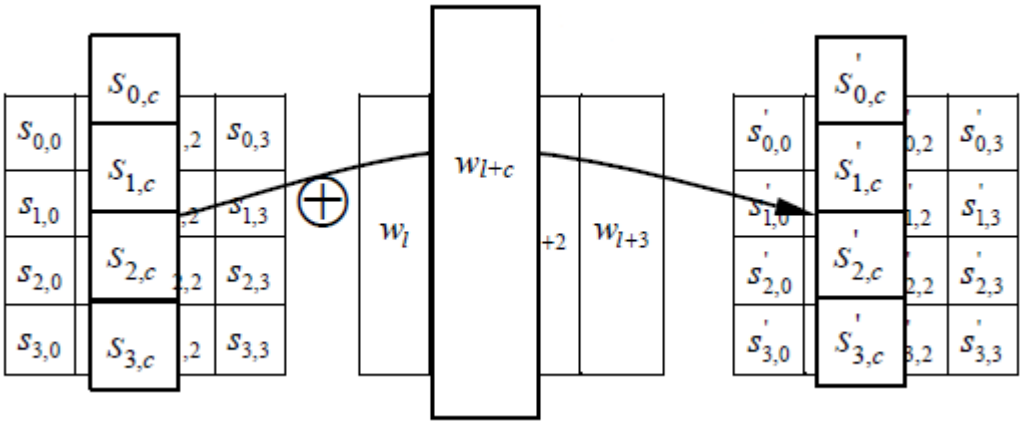


Figure 2.6: AddRoundKey XORs each column of the state with a word from the key schedule

2.2. Key Schedule

Each round key is derived from the cipher key by means of the key schedule algorithm. This process consists of two components: Key expansion and key selection. For every round of AES, a new round key is needed for AddRoundKey transformation. However, since there is different size of key lengths but one block size; each key schedule algorithm for individual key length is slightly different.

For every key scheduling algorithm, cipher key should be considered as sections of 32 bits (4 bytes). Moreover, there exists a round constant value named as r_i which can be found in a predetermined look up table (Table 2.3) with respect to round counter i. Other than that, there is just XOR operations and basic S-box substitutions for every key schedule algorithm. S-box operations are usually applied to each byte of the

last 4-byte section of the input cipher key. The output cipher key of the current round is sent to the next key scheduling round as an input cipher key.

Table 2.3: Round constants for key scheduling (in hexadecimal format)

i (round number)	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
r_i (constant value)	00	01	02	04	08	10	20	40	80	1b	36	6c	d8	ab	4d

2.2.1. AES-128 Key Schedule

For AES-128, there is nothing special for key scheduling algorithm other than shifting the last 4 bytes to left and taking their S-box substitution which is the same procedure for all key lengths. After that, the individual 4-byte sections are XORed according to the proposed key schedule (Figure 2.7). Key selection part is also trivial since the block size and cipher key length is the same. The result of every round of key schedule is a round key waiting to be used in AddRoundKey transformation with respect to round numbers. To illustrate, each round key is given in Table 2.4 for zero valued initial key.

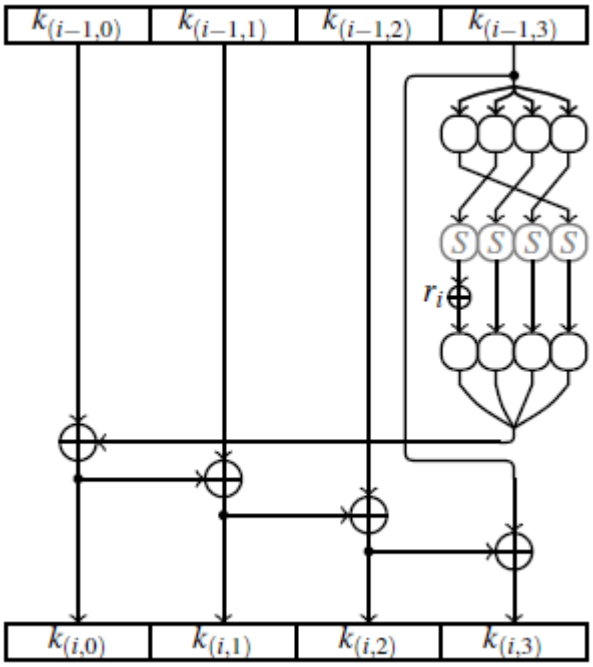


Figure 2.7: AES-128 key schedule algorithm [28]

Table 2.4: AES-128 key schedule test vector for zero valued key

Round Number	Round Key
0	00000000000000000000000000000000
1	62636363626363636263636362636363
2	9b9898c9f9fbfbaa9b9898c9f9fbfbaa
3	90973450696ccffaf2f457330b0fac99
4	ee06da7b876a1581759e42b27e91ee2b
5	7f2e2b88f8443e098dda7cbbf34b9290
6	ec614b851425758c99ff09376ab49ba7
7	21751787355062bacaf6b3cc61bf09b
8	0ef903333ba9613897060a04511dfa9f
9	b1d4d8e28a7db9da1d7bb3de4c664941
10	b4ef5bcb3e92e21123e951cf6f8f188e

2.2.2. AES-192 Key Schedule

For AES-192, key expansion is generally the same with AES-128. The difference lies in the key length. For every round of key expansion, 24 bytes of round key is produced. However, cipher only works on 16 bytes of block size. Ergo, the last two 4-byte sections are passed to the next round. In other words, for each two successive rounds of key expansion, three round keys are produced. To illustrate, each round key is given in Table 2.5 for zero valued initial key.

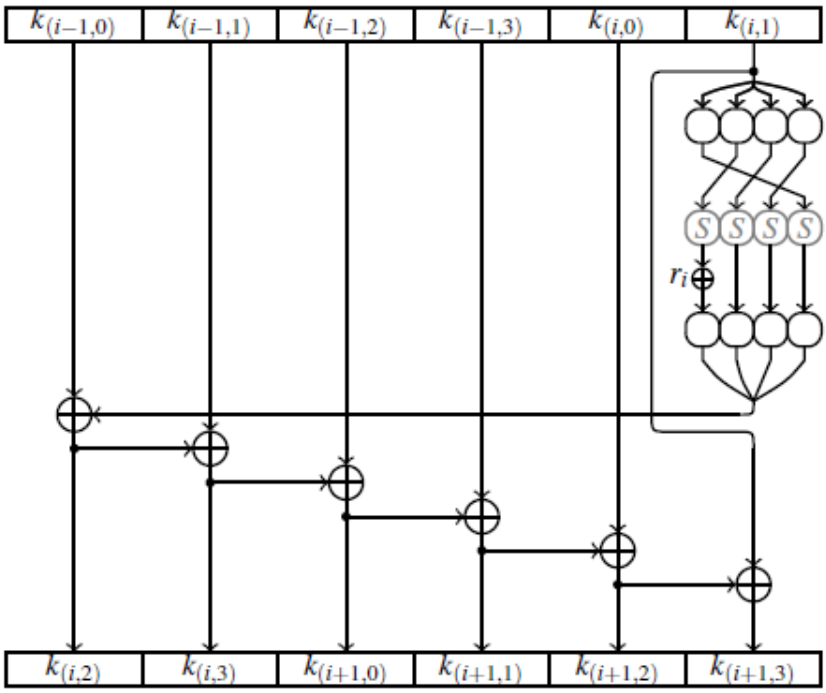


Figure 2.8: AES-192 key schedule algorithm [28]

Table 2.5: AES-192 key schedule test vector for zero valued key

Round Number	Round Key
0	000000000000000000000000000000000000000000000000
1	000000000000000062636362636363636263636263636363
2	626363636263636362636363626363636263636263636363
3	9b9898c9f9fbfbaa9b9898c9f9fbfbaa9b9898c9f9fbfbaa
4	9b9898c9f9fbfbaa90973450696ccffa9b9898c9f9fbfbaa
5	f2f457330b0fac9990973450696ccffa9b9898c9f9fbfbaa
6	c81d19a9a171d65353858160588a2df990973450696ccffa
7	c81d19a9a171d6537bebf49bda9a22c890973450696ccffa
8	891fa3a8d1958e51198897f8b8f941ab90973450696ccffa
9	c26896f718f2b43f91ed1797407899c6891fa3a8d1958e51
10	59f00e3ee1094f9583ecbc0f9b1e0830891fa3a8d1958e51
11	0af31fa74a8b8661137b885ff272c7ca891fa3a8d1958e51
12	432ac886d834c0b6d2c7df11984c5970891fa3a8d1958e51

2.2.3. AES-256 Key Schedule

For AES-256, key expansion is a little bit different than the other key lengths. It has an additional set of operation like the last 4-byte without using round constant. For every round of key expansion, two successive round keys are produced. To illustrate, each round key is given in Table 2.6 for zero valued initial key.

Table 2.6: AES-256 key schedule test vector for zero valued key

Round Number	Round Key
0	000000000000000000000000000000000000000000000000000000000000
1	000000000000000000000000000000000000000000000000000000000000
2	626363636263636362636363626363636263636263636363626363636263
3	aafbfbfbaafbfbfbaafbfbfbaafbfbfbaafbfbfbaafbfbfbaafbfbfbaafb
4	6f6c6ccf0d0f0fac6f6c6ccf0d0f0fac6f6c6ccf0d0f0fac6f6c6ccf0d0f
5	7d8d8d6ad77676917d8d8d6ad7767691aafbfbfbaafbfbfbaafbfbfbaafb
6	5354edc15e5be26d31378ea23c38810e6f6c6ccf0d0f0fac6f6c6ccf0d0f
7	968a81c141fcf7503c717a3aeb070cab7d8d8d6ad77676917d8d8d6ad776
8	9eaa8f28c0f16d45f1c6e3e7cdfe62e95354edc15e5be26d31378ea23c388
9	2b312bdf6acddc8f56bca6b5bdbbaa1e968a81c141fcf7503c717a3aeb07
10	6406fd52a4f79017553173f098cf11199eaa8f28c0f16d45f1c6e3e7cdfe
11	6dbba90b0776758451cad331ec71792f2b312bdf6acddc8f56bca6b5bdbb
12	e7b0e89c4347788b16760b7b8eb91a626406fd52a4f79017553173f098cf
13	74ed0ba1739b7e252251ad14ce20d43b6dbba90b0776758451cad331ec71
14	10f80a1753bf729c45c979e7cb706385e7b0e89c4347788b16760b7b8eb9

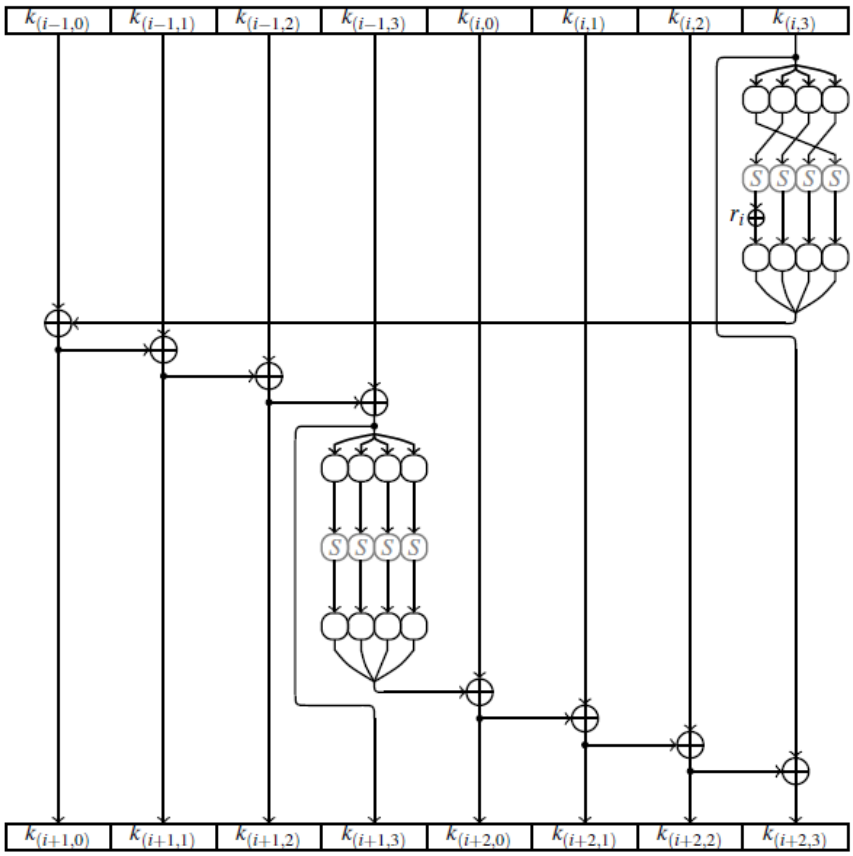


Figure 2.9: AES-256 key schedule algorithm [28]

2.3. Implementation Aspects

There are several ways that AES can be implemented inside a software. These are:

- **No look-up tables:** All transformations of an AES round are calculated in normal way and only SubBytes transformation uses look up tables for S-box calculation. This is the basic AES implementation anyone can think of.
- **Four look-up tables:** The entire round (SubBytes, ShiftRows, MixColumns) of AES is replaced by four look-up tables [29]. Only operation left is adding the round key, addRoundKey. For every round, each byte of state is transformed by using one of the four look-up tables based on the row number where that byte is in the AES state. To illustrate, first row of AES state uses first look-up table, second row uses seconds look-up table and so on. For each resulting AES column, table input bytes are shifted by one byte to the right just like ShiftRows operation only this time every row is shifted. Every result of table look-up is XORed with each other and then XORed with round key. At the end of this process, four columns, each consists of four bytes, are calculated. These columns state the result of round transformations. In Figure

2.10, the first column of an AES state is illustrated. For this column, which is four bytes, four different one-byte table look-ups are used from different parts of the state. Result of each one, which is four bytes as well, is XORed with each other. Table annotations are stated as different colours. The rest of the columns are illustrated as well in Figure 2.11, Figure 2.12 and Figure 2.13.

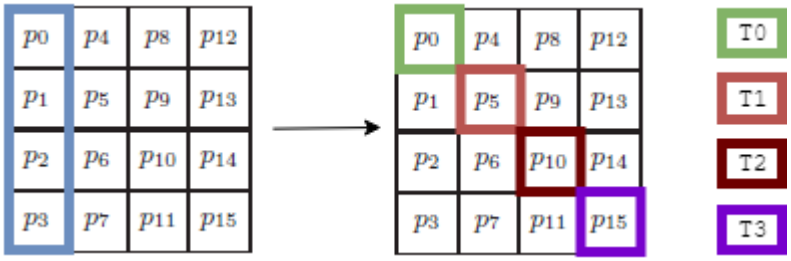


Figure 2.10: Four look-up table implementation for the first column of the state with table annotations

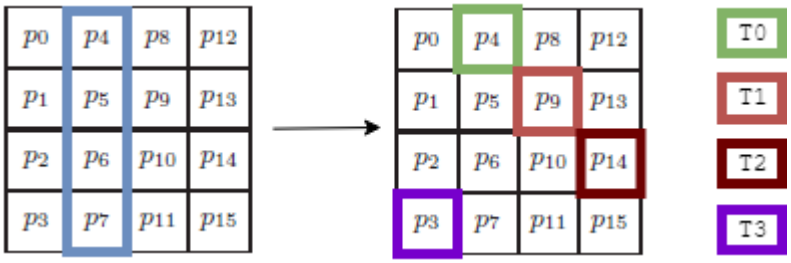


Figure 2.11: Four look-up table implementation for the second column of the state with table annotations

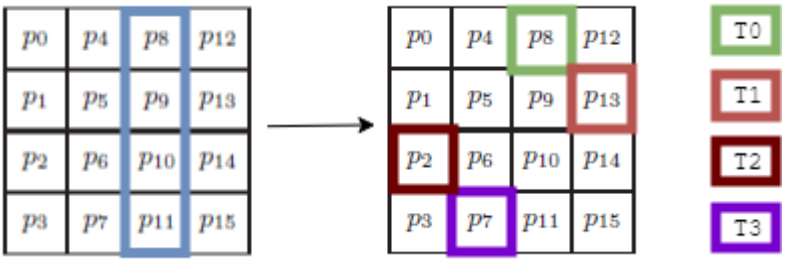


Figure 2.12: Four look-up table implementation for the third column of the state with table annotations

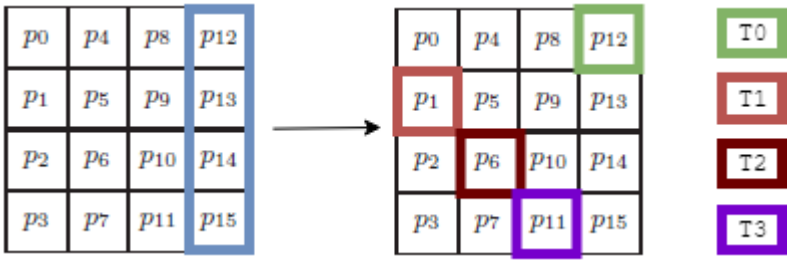


Figure 2.13: Four look-up table implementation for the fourth column of the state with table annotations

- **One look-up table:** This implementation has the same concept of four look-up tables, but it has only one table instead of four [29]. For each round, each byte of state is transformed by using the only look-up table. However, for every row, the output of look-up table is shifted to right by one byte with respect to row number. To illustrate, first row uses the base look-up table while the second row uses one byte right shifted result of look-up table. Third row uses two byte right shifted result and so on. Every result of table look-up is XORed with each other and then XORed with round key just like the four tables look-up implementation. At the end of this process, four columns, each consists of four bytes, are calculated. In Figure 2.14, the first column of an AES state is illustrated. Table annotations are stated with different colours based on bitwise shifting. They are the shifted versions of the initial table. The rest of the columns are illustrated in Figure 2.15, Figure 2.16 and Figure 2.17 respectively.

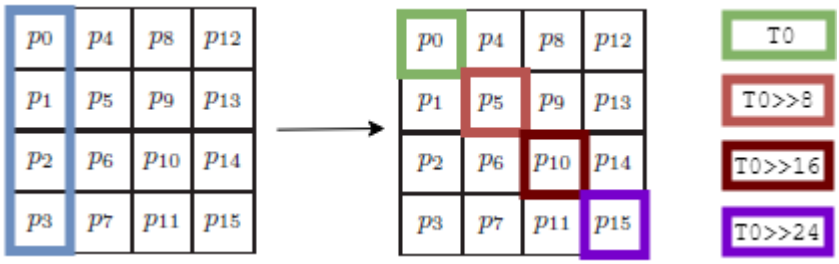


Figure 2.14: One look-up table implementation for the first column of the state with table annotations as bitwise shifting

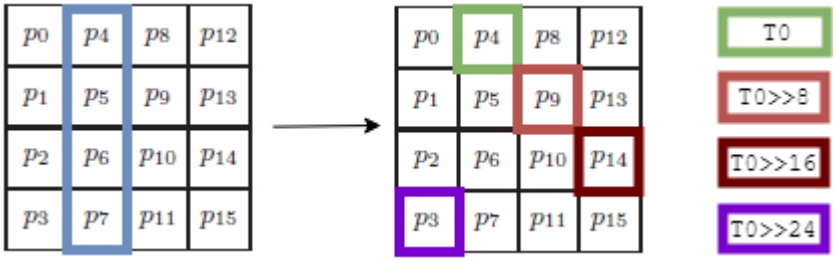


Figure 2.15: One look-up table implementation for the second column of the state with table annotations as bitwise shifting

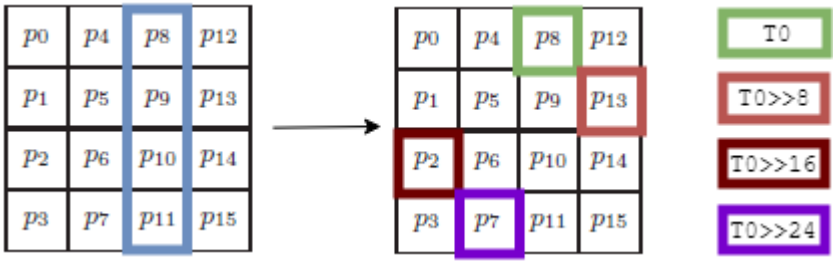


Figure 2.16: One look-up table implementation for the third column of the state with table annotations as bitwise shifting

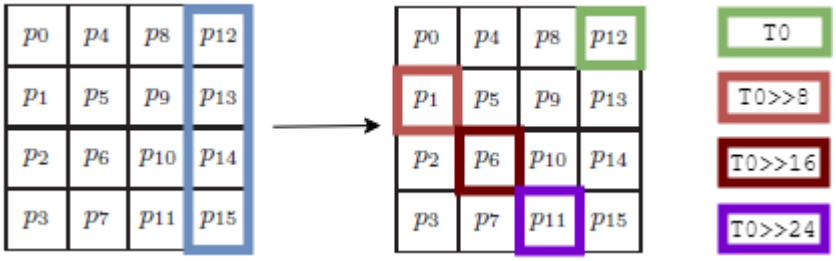


Figure 2.17: One look-up table implementation for the fourth column of the state with table annotations as bitwise shifting

- **AES-NI:** AES-NI architecture consists of instructions for one round encryption and key schedule with full hardware support. Implementation is easy since it can be achieved just by calling instruction functions published by CPU manufacturer. This implementation should be extremely fast concerning the other options since it fully utilizes the hardware. It does not rely on additional look-up tables, working just like the first option. According to CPU manufacturer, published instruction documents must be inspected in order to implement AES-NI version of Intel. In this research, Intel CPU is

used. Implementation is developed through the published paper of Intel AES New Instructions Set [10].

2.3.1. Creating Look-up Tables

There are 4 different look-up tables, each one takes one-byte sized input and gives four-byte sized output. So, they take up for 4 Kbyte of total space. 1 Kbyte space is needed for one table look-up solution since it only relies on one table. In Figure 2.18, table definitions are given. Stated multiplication operator is Galois Multiplication in Galois Field, GF(2^8), explained by the authors of the cipher [9]. S[a] indicates S-box transformation where a is between 0 and 255, which is the available decimal numbers for a byte in computer representation. A shifting pattern can be seen among the tables, in which every table is shifted one byte to the right considering its predecessor. This is the basic concept in the implementation of one look-up table; every table is created from table numbered zero and shifted one, two or three bytes to right according to their number. For instance, table numbered three is generated from table numbered zero by shifting three bytes to right.

$$T_0[a] = \begin{bmatrix} S[a] \bullet 02 \\ S[a] \\ S[a] \\ S[a] \bullet 03 \end{bmatrix} \quad T_1[a] = \begin{bmatrix} S[a] \bullet 03 \\ S[a] \bullet 02 \\ S[a] \\ S[a] \end{bmatrix} \quad T_2[a] = \begin{bmatrix} S[a] \\ S[a] \bullet 03 \\ S[a] \bullet 02 \\ S[a] \end{bmatrix} \quad T_3[a] = \begin{bmatrix} S[a] \\ S[a] \\ S[a] \bullet 03 \\ S[a] \bullet 02 \end{bmatrix}.$$

Figure 2.18: Look-up table definitions [29]

CHAPTER 3

CUDA

3.1. Thread Hierarchy

CUDA allows using thousands of parallel threads with high performance algorithms running on GPUs but in order to understand the efficiency beneath the CUDA, its architecture on GPUs, which is what CUDA is built upon, must be studied. From this point, host keyword will be used for CPU side while device will be used for GPU applications. In CUDA, a function that is called from host and executed N times in parallel by N different CUDA threads on device is called a kernel. Kernels must be defined using global declaration since the initial code will be started from host anyway (Table 3.1). Since C++ version of CUDA is used in this research, every declaration related to CUDA will be C++ version. In C++, kernels launches are specified by using triple angle bracket syntax <<< >>>.

Table 3.1: Kernel Qualifiers for CUDA

Kernel Qualifier	Description
___global___	Exposes the kernel to be called from host (runs on GPU, called from CPU)
___device___	These kernels can only be called from device (runs on GPU, called from GPU)
___host___	These kernels cannot be executed on device (runs on CPU, called from CPU)

A GPU consist of many CUDA threads which are single execution units that run kernels. They can be considered as extremely lightweight CPUs. Each thread is given a unique id by CUDA and has its own registers and private memory. Threads use this unique id to access related memory addresses and make control decisions. Several threads are grouped together, and they form thread blocks which are controlled by SMs. Each SM has memory pool divided between all running threads inside the block. The number of CUDA cores in a SM depends on the GPU. For instance, GTX 970 has 13 SMs with 128 CUDA cores each. CUDA cores can also be called as SP. CUDA needs at least one block for a kernel to be launched. The main advantage of blocks is

that by collecting threads together, they can share memory and perform related tasks together. Each block has a unique identifier given by CUDA as well. There are also grids which are the topmost containers and hold the blocks in an organized way. This thread hierarchy which allows programs to transparently scale to different GPUs is illustrated by Figure 3.1.

In CUDA, the basic execution unit is called the warp. A warp is a collection of 32 threads that are woven together and executed concurrently by an SM. Thus, multiple warps can be executed at once. Each thread in a warp executes the same instruction on different data. Since GTX 970 has 128 CUDA core for each SM, only 4 warps can be executed at a time for a single SM.

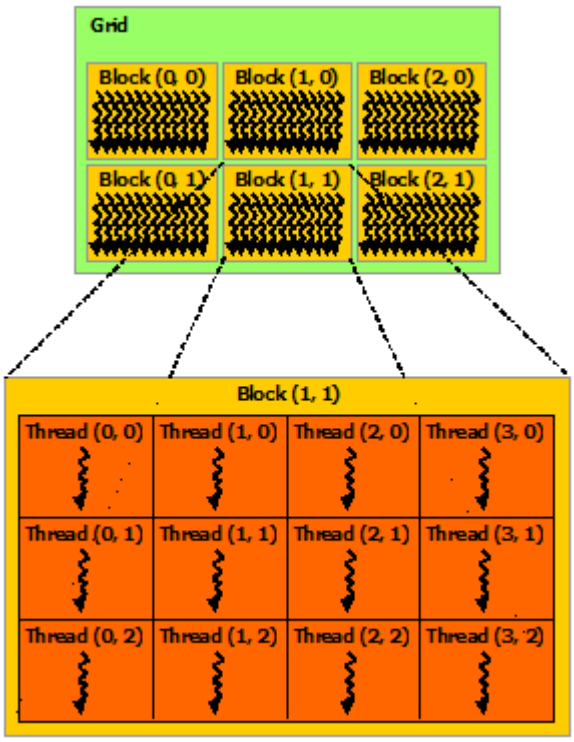


Figure 3.1: Grid of Thread Blocks [30]

In CUDA, kernels are specified with two execution configurations. These are given to kernels with triple angle bracket syntax. The first configuration is the number of blocks that the kernel will be running on and the second one is the number of threads in a thread block. Second parameter which states how many parallel threads will be running for each block should be given as a multiple of 32 in size since threads are executed as warps and it is best to allocate the whole warp rather than wasting some of threads inside a warp. Thus, kernels are initiated with <<<BLOCK_NUMBER, THREAD_NUMBER>>> representation. Considering this, there will be total of BLOCK_NUMBER * THREAD_NUMBER threads across the lifetime of a kernel.

However, not all of them are active at any moment of the kernel. In fact, CUDA manages which blocks and which threads are active.

Sometimes, there would be problems in which every thread needs to access some unique memory space. For these problems, CUDA offers unique thread ids for every running thread on CUDA cores. It is achieved through mapping local thread id to global id by using built-in variables. These variables are started from zero index and can be stated as:

- threadIdx: Thread index within a block
- blockIdx: Block index within a grid
- blockDim: Number of threads inside a block
- gridDim: Number of blocks inside a grid

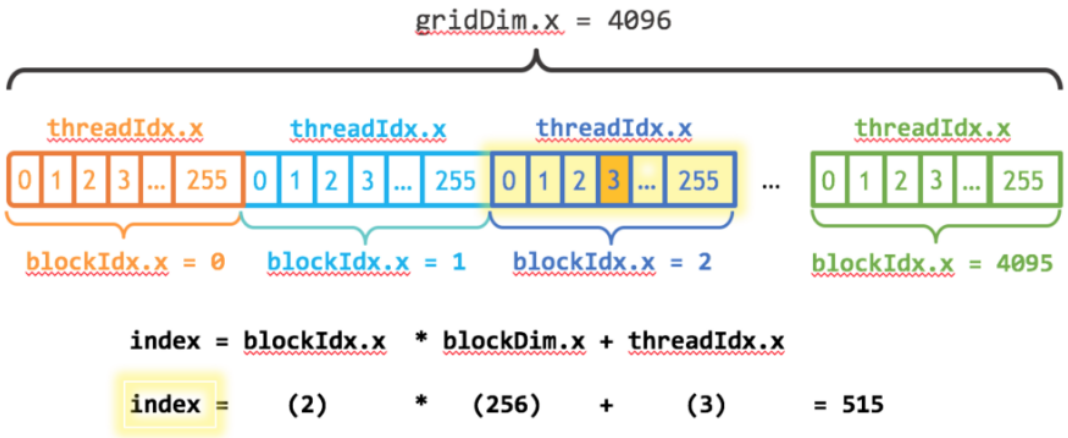


Figure 3.2: CUDA parallel thread indexing pattern example [31]

When a kernel is started, blockDim and gridDim values are determined by execution configurations. Values of blockIdx and threadIdx for a running thread indicate where it is located among the grid. There is a convention for calculating the unique global index for each thread among total threads initiated by kernel. The idea of calculating this unique id resides on each thread getting its index by computing the offset to the beginning of its block. This offset value can be calculated by the block index times the blocks size value, blockIdx * blockDim. After that, the index of the thread within the block, which the current thread belongs to, is added to that value, threadIdx. Ergo, the unique thread index id can be calculated by this idiomatic CUDA phrase:

```
blockIdx * blockDim + threadIdx
```

This calculation is illustrated in Figure 3.2. In this illustration, execution parameters are given to kernel as; number of blocks, gridDim, 4096 and number of threads inside a block, blockDim, 256. Considering the fourth thread inside the third block, unique

global thread id can be calculated as 515, meaning that this is the 515th thread among the total thread space of 1,048,576 which is calculated by gridDim * blockDim.

3.2. Compute Capability

There are several GPUs on the market, and each one has different hardware capabilities. NVIDIA releases GPUs with a specification called compute capability. It is represented by a version number, also called as SM version. This specification determines the general available features supported by the GPU hardware. Some of these features can be listed as thread counts, block counts and memory size. More detailed information such as supported instructions and other related concepts can be found in CUDA Toolkit Documentation [30].

Compute capability and GPU data specifications can be seen in Table 3.2 published by NVIDIA. According to compute capability, SM version also changes. CUDA compiles C++ code with respect to this SM version parameter. Since the value of SM version that GPU can work on is embedded inside the card, changing SM version other than the version of the card is ill advised. For instance, GTX 970 has the compute capability of 5.2 and the code used for this research is compiled with sm_52 parameter. However, sometimes, compiling the kernel code with smaller compute capability versions may reduce register count of kernels or even increase their performance. The same applies for CUDA version as well.

CUDA and compute capability versions should not be correlated with each other. CUDA version is the version of CUDA software platform used by developers to create applications that can be run on many GPU architectures. According to the needs of the developer different versions can be selected for just one card like CUDA 8 or CUDA 9. However, a GPU has only one compute capability version which is the indication of its hardware architecture.

Table 3.2: CUDA compute capabilities [32]

Compute Capability	**5.0**	**5.2**	**5.3**	**6.0**	**6.1**	**6.2**
SM Version	sm_50	sm_52	sm_53	sm_60	sm_61	sm_62
Threads / Warp	32	32	32	32	32	32
Warps / Multiprocessor	64	64	64	64	64	128
Threads / Multiprocessor	2,048	2,048	2,048	2,048	2,048	4,096
Thread Blocks / Multiprocessor	32	32	32	32	32	32

Shared Memory / Multiprocessor (bytes)	65,536	98,304	65,536	65,536	98,304	65,536
Max Shared Memory / Block (bytes)	49,152	49,152	49,152	49,152	49,152	49,152
Register File Size / Multiprocessor (32-bit registers)	65,536	65,536	65,536	65,536	65,536	65,536
Max Registers / Block	65,536	65,536	32,768	65,536	65,536	65,536
Register Allocation Unit Size	256	256	256	256	256	256
Register Allocation Granularity	warp	warp	warp	warp	warp	warp
Max Registers / Thread	255	255	255	255	255	255
Shared Memory Allocation Unit Size	256	256	256	256	256	256
Warp Allocation Granularity	4	4	4	2	4	4
Max Thread Block Size	1,024	1,024	1,024	1,024	1,024	1,024
Shared Memory Size Configurations (bytes)	65,536	98,304	65,536	65,536	98,304	65,536

3.3. Memory Hierarchy

CUDA threads may access data from different memory spaces during their runtime. There are various memory concepts inside GPU which allows diverged implementations for kernel needs. These are called: Registers, local memory, shared memory and global memory. Variable memory space specifiers denote the memory location of a variable inside the kernel. They are listed in Table 3.3.

Registers are the basic data holding structures that are part of CUDA cores. They are extremely fast and resides on top of memory hierarchy. Each thread has a dedicated number of registers allocated while kernel is running. It is important to optimise register numbers for kernels since it is scarce per thread. Maximum number of registers per thread is determined according to compute capability. For instance, for compute capability of 5.2, each thread has 255 32-bit registers [33].

The next memory space available to threads after registers is the local memory of CUDA cores. Local memory space resides in device memory and not on the chip of CUDA cores so accesses to local memory have high latency and low bandwidth.

However, local memory is organized such that consecutive threads can access it in a coalesced way. This make it advantageous as long as all threads in a warp access the same relative address, like the same index in an array variable. Each thread has 512KB of local memory [33]. There are different caching mechanisms for various compute capabilities.

Table 3.3: CUDA memory space specifiers for variables

Memory Space Specifier	**Description**
___device___	Resides in global memory space (VRAM) and is accessible from all threads within the grid
___constant___	Resides in constant memory space and is accessible from all threads within the grid
___shared___	Resides in shared memory space of a thread block and is accessible only from the threads within the block

Shared memory is allocated on the chip, so it has much lower latency and much higher bandwidth considering local or global memory. It is accessible only by threads within the block and exists just for the lifetime of it. So, shared memory is specifically designed for the jointly usage of threads inside a block. In order to achieve this high bandwidth, shared memory is divided into identical memory modules which are called banks.

Banks can be accessed simultaneously according to implementation. Any shared memory read or write accesses can be made in a simultaneous way if each of these accesses is requested to the different memory bank sections of the shared memory. To illustrate, considering making $\boldsymbol{n}$ address requests to shared memory, every request can be serviced simultaneously in an instant if they fall in $\boldsymbol{n}$ distinct memory banks, demonstrated in Figure 3.3. However, if two addresses fall into the same memory bank, there happens a conflict and accesses must be serialized which is illustrated in Figure 3.4. There is another case in which all threads request from the same bank. This is considered as broadcasting and causes no additional bank conflicts since result of the request is broadcasted and shared with each thread. GPU hardware splits the memory requests with bank conflicts into many separate conflict-free requests. This decreases the overall throughput of the application since serialized memory accesses causing other threads to be idle. Therefore, to get maximum performance from a kernel while using shared memory, memory addresses must be mapped to different memory banks in order to minimize bank conflicts.

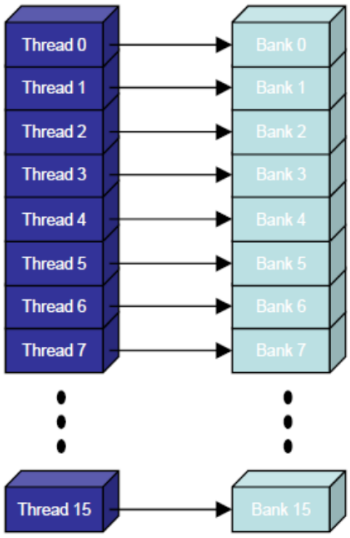


Figure 3.3: Memory access pattern without bank conflicts [34]

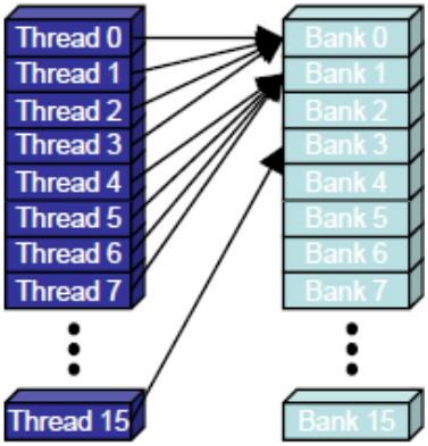


Figure 3.4: Memory access pattern creating bank conflicts [34]

Global memory resides at the bottom of the memory hierarchy. It is accessible by every thread on the grid as well as by the host. It is mostly used to share data between host and device. It is large and resides off-chip, so it is slow considering shared memory. Global memory is persistent across kernel launches so it is not reset after the kernel is finished. This allows data sharing between kernels.

Another concept regarding memory allocation is caching. Understanding cache structure is important in terms of achieving high performance. All modern CUDA capable cards have these two hardware managed caching systems: L1 and L2. L1 cache is bound to each SM individually. To illustrate, there are 13 L1 cache onboard a GTX 970 GPU since there are 13 SMs. Each SM in GTX 970 has 4 warps, 128 CUDA cores, and every warp inside a SM has access to the same L1 cache. L2 cache, on the other hand, stands in front of global memory of GPU. Every access to global memory can

be cached through L2 cache to be used by any SM. So, an access made by an SM can trigger the caching mechanism of L2 to be used by different SMs.

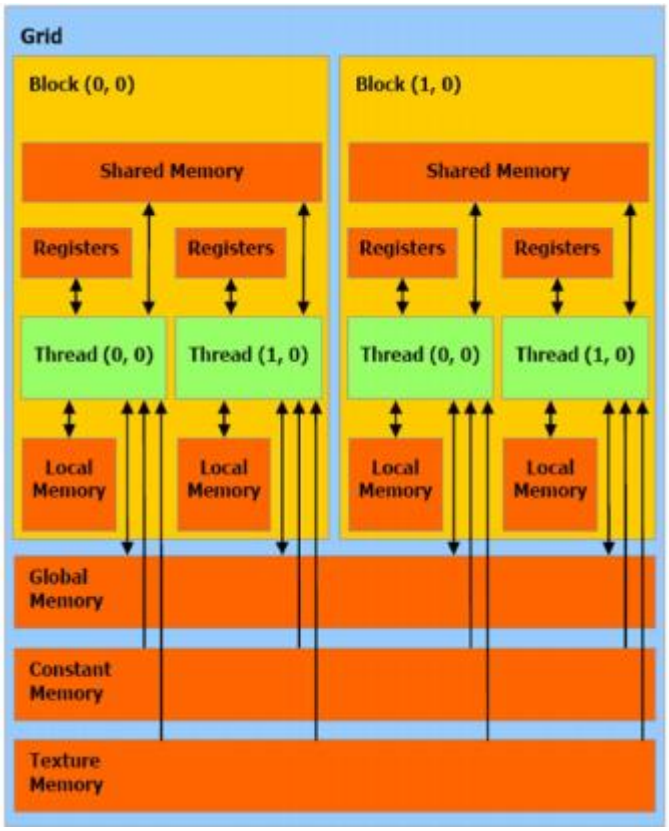


Figure 3.5: CUDA memory hierarchy [35]

If any data is received to a CUDA core through L1 or L2 cache, it is called a cache hit. CUDA holds a percentage statistic, cache hit rate, regarding whether each memory access hits the cache or not. Basically, it is the number of cache hits divided by total number of memory requests. It is crucial to keep this percentage high in terms of performance since every access request through caching is faster than actually retrieving the data from memory.

CUDA offers a special memory model called unified memory which is introduced in CUDA version 6. It can be defined as a shared managed memory space between RAM and VRAM, illustrated in Figure 3.6. Basically, it is a single memory address space accessible from any processor inside the system. Unified memory offers a single pointer to data that can be read or written from code running on either CPUs or GPUs. It eliminates the unnecessary data movement routines between RAM and VRAM. The underlying system inside the card manages data access within a CUDA program without the explicit copy calls. This is beneficial in terms of simplifying development by providing more straightforward integration between CPU and GPU. In order to use unified memory, a GPU with SM architecture 3.0 or higher must be used.

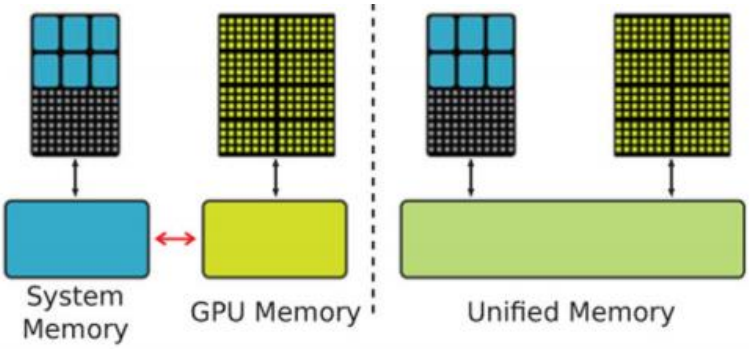


Figure 3.6: Traditional and unified memory access models [36]

3.4. Occupancy

Occupancy is a measure of thread parallelism in CUDA programming. It can be defined as the ratio of active warps to the maximum number of warps supported on SM. Higher occupancy values brings about more computational power for SM. Occupancy is negatively affected by several aspects, which disrupts the efficiency of the kernel. Sometimes, CUDA threads can experience latency issues. This brings about some threads waiting on other threads to finish their jobs since they are all connected to each other as a warp. As a result, kernels can experience under-occupancy states in which there happens to be an insufficient parallelism inside the code. In this case, the performance of the overall kernel can be improved by redesigning the code to use less of the limited resources. Afterall, the resources allocated to each CUDA thread are precious and must be handled with care.

There are several resource limits that cap occupancy of a kernel. Threads per block and registers per thread values can be given as examples to these limitations. Each multiprocessor on GPU has a set of available registers available to be used by CUDA threads. These registers are shared between different thread blocks executing on SM. CUDA compiler aims to minimize the register usage of warps to maximize the number of thread blocks that can be active on GPU. However, it is unwise to entirely rely on CUDA compiler. Developers must optimize kernels to maximize occupancy to some extent. NVIDIA publishes a document called CUDA occupancy calculator [32] in order to assist developers on maxing out occupancy values. Threads per block and registers per thread values are given along with compute capability of the card to this document. According to SM version, document tells you the basic occupancy calculation of the kernel. An illustration for GTX 970 is given in Figure 3.7 and Figure 3.8. In this context, kernel is started with parameters: 1024 threads per block and 64 registers per thread. Since GTX 970 has 64 warps, given parameters most likely result in half of the occupancy on warps. Decreasing registers per thread value vastly increases the resulting occupancy of the kernel as can be seen in Figure 3.8. However,

it is not easy to just decrease registers allocated per CUDA thread since it mostly requires a drastic design change inside the code.

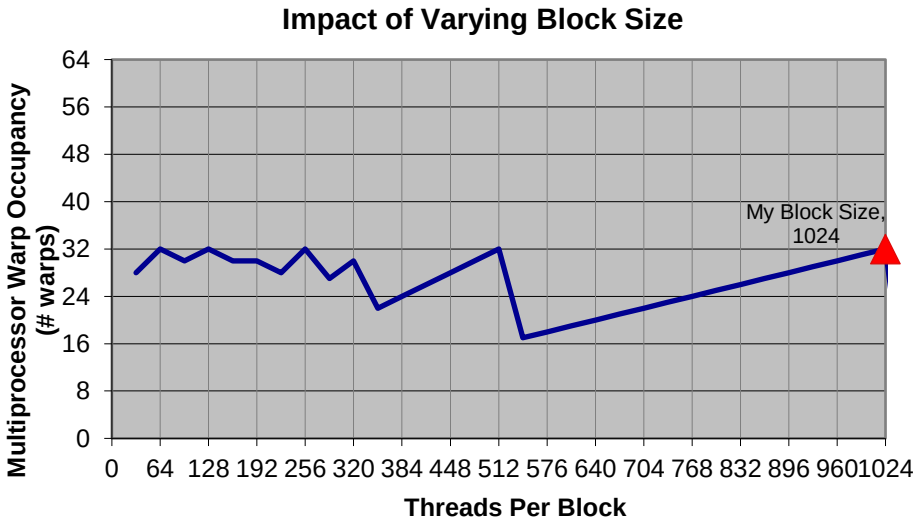


Figure 3.7: CUDA occupancy calculator for block size [32]

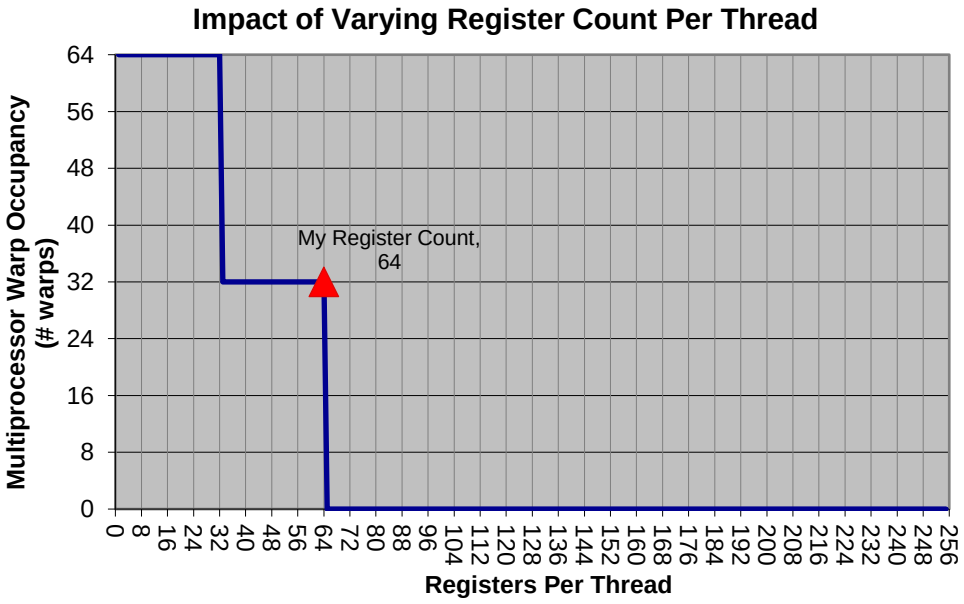


Figure 3.8: CUDA occupancy calculator for registers per thread [32]

CHAPTER 4

CUDA IMPLEMENTATION OF AES

4.1. Exhaustive Search

For comparison between CUDA, AES-NI and C++; we developed different software implementations for each of them. AES-NI one uses new instruction set of Intel while C++ is developed on table look-up implementation of AES. In order to fairly compare CPU and GPU codes, CPU implementations (AES-NI and C++) are designed with multithreading paradigm, which means that several threads can be started for the exhaustive search task at the same time. However, started thread number is inbound by the threading capacity of CPU. Since there are 4 cores inside i7 4470k, CPU implementations will be started with 1, 2, 4 and 8 threads in order to fully extract the CPU power of the system. This brings about CPU utilization issues in which 8 threads version cannot perform in the same efficiency as 4 threads.

Each GPU and CPU implementation have 128 bits, 192 bits and 256 bits versions of AES and each of them is run for 25 times with 10 seconds of intervals to calculate how many seconds it takes to find the correct key. If a key is found; it is written to standard output in each implementation. 6 CUDA kernels with different specifications are created for AES-128 exhaustive search. Each of them is tested for 2^{32} keys and the best among them is selected to be compared with CPU implementations. This GPU and CPU comparison is tested for 2^{30} keys and according to the results, CUDA is by far the fastest implementation on the case of exhaustive search.

4.1.1. AES-128 Implementations for CUDA

In order to optimize the overall output of CUDA implementation, six different kernels on exhaustive search for AES-128 are prepared and compared with each other. Each of these implementations have a special attribute, which differentiates them from others, related to software implementations like using four tables or just one. Ergo, they are only separated with respect to design specifications. AES implementation parts of each of them are identical. According to the produced output from these implementations, the fastest one of them is selected and implemented for AES-192

and AES-256. So, not every one of AES-128 kernels are implemented for 192 bits and 256 bits structures.

Table 4.1: AES-128 kernel function properties

AES-128 Kernel	**Registers**	**Shared Memory (bytes)**	**Constant Memory (bytes)**
Kernel #1	32	5,176	400
Kernel #2	32	2,104	376
Kernel #3	32	33,848	376
Kernel #4	32	33,848	376
Kernel #5	32	41,016	376
Kernel #6	32	36,920	400

Threads of each kernel first populates the allocated shared memory of warps from given inputs like tables and S-box for AES implementations. In this part; syncthreads function for CUDA is used for joining each thread inside a warp in order not to start execution before the population of shared memory is finished. After that, since this is an exhaustive search implementation, different keys must be tried in order to check whether the given ciphertext can be produced by encrypting the given plaintext with the current key. Each thread receives an integer value called thread range which is calculated before the kernels are started. This value states how many encryptions each thread must do in order to fulfil the overall intended range of current iteration. For instance, if 2^{10} keys are wanted to be tried for exhaustive search for 2^5 threads, each thread must do exactly 2^5 encryptions. So, according to this thread range number, calculating the initial key value is done. As explained in CUDA section, each thread has a unique thread index value for identification. According to this identification number and thread range, every thread knows which portion of keys to try among the overall intended key count, no more and no less. Thus, initial key for each thread is calculated and the encryption process is started. At the end of encryption, given ciphertext, residing in shared memory, is compared with the produced ciphertext. If the comparison is a success, tried key is printed out as key is found for exhaustive search. After this encryption cycle is finished, each thread increments its own key by one in order to try new keys to find the real one until the thread range is reached. This whole process is the same for all kernels.

Each of these kernels have 32 registers to be operated on by CUDA. As stated in Figure 3.7, 32 is a special number for GTX 970 card. It allows kernel to use every warp available in an extremely effective way while achieving %100 occupancy rate. So, every kernel reaches its full potential in terms of computational power. Moreover,

shared memory is changed according to design principals of the related kernel. On the other hand, constant memory mostly stays the same for each of them. To elaborate more on the kernels and their different design principals:

- **Kernel #1:** Basic implementation with 4 tables and usual S-box
  - 4 tables in shared memory (4 x 256 integers)
  - S-box in shared memory (1 x 256 integers)
  - No additional shifting is needed for tables
- **Kernel #2:** Implementation with 1 table and usual S-box
  - 1 table in shared memory (1 x 256 integers)
  - S-box in shared memory (1 x 256 integers)
  - Arithmetic shifting for tables is done with usual instructions
- **Kernel #3:** Implementation with 1 extended table and usual S-box
  - 1 extended table in shared memory (1 x 256 x 32 integers)
  - S-box in shared memory (1 x 256 integers)
  - Arithmetic shifting for tables is done with usual instructions
- **Kernel #4:** Implementation with 1 extended table and usual S-box
  - 1 extended table in shared memory (1 x 256 x 32 integers)
  - S-box in shared memory (1 x 256 integers)
  - Arithmetic shifting for tables is done with byte_perm CUDA function
- **Kernel #5:** Implementation with 1 extended table and partly extended S-box
  - 1 extended table in shared memory (1 x 256 x 32 integers)
  - S-box is partly extended in shared memory (1 x 256 x 8 integers)
  - Arithmetic shifting for tables is done with byte_perm CUDA function
- **Kernel #6:** Implementation with 1 extended table and 4 shifted S-box tables
  - 1 table in shared memory (1 x 256 x 32 integers)
  - 4 S-box tables in shared memory (4 x 256 integers)
  - Arithmetic shifting for tables is done with byte_perm CUDA function

For every kernel, 128 bits of plaintext or ciphertext is represented as 4 integers of 32 bits. So, every kernel receives 4 integers of plaintext, ciphertext and encryption key for each of them. This initial approach is selected because of efficiency. Should the representation be selected as character array of 16 bytes, each consists of 8 bits; operations like shifting or XOR would take a lot of time due to traversing the array every time. However, in this way, with 4 integers design; it is easier and highly efficient to make basic operations like XOR since these operations work faster on integers than a stream of character array.

Each kernel has 10 integers as round constants and ciphertext of 4 integers stored in shared memory. This design is the same for every one of them. They do not cause any

bank conflicts since each thread uses the same constants while requesting from the same bank location in shared memory, so data is broadcasted to each thread. Round constants are used for key scheduling while different keys are tried and ciphertext is used for comparing whether the tried key is the right one or not. Round constants are needed for every iteration of different keys while at the end of each iteration, produced ciphertext needs to be compared to the initial given one. Accessing to them needs to be faster in order not to compromise the overall efficiency. This is the reason that they are stored in shared memory rather than global memory. There was a faster option in which they are stored in registers but storing them in thread registers disrupts register count of kernels since registers are scarce and more used registers means lower occupancy rates which is explained in occupancy section of CUDA.

Kernel #1 is just the basic AES-128 implementation with 4 tables on CUDA. These tables are allocated in shared memory which makes read and write operations extremely efficient for warps. S-box is stored in shared memory as well. Other than that, there is nothing special about this kernel. Kernel #2 is the improved version of the first one in which #2 is implemented as having 1 table for AES operations rather than 4. One table implementation, explained in implementation section of AES as one-lookup table, must use arithmetic shifting in order to compensate the loss of other 3 tables. In this way, kernel #2 only has 1 table and S-box stored in shared memory. So, it uses fewer shared memory than kernel #1. However, each look-up operation must use arithmetic shifting for one byte, 8 bits, shift to right with respect to the location of byte. The first row does not need any more shifting, but the second row must be shifted by 1 byte. The third one needs to be shifted by 2 byte and the fourth one must be shifted 3 bytes. This shifting operation consists of 2 shifts and 1 AND in terms of integer instructions so total of 3 operations are performed for per shifting.

Kernel #3 has the same design, one table implementation and S-box, as does kernel #2. However, it has a special configuration for one look-up table as it is expanded in order to be less effected by bank conflicts. This expansion consists of duplicating every row element of look-up table 32 times which can be considered as creating a 2D array instead of the original 1D version. In this way, every element of 2D look-up table is the same one. This approach is taken in order to decrease the encountered bank conflict counts which delays threads from accessing the shared memory. Kernel #4 is basically the same one as kernel #3 with the only difference being the shifting operation of table look-ups is done by byte_perm CUDA function instead of usual instructions which consist of two bitwise shifting and one bitwise and. This CUDA function, byte_perm, decreases the overall shifting counts achieved by the kernel which makes it more optimized than arithmetic shifting with usual instructions.

Kernel #5 is a variation of kernel #4 with S-box being partly expanded as 8 times. The original expansion of kernel #3 is done as 32 times. However, there is a limit to how much shared memory can be used and expanding S-box along with one look-up table does not fit into the shared memory of GTX 970. Due to this reason, S-box is partly expanded in order to further decrease bank conflict count. Expanding 8 times does not bode as well as 32 times but it does help on the bank conflict problem.

Kernel #6 is also a variation of kernel #5, but it has 4 different S-box tables residing in shared memory. Each one of them only allocates a certain byte of 4-byte S-box element. The other bytes are allocated as zero bytes. To elaborate further, S-boxes in other kernels have the same byte for 4-byte table elements while this kernel has indexed S-boxes. For instance, the first element of one S-box implementation is 0x63636363 just like the cases in other kernels. However, this kernel has 4 different S-boxes with each allocating just one byte like 0x00000063, 0x00006300, 0x00630000 and 0x63000000. The reason behind this lies in the unnecessary AND operation while using S-boxes. Considering the implementation aspects of AES, each byte of the state is calculated independently. Since every row is stored as a 4-byte integer, any operation done on any of the bytes is directly affecting other bytes unless the other byte values of that particular row are zero due to the fact that zero value is considered ineffective element of XOR operation.

In order to investigate which of the kernels is the fastest one, implementations are run for exhaustive search of 2^{32} keys. Every CUDA implementation in this research is run with 1024 blocks and each block contains 1024 threads. The reason behind this decision lies in Figure 3.7 in which the best number of threads per block is stated as 1024. Each iteration is run 25 times with 10 seconds of intervals between them. The results are obtained as how many seconds for exhaustive search to be finished (Figure 4.1) and averaged in order not to be affected by one bad iteration or a very good one. According to them, kernel #5 appears to be the fastest one among others. Some of the kernels are similar in terms of duration because they share similar designs like #1 and #2 or #3 and #4. Some of these architectural changes affect the duration in a drastic way like extending the table for kernels #2 and #3 while other ones affect only a little like adding byte_perm function for shifting for kernels #3 and #4.

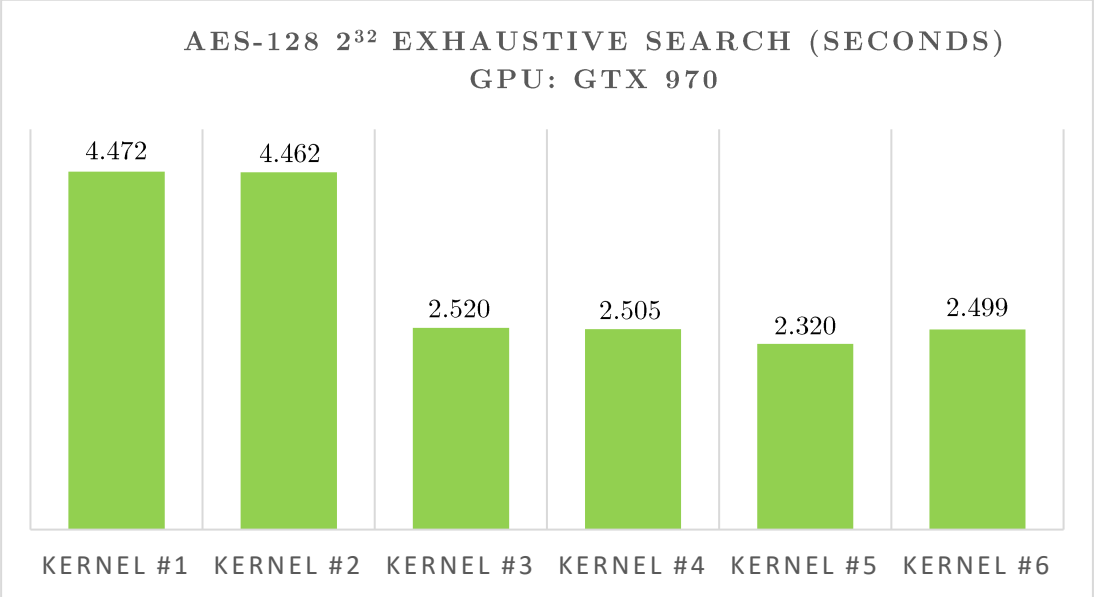


Figure 4.1: AES-128 CUDA 2^{32} exhaustive search results in terms of seconds

In order to test performance differences between the kernels, NVIDIA Nsight software is used. Nsight is a system wide performance tool designed to visualize optimization opportunities and help tuning the code efficiency for GPUs [37]. It basically provides insight to developers for further optimizing their software. For this research, Nsight with version of 5.6.0.18099 is used. Nsight can give occupancy rates, instruction statistics, issue efficiency, achieved IOPS values and memory statistics as well as other important information related to kernels. For this purpose, every CUDA kernel is consecutively run for exhaustive search of 2^{27} keys in Nsight software. The obtained results are inspected to extract information on the efficiency of kernels and compared with each other.

4.1.1.1. Occupancy

Occupancy is basically the ratio of active warps on an SM to the maximum number of warps supported by the SM [38]. In other words, it is the indicator for specifying how much performance extracted from the GPU. Low occupancy results generally indicate poor instruction efficiency because there are not enough eligible warps to hide the latency between instructions. However, increasing occupancy further does not always lead to increased performance due to the reduction in resources per thread.

During the Nsight performance test, total duration times of kernels are recorded which can be found in Table 4.2. In terms of how many seconds passed for each kernel, duration values fairly resemble the results obtained in Figure 4.1. Every kernel has the register count of 32 (Table 4.1) which makes every kernel stays inside register

limit stated in Figure 3.8. For this reason, every kernel has occupancy values higher than 90% (Table 4.2). Register block limit values also states that every kernel has little blocking factors in terms of registers. This proves that kernels are thoroughly optimized for performance. However, since register block limit is not 0, kernels still have potential to be improved.

Shared memory block limit basically states how much SMs are blocked with respect to shared memory usage. If this factor is high, shared memory of the kernels can be reduced in order to increase occupancy. For the case of kernels #1 and #2, block limits are quite high meaning the performance of these kernels are hindered by their shared memory usage. In order to overcome this situation, kernels #3, #4, #5 and #6 have extended tables residing in shared memory. This extension process drastically increases their shared memory usage but allows the kernels to decrease their bank conflict values. In this way, kernels are improved almost two times in terms of duration.

Kernel #1 and #2 have the best occupancy rates but this does not mean that they are the fastest among others. As a matter of fact, they are the slowest two of the six tested kernels. The reason behind this lies in the different structures of the kernels which allows using resources more efficiently with lower occupancy rates just like the table extension case of shared memory. Active warps values are similar to occupancy ratios in which occupancy is calculated by the ratio of active warps to maximum supported warps which is 64 for GTX 970.

Table 4.2: CUDA Nsight occupancy results for 2^{27} AES-128 keys

AES-128 Kernel	**Duration (seconds)**	**Occupancy**	**Active Warps**	**Register Block Limit**	**Shared Memory Block Limit**
Kernel #1	0.1298	92.71%	59.33 / 64	2 / 32	18 / 32
Kernel #2	0.1313	92.72%	59.34 / 64	2 / 32	42 / 32
Kernel #3	0.0698	92.27%	57.77 / 64	2 / 32	2 / 32
Kernel #4	0.0703	91.3%	58.43 / 64	2 / 32	2 / 32
Kernel #5	0.0696	91.51%	58.56 / 64	2 / 32	2 / 32
Kernel #6	0.0698	91.18%	58.35 / 64	2 / 32	2 / 32

4.1.1.2. Instruction Statistics

This section provides assessment of overall utilization of the GPU while executing the kernels. It provides answers to questions like how efficient kernels are and how close

their instruction throughput is to the peak performance of the GPU [39]. GPU works in 32 threads of warps. GTX 970 has 4 warp schedulers stated as warp allocation granularity in Table 3.2 and at every instruction, warp schedulers select one warp which has 32 threads that can make forward process. However, a warp scheduler might need to issue an instruction multiple times to complete the execution for all 32 threads of a warp. This can happen for two primary reasons: First, address divergence and bank conflicts on memory operations and secondly, special instructions that can only be issued for half-warp per cycle and thus need to be issued twice. This brings about two additional definitions to instruction statistics which are called as instructions issued and instructions executed. Instructions issued is the total number of times warp schedulers issue a single instruction while instructions executed is the value of how many instructions are successfully executed for every 32 threads inside a warp. Issuing an instruction multiple times is referred as instruction replay which impacts the performance of kernel execution. Each replay takes away the ability to make forward progress by issuing new instructions on warp scheduler. Ergo, it is important to have the same values for issued and executed instructions making every issued instruction is successfully executed with no waste.

IPC is the average number of instructions per cycle. Issued and executed IPC values for every kernel is given in Table 4.3. Every kernel has equal issued and executed IPC value which is the ideal case in which no warp scheduler wastes any instruction. Kernel #1 and #2 have lower IPC values considering other ones. The gap between #2 and #3 is solely based on extending the table in order to decrease bank conflicts. This is the reason that kernels #3, #4, #5 and #6 have that boost over #1 and #2. It allowed warp scheduler to issue more instructions per cycle. Partly extending S-box, for the kernel #5 case, increases IPC value as well. So, fully expanding S-box most probably would bring bigger IPC values than the current one of kernel #5. However, shared memory of GTX 970 is not enough to make such optimization, so; S-box is only partly extended. To the best of our knowledge, currently there are no NVIDIA GPUs that come with more than 64 KB shared memory. Some GPUs have more than 64 KB shared memory which can be seen in *Table 3.2* as compute capability 5.2 has 98,304 bytes. However, these GPUs do not allow developers to use the those extra 32 KB space since it is reversed for GPU usage for caching. Furthermore, IPC values of kernel #3 and #4 are the same which means byte_perm function of CUDA does not change the total instruction count. However, there is a slight performance gain which can be observed in Figure 4.1. Ergo, it can be said that using byte_perm function for shifting purposes helps increasing the overall performance. Having 4 different S-boxes also increases performance like in the cases of kernel #3 and #6. By having extra S-boxes, kernel #6 eliminated unnecessary AND instructions which can be seen in Table 4.3 as having lower IPC value with respect to kernel #3. This improved the overall

performance of kernel #6 which can be seen in Figure 4.1 as well. However, the performance gain is not big considering the result of kernel #3.

SM activity is the indication for active percentage of multiprocessors. A multiprocessor is considered as active if at least one warp is executing. SMs can be inactive if there exist high workload imbalances between blocks. Some SMs can go idle while the kernel is waiting to be finished. This is not the ideal case in which the performance is hindered by idle SMs. Every kernel has high SM activity ratio since they all have high occupancy values. This means that SMs are hardly idle in terms of execution. Kernel #5 is the most active one according to the results, which corresponds to it being the fastest kernel among them (Figure 4.1).

IPW value in Table 4.3 shows the value of average instruction per warp for SMs. Higher amounts indicate executing more instructions per warp. IPW is very useful to understand the SM activity. To illustrate, kernel #5 has the highest SM activity just as the case of IPW. Nsight offers IPW value of each SM as well as the average of them. High variations in the IPW metric across the SMs indicate non-uniform workloads for the blocks of the kernel. Such imbalances most of the time result in low performance values. However, in every kernel tested, every SM shows very similar IPW values regarding other SMs. This indicates that every SM is actively executing its instructions while not hindering the overall execution, which will not create any imbalanced situations.

Table 4.3: CUDA Nsight instruction statistics for 2^{27} AES-128 keys

AES-128 Kernel	**Instructions Per Clock (IPC) (Issued / Executed)**	**SM Activity**	**Instructions Per Warp (IPW)**	**Warps Launched**
Kernel #1	1.57 / 1.57	99.07%	1,06,077.71	2,520.62
Kernel #2	1.78 / 1.78	99.38%	1,198,98.22	2,520.62
Kernel #3	3.90 / 3.90	98.82%	1,383,36.97	2,520.62
Kernel #4	3.90 / 3.90	99.46%	1,383,36.97	2,520.62
Kernel #5	4.11 / 4.11	99.49%	1,452,52.22	2,520.62
Kernel #6	3.86 / 3.86	98.82%	1,366,76.47	2,520.62

Warps launched values shows the average number of warps launched by SMs. Just like IPW case, Nsight offers the number of warps launched per SM. Large differences in this number mostly results in insufficient amount of parallelism within the grid. However, every kernel has the same number of average warps launched (Table 4.3).

This is because the kernels are perfectly capable of performing in efficiently parallelized ways. So, there is no need to launch additional warps since every warp is successfully completed.

4.1.1.3. Branch Statistics

Branch efficiency is the ratio of executed uniform flow control decisions over all executed conditionals. It can have serious impact on the efficiency of executing a kernel [40]. For instance, the result of some control decisions can diverge into complex executions while others can lead to simple ones. This negatively affects the execution flow since some threads inside the warps are finished while others continue to execute, which decreases IPC and IPW values. However, since AES execution flow is a solid one, meaning there is no diverged flows in terms of computation, Nsight gives %100 branch efficiency on all control flows. This greatly helps in the overall warp efficiency without any additional optimization is done on the code in terms of diverged flows.

4.1.1.4. Issue Efficiency

Issue efficiency section of Nsight provides information concerning the ability of device to issue instructions. If device is not able to issue instructions at every cycle, potential performance of the kernel is hindered. Nsight also provides some issue stall reasons which can be used to optimize the code [41]. These issue stall reasons bring about stalled warps which cannot make forward the progress.

GTX 970 has the theoretical warps per SM value of 64 which cats as the upper limit to active warps. A warp is considered as active if it is scheduled on SM to complete its instructions. In other ways, if the warp is doing something, it is active. Active warps per SM value in Table 4.4 can be explained as the average number of active warps allocated to SMs across the kernel execution. Higher values might allow hiding warp latencies more efficiently which will increase the performance. Eligible warps per SM value, on the other hand, indicates the number of average warps which can forward progress into the next instruction. In other words, eligible warps are the ones which are waiting to be executed in the next cycle. Warps that are not eligible will report an issue stall reason which blocks the execution of that warp in the next instruction cycle. The optimum target for eligible warps is to have at least one eligible warp per scheduler per cycle. However, higher values indicate that warps are quickly finishing their instructions and waiting to be instructed again. Extending the table in kernel #3 with respect to #2 results in increased eligible SM ratios. This means that kernel #3 was able to outperform #2. The same case also holds for kernel #4 and #5 in which partly extending S-box increases eligible warp value.

On every clock cycle, a warp scheduler tries to issue an instruction from one of its warps. When a warp issues an instruction, it takes at least a few cycles before it becomes eligible to issue again. This is also proportional to occupancy of kernel. Warps issue efficiency column of Table 4.4 shows the average values across all warp schedulers for all kernels. No eligible value indicates the number of cycles that warps scheduler could not find warps to select from. This is not the intended case since warp scheduler could not issue an instruction which slows down the kernel. The lower percentage of cycles with no eligible warp value indicates that the code runs more efficiently on the target device. In order to decrease no eligible ratio, issue stall reasons which keeps warps from becoming eligible must be investigated. Moreover, one or more eligible values show the ratio of a warp scheduler that had at least one eligible warp to select. Nsight documents reveal that values with a target of getting close to 100% are better in terms of performance. Just like the rise in warps per SM value between kernels #2 and #3; no eligible percentage of kernel #3 is much lower than #2 due to extended table, which makes kernel #3 a lot faster than #2. This also holds for #4 and #5. Warp issue cycle values, indicating the total warp cycles issued from warps in order to finish execution, also prove this assumption. There is a huge decrease between kernel #2 and #3, also a slight one between #4 and #5.

Table 4.4: CUDA Nsight issue efficiency stat statistics for 2^{27} AES-128 keys

AES-128 Kernel	**Warps Per SM (Active/Eligible)**	**Warp Issue Efficiency (No Eligible/One or More Eligible)**	**Warp Issue Cycles**
Kernel #1	59.37 / 1.53	69.25% / 30.75%	8,847,069,145
Kernel #2	59.33 / 2.00	64.56% / 35.44%	8,847,251,157
Kernel #3	57.69 / 7.69	16.87% / 83.13%	4,645,012,486
Kernel #4	58.45 / 6.34	20.09% / 79.91%	4,643,595,118
Kernel #5	58.60 / 8.54	10.88% / 89.12%	4,643,260,922
Kernel #6	58.34 / 5.93	22.44% / 77.56%	4,643,948,051

Main issue stall reasons which are provided by Nsight in issue efficiency section are listed in Table 4.5. Table also has warp cycle column which indicates how many warp cycles are required to complete the kernels. According to it, execution dependency is the one stalling the warps the most. It is basically caused by warps waiting for an input required by the instruction. Extending the table also greatly decreases execution dependency values which in return speeds up the kernel. Additional extending S-box in kernel #5 also decreases execution dependency stall reason considering the change between #4 and #5. However, doing so create additional stall reasons like pipe busy, resource required by instruction not yet available, and instruction fetch which is

caused by the next assembly instruction is not yet fetched. Still, the performance difference between #2 and #3 is so great that creating additional stall reasons are neglected. Pipe busy stall reason can be considered as it increases when memory operations are accumulated. For instance, 4 table structure in kernel #1 is changed to 1 table structure in kernel #2 which increases pipe busy stall reasons. However, this also increases performance slightly since kernel #2 can be completed in smaller warp cycles. Ergo, increasing some stall reason might result in greater performance.

Table 4.5: CUDA Nsight issue stall reasons for 2^{27} AES-128 keys

AES-128 Kernel	Warp Cycles	Instruction Fetch	Pipe Busy	Execution Dependency
Kernel #1	127,945,964,330	1,266,962,428	948,297,786	61,956,466,874
Kernel #2	126,750,484,614	1,282,631,087	3,647,055,003	63,623,757,543
Kernel #3	57,933,760,551	1,907,006,530	3,145,526,887	17,327,349,043
Kernel #4	60,526,582,607	1,830,931,062	1,900,798,955	23,521,122,301
Kernel #5	57,923,638,478	3,457,882,740	2,415,918,323	20,863,964,657
Kernel #6	60,787,971,042	1,673,265,824	1,809,299,404	20,728,639,606

4.1.1.5. Achieved IOPS

IOPS stands for integer operations per second which is a metric used for investigating integer operations and their instructions. Its primary benefit is tracking and evaluating differences in performance of the code changes [42]. Since one round of AES encryption mostly revolves around XOR and shifting operations, they are the most important aspect in terms of understanding IOPS performance of kernels. According to the arithmetic instructions table [33] provided by NVIDIA for different compute capabilities, 32 bits integer add operation gives throughput of 128 while 32 bits integer shift produces 64 in terms of per clock cycle for compute capability of 5.2. Ergo, add operation is two times faster than shift operation. So, using add operation instead of shift increases the performance.

Kernel #2 uses 1 table while kernel #1 is using 4. This increases shift operations as 1 table needs to be shifted in order to be operated on. This gives a slight performance gain (Figure 4.1) but no so much to make a difference since shifting operation is expensive and adding almost 55% more shifting operation cancels out the performance gain. Moreover, adding extended table for the case of kernel #3 greatly increases add operation number considering kernel #2 but it also vastly improves performance of the kernel as #3 is 1.77 times faster than #2. So, increasing add operation count does not always result in performance hindrance. Another important factor concerning shift

operations exists between kernel #3 and #4. Kernel #4 uses byte_perm CUDA function for arithmetic shifting while kernel #3 uses traditional 2 shifts 1 AND method. As can be seen in Table 4.6, using byte_perm reduces shifting operation count by 35%. However, performance gain from it is not very noticeable as it only offers roughly 1% decrease in duration (Figure 4.1). Furthermore, it decreases occupancy but increases active warps (Table 4.2) and SM activity (Table 4.3). Just like the case of kernel #2 and #3, additional extension of S-box increases add operation count by 28% while decreasing the duration by 8%.

Table 4.6: CUDA Nsight achieved IOPS statistics for 2^{27} AES-128 keys

AES-128 Kernel	ADD Operations	Shifts Operations	ADD per Second	Shifts per Second
Kernel #1	1,630,011,392	26,445,873,164	12.56	203.71
Kernel #2	1,628,438,528	40,941,387,788	12.40	311.78
Kernel #3	20,955,791,360	40,941,649,932	300.15	586.41
Kernel #4	20,955,791,360	26,446,135,308	298.20	376.33
Kernel #5	26,861,371,404	26,580,090,892	385.87	381.82
Kernel #6	20,957,364,224	26,446,135,308	300.37	379.04

4.1.1.6. Pipe Utilization

Each SM of a CUDA device features several hardware units which are specialized in performing specific tasks. At the hardware level, these units are maintained by execution pipelines to which warp schedulers dispatch instructions to. For instance, load/store units save data or fetch from memory. Understanding the utilization of these pipelines help in reaching to peak performance of the device [43]. It also allows bottlenecks caused by overusing a certain type of pipeline.

Nsight offers shared memory, texture and arithmetic utilization ratios. Since no kernels use any texture, it is shown as 0%. Shared memory pipe utilization covers all issued instructions that trigger a request to the memory system of the device. It consists of load/store operations on global, local and shared memory. Since AES-128 kernels only use shared memory, only utilization of shared memory is recorded by Nsight and shown in Table 4.7. According to the obtained results, extending table mechanism in kernel #2 increases shared memory utilization almost two times considering kernel #3. However, additional extension on S-box does not greatly increments utilization ratio in the case of kernel #4 and kernel #5 but it helps in terms of performance, nonetheless. Since there is not any distinctive architectural

change between other kernels, their shared memory pipe utilization values stay the same.

Arithmetic pipe utilization ratio covers floating point and integer instructions. Since AES code base of kernels does not use any floating points, Table 4.7 only shows utilization of integer operations. Just like the shared memory case, extending table also increases arithmetic pipe utilization by 49% in the cases between #2 and #3, 4% for kernel #4 and #5. Considering kernel #3 and #4, adding byte_perm CUDA function for shifting tables decreases arithmetic pipe utilization by 6% but it also decreases shifting operation count by 35% which can be observed in Table 4.6. While arithmetic utilization ratio is dropped, performance is slightly increased considering Figure 4.1. Moreover, adding 4 different S-box tables also causes arithmetic pipe utilization to decrease by 1.5% while offering very little performance gain in kernel #4 and #6 cases.

Table 4.7: CUDA Nsight achieved IOPS statistics for 2^{27} AES-128 keys

AES-128 Kernel	**Shared Memory Pipe Utilization**	**Arithmetic Pipe Utilization**
Kernel #1	37.55%	29.76%
Kernel #2	37.55%	37.31%
Kernel #3	71.52%	86.56%
Kernel #4	71.55%	80.53%
Kernel #5	71.70%	84.28%
Kernel #6	71.55%	78.99%

4.1.1.7. Memory Statistics

Nsight performs several experiments concerning the usage of memory systems during kernel execution. These consist of global, local, shared, caches and buffers statistics. It is important to locate which type of memory is bottlenecking the kernel in order to create more efficient code [43]. Since every kernel uses only shared memory type for look-up tables, it is the main reason for performance stalling and the area for kernels to be improved on. Global memory is only used for transferring data from host to device. For exhaustive search, data transferred from host to device consists of look-up tables, S-box, initial plaintext and ciphertext, round constant array for key scheduling and key range for each thread. They are allocated in both RAM and VRAM at the same time by using cudaMallocManaged function which uses unified memory concept of CUDA. After the data is received by threads, they are immediately stored in shared

memory. Thus, there is not any store operation issued on global memory during the execution, only read operations at the start for each thread.

Global memory statistics are shown in *Table 4.8*. Since the size of data fetched from global memory is so small considering VRAM of the device, it almost every time hits L2 cache. L2 cache stands before the global memory while there is an L1 cache for each SM. So, the architecture of device works like; for a request sent to L1 cache, if it is a hit, the required data is returned immediately. However, if it is a miss, it is forwarded to the L2 cache which is the main cache point of global memory for all SMs. However, L1 cache is more important in terms of memory hierarchy so increasing L1 cache hit rate is far more crucial for performance.

Load requests are directly proportional to design used in kernels. For instance, kernel #1 uses 4 tables and 1 S-box. Fetching them to shared memory requires 352,256 requests. On the other hand, kernel #2 uses just 1 table and 1 S-box. This decreases load requests to global memory. Moreover, changing 4 table structure into 1 increases L1 cache hit rate by almost 3% since threads will request less different data from global memory. These two improvements increase performance a little bit in the favour of kernel #2 which can be seen in Figure 4.1. The same type of decrease can be seen between kernel #4 and #6. Just like kernel #1, #6 uses 4 different S-box along with 1 table which increases global memory load requests and decreases L1 cache hit rate. However, #6 seems a little bit faster than #4 because it eliminates the need of performing extra operations to the result of S-box. This can be seen in Table 4.5 that execution dependency stall reason of kernel #6 is almost 12% lower than of #4. Nonetheless, the performance difference is not so much to make a difference in the end. Therefore, using architectural designs like 1 table instead of 4 and 4 S-box instead of 1 contributes to increasing overall performance a little bit.

Table 4.8: CUDA Nsight global memory statistics for 2^{27} AES-128 keys

AES-128 Kernel	**Global Memory Load Requests**	**Global Memory Size**	**L1 Cache Hit Rate**	**L2 Cache Hit Rate**
Kernel #1	352,256	14.11 MB	69.42%	99.99%
Kernel #2	327,680	11.11 MB	72.32%	99.97%
Kernel #3	327,680	11.11 MB	72.32%	99.99%
Kernel #4	327,680	11.11 MB	72.32%	99.99%
Kernel #5	327,680	11.11 MB	72.32%	99.99%
Kernel #6	352,256	14.11 MB	69.42%	99.99%

CUDA offers a concept called shared memory which can be considered as the same structure as L1 cache. However, the contents of shared memory are managed explicitly

by the code while L1 cache is automatically managed by the device. This is equivalent to a user managed cache. Shared memory has much smaller space considering global memory, but it is immensely fast since it is much closer to threads in terms of memory hierarchy. After receiving global memory data, each thread stores tables, round constants and ciphertext into the shared memory. Since every table has at most 256 elements, this allocation is done by the first 256 threads inside the block. Each thread in a block that has index value bigger than 256 just waits the first 256 to finish allocation process and each of those first 256 threads performs just one allocation for tables and S-boxes with respect to index number obtained by threadIdx. To illustrate, a thread with index number 5 allocates only 5^{th} positions of the tables and S-boxes. In this way, even allocation is done in a paralleled way.

To achieve high bandwidth, shared memory space is divided into equally sized memory modules called banks. Banks inside shared memory can be accessed simultaneously if every request fall to distinct banks. This creates higher bandwidth than a single module, almost as many times as the concurrent request number. However, if any two of the requests fall into the same bank, shared memory cannot serve that bank simultaneously so the transaction of data must be serialized. This slows down the performance of shared memory, so it is important to understand how memory addresses map to banks in order to schedule memory requests to minimize bank conflicts and get maximum performance. Shared memory has 32 banks and each bank consists of 32 bits. So, if a warp of 32 threads requests something from different banks, all their transactions are done simultaneously without any unnecessary waiting for each thread.

In order to reduce bank conflicts, a special array storing technique is used for shared memory transactions. Each table and S-box array inside the kernels consist of 256 elements. Traditional allocation of such tables which is used in kernel #1 and #2 is illustrated in Figure 4.2. According to it, array elements are stored as a 2D array with exactly 32 columns. So, when an array that consists of 256 elements is stored in shared memory, it exists as a 2D array with dimensions of 8x32. In this representation, each column can be considered as a bank. When two threads access 1^{st} and 2^{nd} elements of the array, there is no bank conflict. In this way, access transactions are broadcasted to these threads simultaneously. However, when two threads access 1^{st} and 33^{rd} elements of the array, these two access requests fall onto the same bank. This creates conflict and access transactions will be serialized.

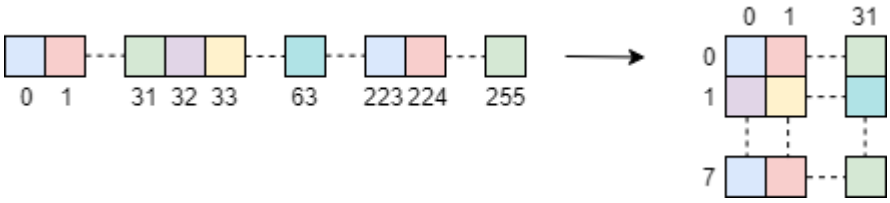


Figure 4.2: Array allocation visualization on shared memory without extension

In order to escape the serialization of banks for shared memory, extension technique is used. This method consists of extending the array to 32 times so that each bank of the shared memory has at least one value of each element of the initial array. The main reason behind this decision lies in the idea of assigning different banks for each thread of a warp. In other words, threads inside a warp are blocked from accessing the banks of other threads. Figure 4.3 show the illustration of this extension method. 1st element of the array is extended 32 times which covers the 1st line of the shared memory. 2nd element covers the 2nd line and so on. As a result, an array with 256 elements are stored in shared memory as a 2D array with dimensions of 256x32.

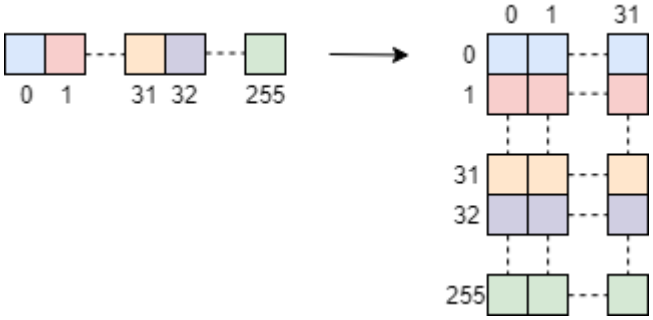


Figure 4.3: Array allocation visualization on shared memory with extension

After extending the table, when a thread requests access to shared memory, it accesses through its thread index inside a warp. Since a warp consists of 32 threads, it makes certain that accesses coming through a warp do not resolve in bank conflicts. To illustrate, T0 thread inside a warp always accesses the shared memory from the 1st column. T1 thread always accesses from 2nd column and so on. For instance, if T0 and T1 threads inside a warp want to access 2nd element of the array, it should access through 1st and 2nd columns of the 2nd row with respect to their thread indexes. The access map difference on shared memory is visualized in Figure 4.4 for every thread accessing 2nd element of the array. In the first illustration, resulting bank conflicts can clearly be seen since every thread in a warp maps to the same element of the shared memory and its bank, which is the case for non-extended tables. However, the second illustration shows that each thread accesses the 2nd element of the array through different banks with the help of extending the array.

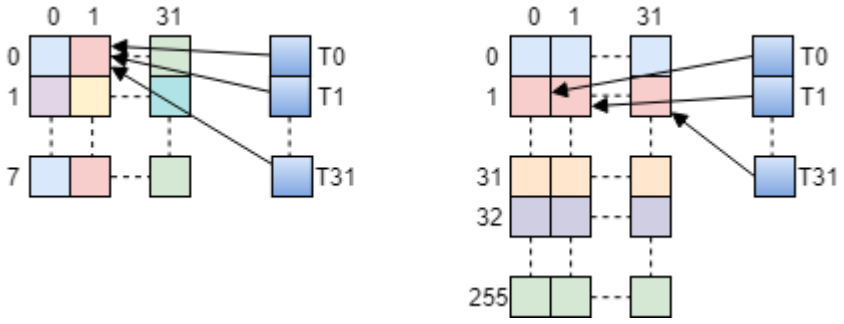


Figure 4.4: Shared memory access map of warp threads for non-extended and extended arrays.

Extending table technique is first introduced in kernel #3 and used in #4, #5 and #6 as well. They all have one table implementation with extension. Extension of one table with 256 elements as integers requires additional 31,774 bytes of allocation (256 * 4 * 31) on shared memory which is the difference between kernels of #2 and #3 (Table 4.1). However, considering kernel #3, S-box is still not expanded so while using it, kernel would still experience bank conflicts. In order to overcome this problem, kernel #5 uses extended S-box. However, this extension is only partly implemented since the shared memory of GTX 970 card does not allow allocating two tables to be extended at the same time. It has capped 49,152 bytes of shared memory per SM (Table 3.2) and kernel #4 uses 33,848 bytes of it (Table 4.1). GTX 970 does not have enough hardware to support a second extension. So, S-box is partly extended in 8 banks instead of 32. By extending S-box 8 times, additional 7,168 (256 * 4 * 7) bytes are allocated for kernel #5 which can be seen in Table 4.1. The visualization of extending S-box 8 times in shared memory is shown in Figure 4.5.

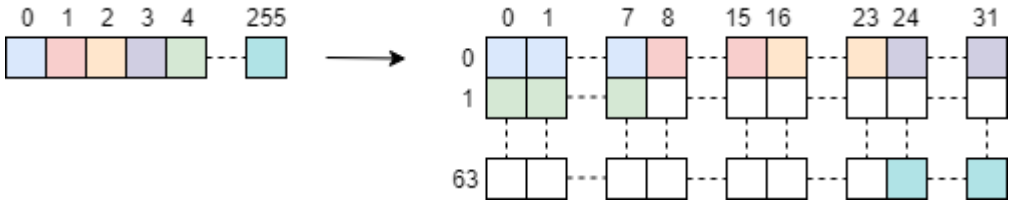


Figure 4.5: Array allocation visualization on shared memory with partly extension

In this partly expanded S-box implementation, just like full table extension of 32 times, threads inside the warps access banks of extended array with respect to their thread index. However, access is done in modularity of 8 since it is extended 8 times. To illustrate, Figure 4.6 shows the access map of threads inside a warp requesting transaction from the 4th element of the array shown in Figure 4.5. The T0 thread accesses through 1st column and T1 thread through 2nd column and so on. However, T8 (thread number 9) accesses 4th element of the array through the 1st bank since the initial array is extended only 8 times and it coincides with the T0 thread. Ergo, in

this approach, each thread shares its banks with 4 other threads. This helps in decreasing bank conflict number but not as many as extending 32 times.

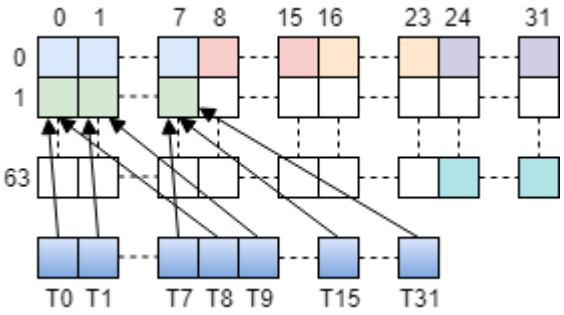


Figure 4.6: Shared memory access map of warp threads for partly extended arrays.

Table 4.9 holds the shared memory statistics for storing operations. The number of store requests equals the amount of shared memory instructions executed. When a warp executes an instruction that accesses shared memory, bank conflicts might block the transaction of that request. In this way, each bank conflict forces a new memory transaction which reduces the instruction throughput accordingly. Moreover, transaction per request ratio shows the average number of shared memory transactions required per executed shared memory instruction. Lower numbers are preferred as each request can be executed in smaller number of transactions. For the case of kernels #1 and #2, transaction per request ratios seem a lot smaller than other kernels. This is the result of extending the table in order to reduce bank conflicts. Extension process greatly increases store transaction values in return transaction per request value also rises. Considering kernel #2, kernel #3 stores almost 115 times more data and produces 216 times faster bandwidth. Ergo, bigger transaction per request values do not always lead to reduced performance. Kernel #5 has partly extended S-box considering #4 which increases store transactions size by 15 MB and store transactions by 6%. However, store transaction per request ratio seems to get lower by 1 point which makes kernel #5 more efficient in storing data to shared memory than #4. It also has the biggest store transaction bandwidth value.

Table 4.9: CUDA Nsight shared memory store statistics for 2^{27} AES-128 keys

AES-128 Kernel	**Store Requests**	**Store Transactions**	**Store Transaction Per Request**	**Store Transactions Size**	**Store Transactions Bandwidth**
Kernel #1	57,344	43,008	0.75	5.25 MB	40.44 MB/s
Kernel #2	32,768	18,432	0.56	2.25 MB	17.13 MB/s
Kernel #3	90,112	2,107,392	23.39	257.25 MB	3.60 GB/s
Kernel #4	90,112	2,107,392	23.39	257.25 MB	3.57 GB/s

Kernel #5	98,304	2,230,272	22.69	272.25 MB	3.82 GB/s
Kernel #6	114,688	2,131,968	18.59	260.25 MB	3.64 GB/s

Nsight also provides load statistics of shared memory for each kernel, which can be found in Table 4.10. Load statistics are far more important than store since the execution depends on the efficiency of load operations while storing data to shared memory only takes place at the beginning of the kernel. Kernel #2 is the one table version of kernel #1 which uses 4 table. This improvement decreases load requests issued from kernel by 1% since there is less data to be looked upon. However, it does not lower load transaction per request ratio since kernel #2 also experiences bank conflicts just like kernel #1. They both fetch almost 260 GB of data during the execution of kernels. Extension mechanism of the look-up table which is introduced in kernel #3 increases load requests by 88% while load transactions value almost stays the same with respect to kernel #2. This also decreases load transaction per request value by 48% which proves that the efficiency of load transactions is increased almost two times while bandwidth stays the same. Figure 4.1 show that kernel #3 is 1.77 times faster than kernel #2, almost proportional to load transaction per request ratio. The same type of improvement can be seen between kernel #4 and #5. Partly extending S-box introduced in kernel #5 increases load request count by 1% while load transaction per request value stays the same. With this improvement, kernel #5 works 7% faster than kernel #4 which can be seen in Figure 4.1. This is a small improvement considering the extension used in kernel #3 because S-box is only partly extended due to shared memory size. However, it still improves the overall efficiency of kernel #5. Load transaction bandwidth of each kernel is closing in to 2 TB data per seconds. This means that almost 2 TB of data stored in shared memory flows to kernels each second, which is a huge number and proves how fast shared memory is. The reason behind every kernel sharing almost identical bandwidth values lies in the speed differences. Kernel #1 and #2 are much slower and each loads total of 263 GBs of data while others are considerably faster (Figure 4.1) but they also have lower load transaction sizes. So, they load smaller data in faster times which does not change the produced load transaction bandwidth.

Table 4.10: CUDA Nsight shared memory load statistics for 2^{27} AES-128 keys

AES-128 Kernel	**Load Requests**	**Load Transactions**	**Load Transaction Per Request**	**Load Transactions Size**	**Load Transactions Bandwidth**
Kernel #1	6,397,109,000	17,034,820,000	2.66	263.63 GB	1.98 TB/s
Kernel #2	6,324,290,000	16,840,910,000	2.66	263.63 GB	1.96 TB/s
Kernel #3	11,894,940,000	16,579,440,000	1.39	137.99 GB	1.93 TB/s

Kernel #4	11,817,770,000	16,471,880,000	1.39	137.99 GB	1.92 TB/s
Kernel #5	11,929,770,000	16,556,780,000	1.39	137.40 GB	1.93 TB/s
Kernel #6	11,902,850,000	16,590,460,000	1.39	137.99 GB	1.93 TB/s

Table 4.11 indicates how many bank conflicts each kernel experienced throughout their execution. Kernel #1 and #2 do not have any extended tables so they have the highest bank conflict values among other kernels, which means they suffer the most from lack of efficiency on shared memory. Kernel #3, which is the first kernel to have extended look-up table, has 76% fewer bank conflicts considering kernel #2. This improvement also decreases bank conflict per request ratio almost 4 times. In other words, kernel #3 is 4 times less likely to hit bank conflicts than kernel #2 which is its equivalent without look-up table extension. This is a huge improvement which increases the overall performance of kernel #3 by 44% (Figure 4.1). Likewise, kernel #5 experiences a 15% drop in bank conflict count by having additional partly extension on S-box considering kernel #4. This also help increasing the efficiency of the performance of kernel #5 by %7 (Figure 4.1). However, bank conflicts per request ratio stays the same since the reduction in bank conflict value is considerably small.

Table 4.11: CUDA Nsight shared memory bank conflicts statistics for 2^{27} AES-128 keys

AES-128 Kernel	**Bank Conflicts**	**Bank Conflicts Per Request**
Kernel #1	1,376,793,000	1.66
Kernel #2	1,376,793,000	1.66
Kernel #3	324,699,300	0.39
Kernel #4	324,699,300	0.39
Kernel #5	319,808,800	0.39
Kernel #6	324,699,300	0.39

4.1.1.8. Overview

According to the results of Nsight experiment, bank conflict problem seems to be the main decisive factor of performance difference between kernels. Extending the look-up table 32 times in shared memory gives 44% more performance which is observed between kernel #2 and #3 in Figure 4.1. Moreover, extending the S-box 8 times, which cannot be done 32 times due to shared memory shortage, advances the performance of kernel #5 by almost 7% considering kernel #4. Other problem-solving techniques contribute very little on the efficiency of the kernels like using byte_perm CUDA function or 4 shifted S-box tables. CUDA byte_perm function decreases

shifting count while increasing a little efficiency. So, using byte_perm function for shifting purposes seems to increase the overall performance of the kernel but not so much to make a big difference. Likewise, using 4 shifted S-box instead of just 1 decreases execution dependency stall reason which increases the overall performance of the kernel #6 considering #4 but not so much to make a real difference in the end. Thus, it is concluded that extended look-up table along with partly extended S-box implementation, which is kernel #5, is the fastest and the most efficient one among others. So, from this part of the research, only kernel #5 is selected and will be used for experiments of AES-128, AES-192 and AES-256.

4.1.2. AES-128 Comparison

For comparing GPU efficiency with CPU, an exhaustive search with 2^{30} keys is started for all kernels and the results are recorded. Each kernel is run for 25 iterations in order to take the average duration. Kernel #5, which is the fastest CUDA kernel, is used for GPU calculations. CPU implementations are designed in a way suitable for working with threads. Figure 4.7 shows the overall duration of exhaustive search iterations. 1T stands for thread numbers, NI stands for new instructions and CPP is for C++ implementation. According to the obtained results, kernel #5 is almost 21 times faster than the best CPU implementation which is 8 threads version of AES-NI and 36 times faster than the same C++ version. This is a huge performance difference which proves how fast GPU is. On the other hand, kernel #5 is almost 73 times faster than AES-NI with 1 thread and 138 times faster than C++ implementation with 1 thread. This demonstrates that using threads for CPU implementations is essential.

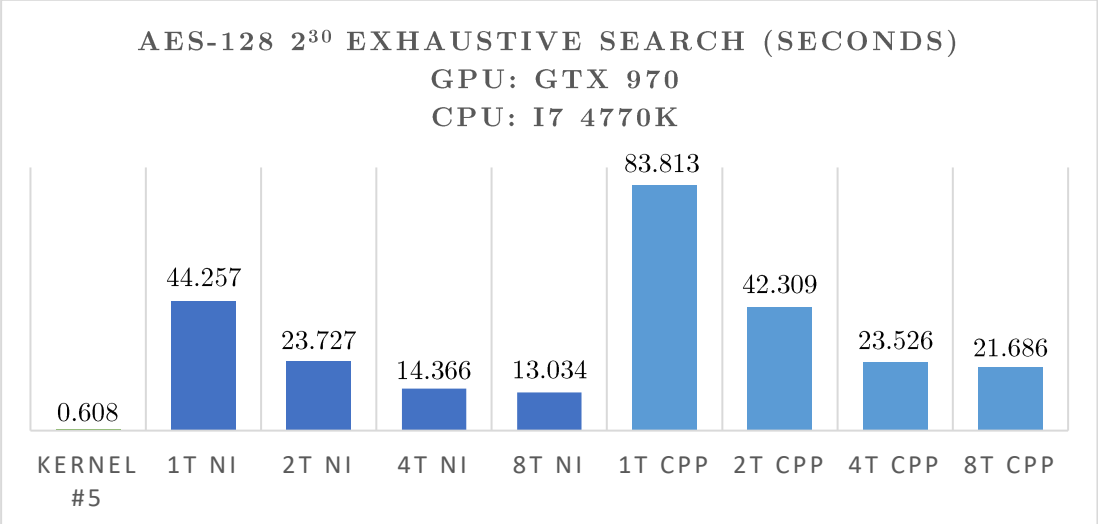


Figure 4.7: CUDA, AES-NI and C++ exhaustive search durations for 2^{30} AES-128 keys

AES-NI efficiency with respect to thread counts considering C++ implementations are as follows: 47%, 44%, 39% and 40%. There is a stable gap between them, and this shows that exhaustive search C++ version does not stay behind for all thread version cases. However, AES-NI version is still far superior than C++ considering it is implemented on hardware level, which indicates the speed difference between hardware implementation and the software. NI version is faster, but it still is not good enough to compete with GPU power even though it is implemented on hardware level.

Performance difference between 1, 2 and 4 threads are almost proportional but 8 threaded versions experience a capped utilization threshold in which the performance is hindered. Since CPU used in this experiment has 4 physical cores, utilization percentage of it goes to %100 during the execution of 8 threaded versions. So, after a certain point, they cannot produce the same performance efficiency and are slowed down. To illustrate this, the efficiency of CPU implementations with each increase in thread counts are: 47%, 39%, 9% for AES-NI and 50%, 44% and 8% for C++. 8 threads versions can only improve the overall efficiency by 9% and 8% in terms of duration. This proves that 8 thread implementations are severely affected by CPU utilization cap.

4.1.3. AES-192 Comparison

Just like AES-128, versions of AES-192 exhaustive search for 2^{30} keys are implemented. The results are recorded and can be seen in Figure 4.8. Kernel #5 version of AES-192 uses the same shared memory but it has 40 registers. So, occupancy value is not the same as AES-128 version which can be seen in Figure 3.8 that 40 registers per thread values produces reduced occupancy. However, CUDA version is still the fastest among other implementations as can be seen in Figure 4.8. CUDA is almost 19 times faster than AES-NI 8 threads implementation, which is the fastest AES-192 CPU implementation and 41 times faster than C++. C++ implementation is still behind AES-NI implementation on the case of AES-192 as NI is more than 2 times faster than C++.

NI implementations show efficiency percentages of 61%, 58%, 55% and 54% considering C++ with respect to thread counts. Like AES-128 case, one efficiency ratio is not peaked for a particular thread count. This is an indication that, C++ versions are able to keep up with AES-NI ones for all cases. However, C++ versions are a little bit behind of AES-NI for AES-192 performance considering the ratio difference of AES-128. Efficiency ratios between thread counts are 46%, 39% and 9% for AES-NI and 50%, 43% and 10% for C++ respectively to increased thread iteration.

According to them, both CPU implementations are affected from CPU utilization cap as 8 threads version only improved their duration by 9% and 10% respectively.

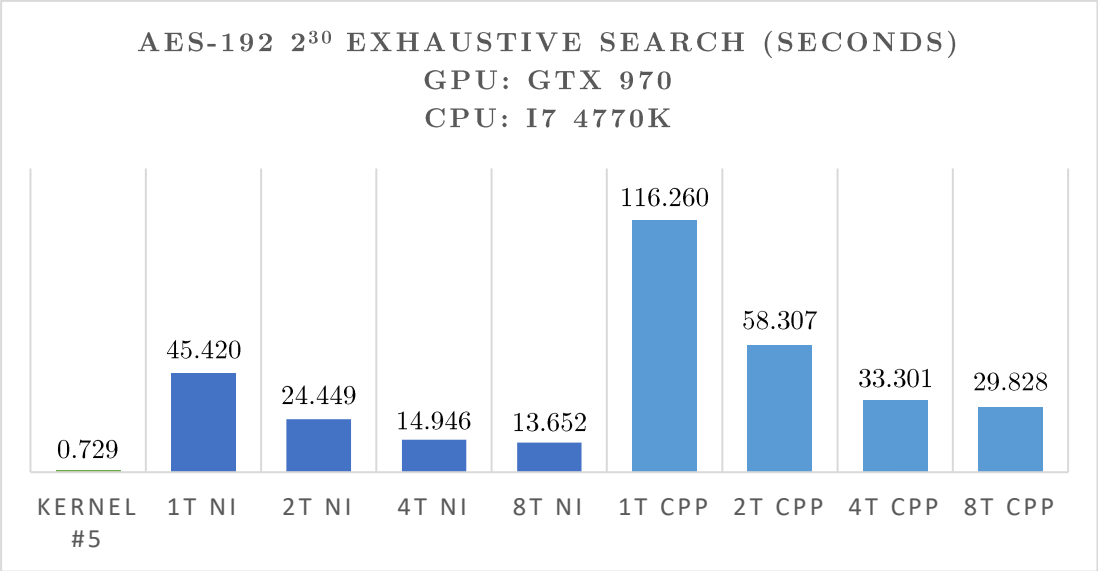


Figure 4.8: CUDA, AES-NI and C++ exhaustive search durations for 2^{30} AES-192 keys

4.1.4. AES-256 Comparison

AES-256 version of kernel #5 uses 40 registers just like the AES-192 version. It also uses the same shared memory configuration and size. Exhaustive search implementations for 2^{30} keys are run for 30 times and the results are averaged. According to them, shown in Figure 4.9, GPU is the fastest execution among others. Kernel #5 is almost 18 times faster than 8 threads AES-NI implementation which is the fastest one implemented on CPU. Moreover, it is also 38 times faster than the best C++ implementation. AES-NI is still the superior case among CPU implementations as it is more than 2 times faster than C++ implementation.

AES-NI shows efficiency values of 58%, 56%, 55% and 53% to the same threaded versions of C++. Just like other exhaustive search cases, 8 threaded versions are affected from CPU cap as efficiency ratio of increased threads are 47%, 42% and 7% for AES-NI and 49%, 43% and 12% for C++.

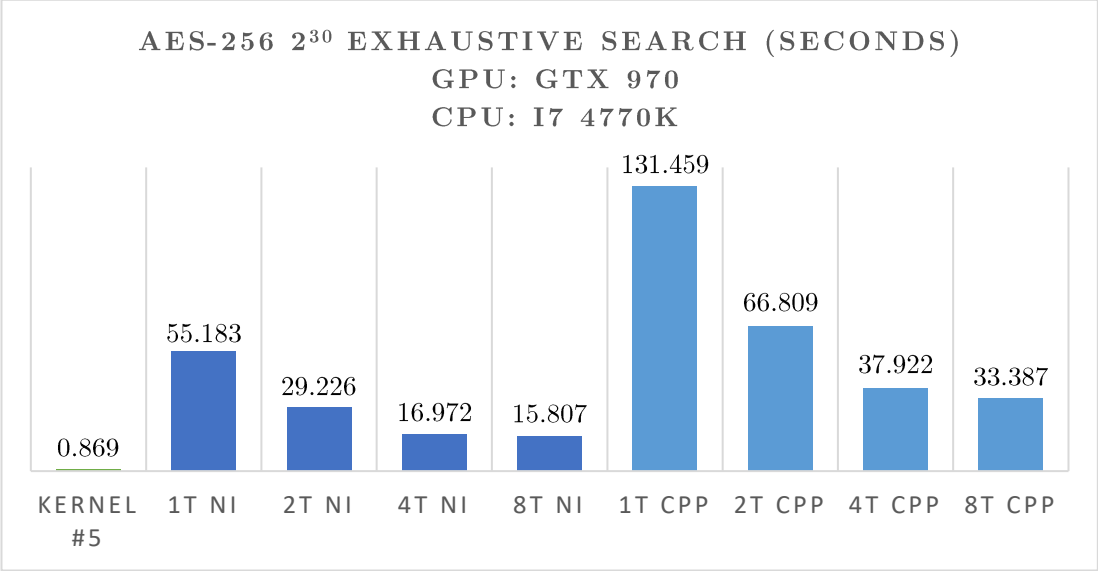


Figure 4.9: CUDA, AES-NI and C++ exhaustive search durations for 2^{30} AES-256 keys

4.1.5. Overview

In this research, 6 different AES-128 kernels are implemented for CUDA in order to compare speed and efficiency differences between them. According to Nsight results, kernel #5 was selected to be the most efficient kernel in terms of performance. After that, it is compared with AES-NI and C++ implementations in order to decide which one is better for exhaustive search. According to results which are shown in Figure 4.7, CUDA implementation is by far superior among others as it is 21 times faster. Thus, if any need arises for problems similar to exhaustive search, CUDA is the best possible conclusion. The same outcome can be observed for AES-192 and AES-256 cases in which CUDA also outperforms every CPU implementation by 19 and 18 times respectively.

CPU implementations suffer from utilization cap such that after 4 threads, increasing thread number does not result in the same efficiency increase as versions before it. Average numbers for efficiency between implementations of increased threads are 46%, 40% and 8% for AES-NI and 50%, 44% and 10%. 8 threads versions indicate that they are slowed down by CPU utilization cap as CPU is always at 100% utilization and working with its full resources while executing 8 threads versions.

Figure 4.10 shows the efficiency difference between each implementation in how many millions of keys can be tried per second. Values are calculated from 2^{30} exhaustive search implementations shown in Figure 4.7, Figure 4.8 and Figure 4.9 respectively. CPU implementation numbers are taken from the most efficient ones which are the 8

threads versions while CUDA metrics are taken from kernel #5. According to Figure 4.10, CUDA can try 1.7 billion of AES-128 keys per second while AES-NI can only process 82 million. In this regard, only by having 22 concurrently workings CPUs can CUDA efficiency be outperformed. The same type of efficiency difference can be seen for AES-192 and AES-256 cases as well.

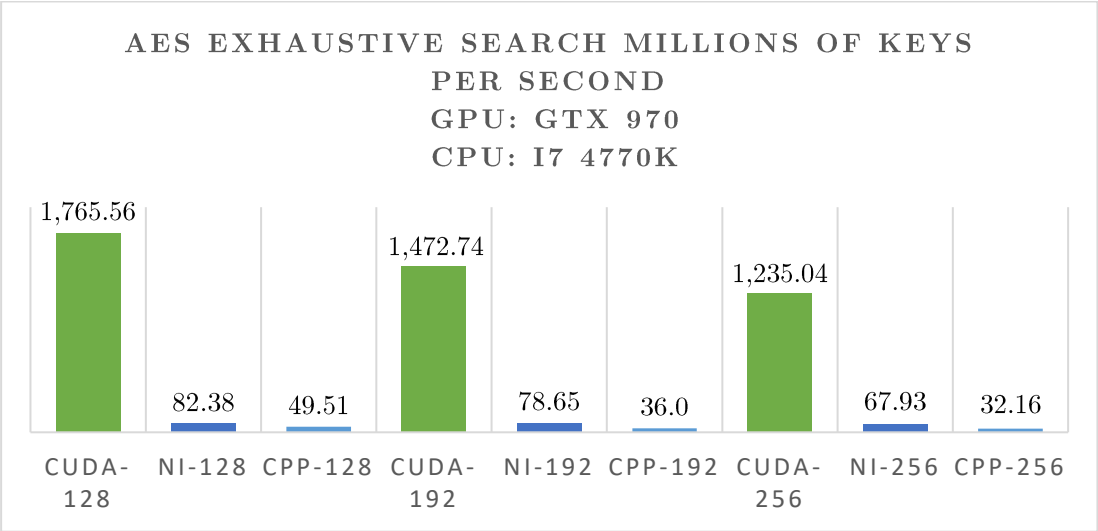


Figure 4.10: CUDA, AES-NI and C++ exhaustive search performance difference

Table 4.12 shows additional information regarding cycles per byte and throughput values. Throughput is shared in terms of GBs per second. Since 8 threads versions are the fastest among CPU implementations, their values are shown along with 1 thread.

Table 4.12: CUDA, AES-NI and C++ exhaustive search results

Architecture	AES		Cycles/byte	GB/sec
NVIDIA GTX 970, 1.4 GHz	128		0.05	26.31
	192		0.06	21.95
	256		0.07	18.40
Intel i7 4770K, 3.9 GHz AES-NI	128	8T	2.96	1.23
		1T	10.05	0.36
	192	8T	3.10	1.17
		1T	10.31	0.35
	256	8T	3.59	1.01
		1T	12.53	0.29
Intel i7 4770K, 3.9 GHz CPP	128	8T	4.92	0.74
		1T	19.03	0.19
	192	8T	6.77	0.54

		1T	26.39	0.14
	256	8T	7.58	0.48
		1T	29.84	0.12

Another point on CPU implementations is that while CPU is directly connected to operating system, GPU is free to execute its kernels. This brings about some performance issues such that when CPU is overloaded with instructions, the performance of exhaustive search is affected. In this research, CPU durations are recorded in idle state where only the exhaustive search command line application is running. However, when the computer is not idle, operating system will allocate some CPU resources to the tasks that are issued by the user. In this way, CPU implementations are slowed down and will not give the same output performance. However, this is not the case for GPU as it can exist inside the system as a coprocessor. So, CPU and operating system are not affected by the highly utilized exhaustive search program. This is mostly the case for 8 threads version since utilization of CPU is so high on these implementations, operating system can be stalled while they are working. This is another advantage of CUDA as the user experience on operating system will not be affected while an exhaustive search is being executed on GPU. As a result, users can continue to use the computer while they must wait for exhaustive search to be finished on CPU.

4.2. CTR Implementation

CTR is a mode of operation which allows AES to be implemented in multithreading way. It Is important since file encryption section uses CTR mode of operation. In this research, it is used for this section and file encryption. This section investigates on the fly CTR encryption which is basically exhaustive search without key scheduling part. Before the beginning of execution, a key is selected, and each round of the key is calculated. Then, they are used as precalculated keys for AES rounds while execution is ongoing. In other words, in CTR implementation, each thread increments plaintext instead of key, which eliminates the need of key scheduling each iteration. So, encryption takes less time than exhaustive search. Each CTR implementation is run for 25 times with the aim of doing 2^{30} encryptions and their results are averaged for AES-128, AES-192 and AES-256 which can be found in Figure 4.11, Figure 4.12 and Figure 4.13. respectively. After each iteration, 10 seconds of intervals are put between executions just like exhaustive search case.

CUDA version of CTR is derived from kernel #5, fastest kernel of AES-128 exhaustive search experiment. In this version, keys are sent to kernel via global memory with cudaMallocManaged function and fetched into shared memory at the beginning of

execution just like exhaustive search implementations. However, instead of just 128 bits, total of 11 128 bits of keys are stored for AES-128, 13 128 bits for AES-192 and 15 128 bits for AES-256. Other than using these keys for each round, there is not any difference between CTR and exhaustive search versions of kernel #5 in terms of design. Each CUDA kernel has 32 registers so occupancy ratio will be about maximum efficiency according to Figure 3.8.

4.2.1. AES-128 Comparison

Experiment results of AES-128 for all implementations can be found in Figure 4.11. Kernel #5 used in exhaustive search is modified for using precalculated keys instead of calculating different keys for each iteration. AES-NI and C++ implementations are also altered in the same manner. Results are calculated and according to them, kernel #5 is more than 4 times faster than the fastest CPU implementation which is 8 threaded version of AES-NI. It is also faster than 8 threads C++ implementation by 39 times. 4 and 8 threads versions of AES-NI are almost identical in terms of performance.

AES-NI version seems to be the efficient one of CPU implementations just like it is in exhaustive search since it is 9 times faster than C++ implementation. AES-NI implementations increase efficiency by 93%, 92%, 90% and 89% with respect to thread numbers considering C++ implementations. The efficiency differences between threads for AES-NI are 39%, 30% and 0% while C++ have 49%, 43% and 12%. 8 threads version of AES-NI takes almost the same time as 4 threads version so increasing thread number after 4 does not make any difference for the case of AES-NI. However, 8 threads C++ implementation improves its duration by 12%. This shows that, increasing thread count to 8 from 4 can further improve the efficiency of C++ implementation while AES-NI is not affected.

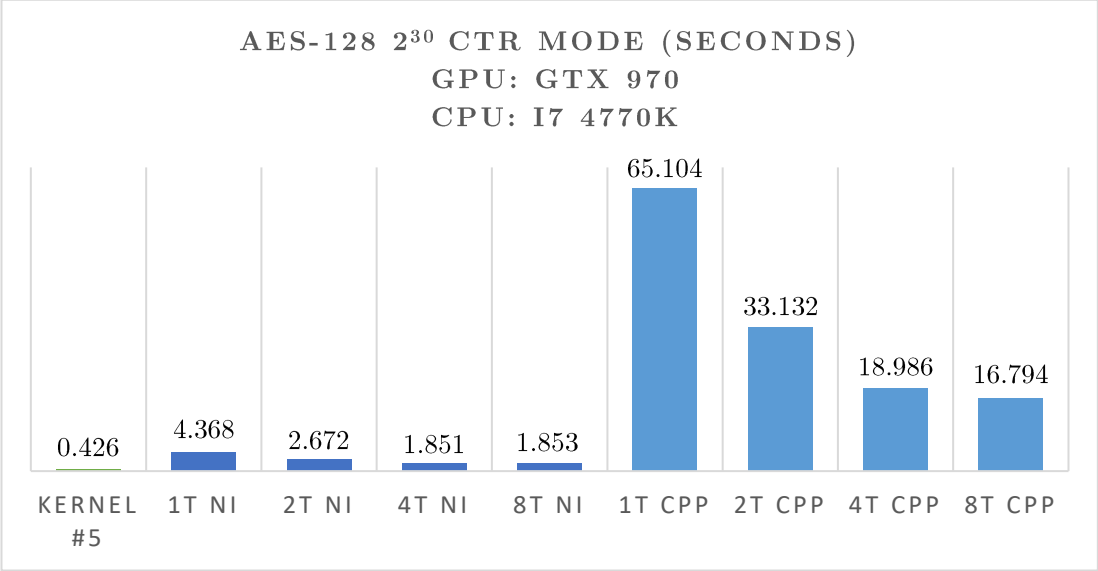


Figure 4.11: CUDA, AES-NI and C++ AES-128 CTR durations for 2^{30} encryptions

4.2.2. AES-192 Comparison

Just like AES-128 CTR implementation, AES-192 versions are also implemented, and the results are recorded in Figure 4.12. According to them, CUDA is the superior implementation by far with it being 4 times faster than the most efficient CPU implementation, 8 threaded AES-NI. It is also 39 times faster than C++ implementation. This shows that CUDA is working 75% more efficiently than the best AES-NI implementation and 97% than C++ one. 4 and 8 threads versions of AES-NI produce identical performance in terms of duration.

The efficiency differences between AES-NI and C++ with respect to thread count are: 93%, 92%, 90% and 90%. 8 threads version of AES-192 NI even performs more efficiently than AES-128 version considering C++ implementation. Performance improvements between threads for AES-NI are 41%, 33%, 0% while they are %50, %45 and %7 for C++. Like AES-128 case, increasing thread count of AES-NI implementation for CTR mode does not affect the performance output after 4 threads. While C++ implementation shows the potential of 7% improvement in terms of duration, AES-NI is not able to increase its efficiency after 4 threads.

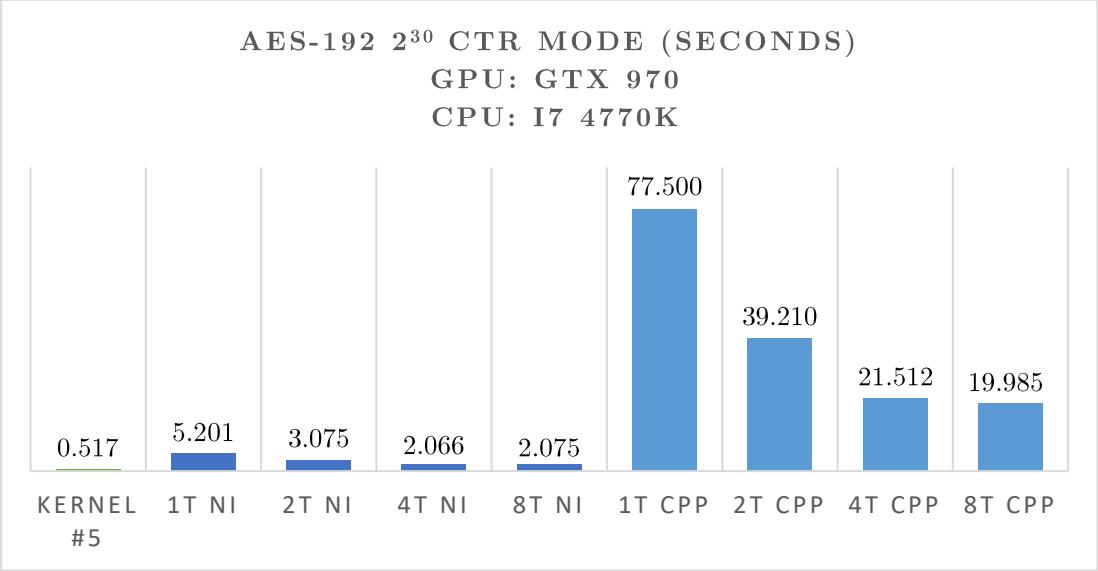


Figure 4.12: CUDA, AES-NI and C++ AES-192 CTR durations for 2^{30} encryptions

4.2.3. AES-256 Comparison

CTR versions of AES-256 are also implemented and they are run for 25 times just like every other implementation. The results are averaged and recorded in Figure 4.13. CUDA implementation is still the fastest among other ones with it being almost 4 times faster than the fastest CPU implementation. Moreover, CUDA is faster than the best C++ implementation by 38 times. This shows that CUDA is still the most efficient implementation among others. 4 and 8 threads versions of AES-NI are almost identical in terms of performance just like every other CTR implementation.

AES-NI is still the better one considering C++ version of AES-256. Its efficiency ratios against C++ versions are: 93%, 92%, 91% and 90% with respect to thread counts. AES-NI shows efficiency ratios between different thread counts like 42%, 34% and 0% while for C++ these values are 49%, 45% and 7%. Like other CTR implementations, 8 threads AES-NI could not improve its performance in terms of duration while C++ boosts its efficiency by 7%. This shows that C++ implementation has the potential for self-improvement for 8 threads while AES-NI produces the same result as 4 threads version.

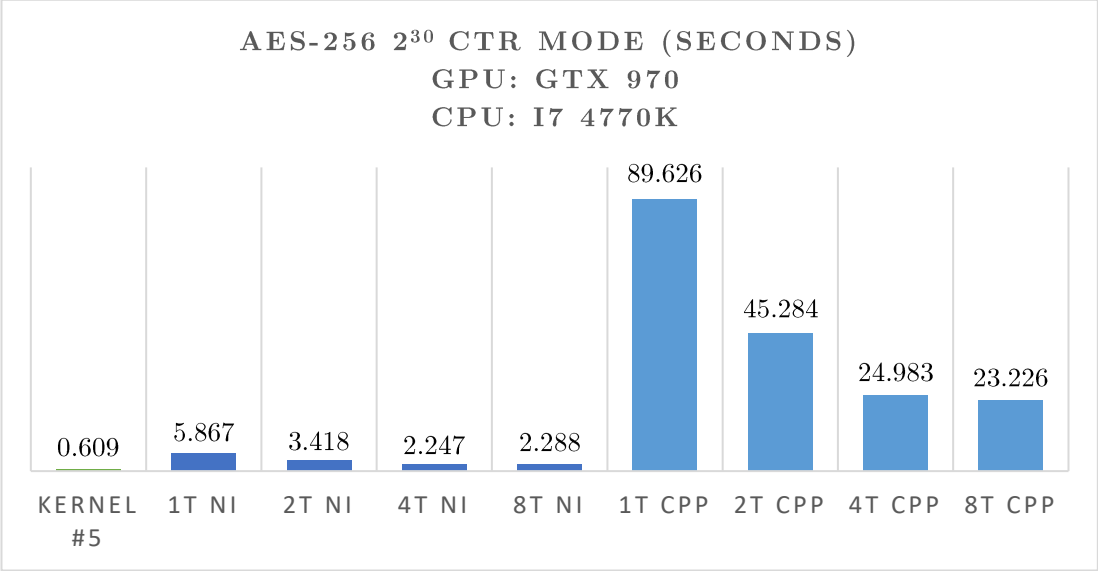


Figure 4.13: CUDA, AES-NI and C++ AES-256 CTR durations for 2^{30} encryptions

4.2.4. Overview

On the fly CTR implementations proved that they are faster than exhaustive search versions by 7 to 10 times. This was the expected case since CTR versions do not execute key scheduling for different keys like exhaustive search. Round keys are precalculated and sent to execution beforehand so there is no need to calculate them in each iteration. This is the basic speed difference between exhaustive search and CTR implementations.

CUDA is by far the best choice for on the fly CTR implementation as well. It produces 4 times faster performance than CPU implementations. This performance difference was 21, 19 and 18 times for exhaustive search cases. So, in this regard, it can be said that CPU implementations are faster for the case of CTR than exhaustive search. In other words, exhaustive search is where CUDA extracts the most power from GPU and makes a difference. CUDA is also advantageous in CTR implementation as well since while execution is ongoing, CPU should not be used in order to extract the full power from CPU, especially for 8 threads versions in which CPU utilization is capped on 100% and no resource is allocated to other CPU tasks. This is not the case for CUDA since GPU can coexist with CPU without affecting its utilization and in return; using CUDA version does not affect user experience while computer is being used.

Just like the exhaustive search case, CPU implementations of 8 threads experience a little hindrance on performance due to utilization cap. This cap is very visible in AES-NI implementations because NI versions are heavily optimized in terms of hardware and use all CPU resources while working. Average performance difference between

AES-NI implementations are 40%, 33% and 0% with respect to thread numbers. These values are 49%, 44% and 9% for C++ implementations. The hindrance can be seen clearly for 8 threads version of AES-NI and C++. AES-NI version also experiences another reduction of performance in which the performance does not increase after 4 threads. This was not the case for exhaustive search in which 8 threads AES-NI implementation performs 8% better than 4 threads while this ratio is 0% for CTR. This indicates that 8 threads CTR AES-NI could not produce more performance output than 4 threads version while exhaustive search one does.

Figure 4.14 shows the performance difference of implementations with different key lengths in terms of how many GBs of data each implementation can encrypt per seconds. CUDA outperforms other implementations by a big margin. If 128 bits of key is used, kernel #5 can encrypt 37.52 GBs of data per second without considering I/O operations using CTR mode of operation. This rate falls to 8.64 GBs per second for AES-NI case. On the other hand, C++ implementation can encrypt 1 GB of data per second. Results of other key lengths show similar outcomes in favour of CUDA. So, using CUDA for CTR encryptions is vastly advantageous considering CPU implementations.

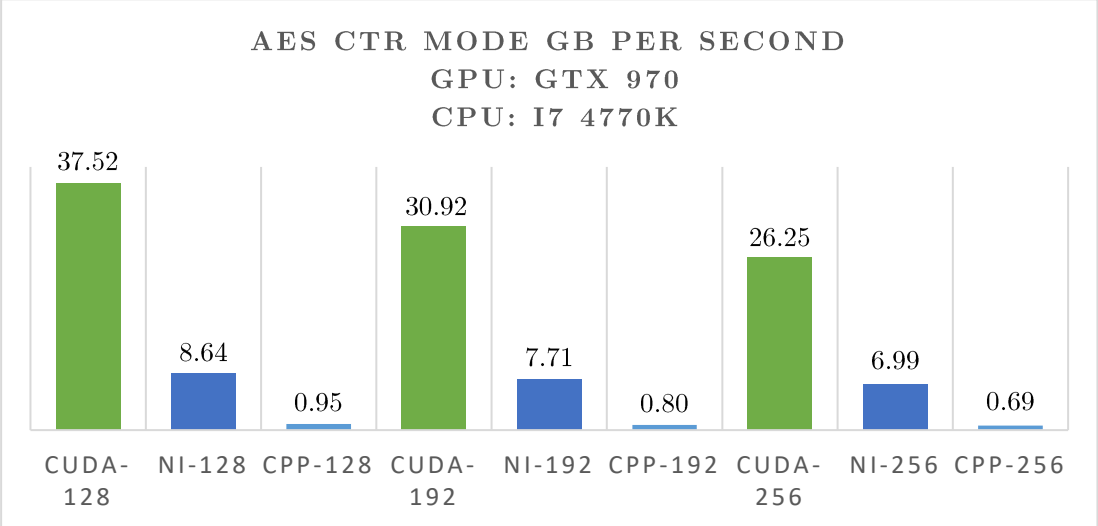


Figure 4.14: CUDA, AES-NI and C++ CTR performance difference

There is a Windows program that benchmarks various components of machines while they are running. It is called WinSAT which is short for Windows System Assessment Tool. The executable file can be found in the system32 folder of the Windows installation. This program can assess AES-256 performance using CPU. According to the tests done on the same machine with i7 4770K CPU, AES-256 encryption speed is found as 4,112 MB/s which is around 4 GBs per second. There is no detailed documentation on this tool, so it is unknown what mode of operation is used. However, while CPU assessment is ongoing, WinSAT uses all available CPU cores which can be

seen in task manager of Windows. According to this finding, it can be assumed that WinSAT uses 8 threads while calculating this AES-256 speed. Our findings show that our 8 threads AES-NI implementation can encrypt 6.99 GBs per second using CTR mode while the implementation of Microsoft can only encrypt 4 GBs per second. This indicates that our AES-NI implementation is faster than the CPU assessment tool of Microsoft.

Table 4.13 shows the overall results for CTR experiments. Cycles per byte and throughput as GB/sec values are shared for every CTR implementation. The table has both 1 and 8 threads versions of CPU implementations indicated as 1T and 8T.

Table 4.13: CUDA, AES-NI and C++ CTR results

Architecture	**AES**		**Cycles/byte**	**GB/sec**
NVIDIA GTX 970, 1.4 GHz	128		0.03	37.52
	192		0.04	30.92
	256		0.05	26.25
Intel i7 4770K, 3.9 GHz AES-NI	128	8T	0.42	8.64
		1T	0.99	3.66
	192	8T	0.47	7.71
		1T	1.18	3.08
	256	8T	0.52	6.99
		1T	1.33	2.73
Intel i7 4770K, 3.9 GHz CPP	128	8T	3.81	0.95
		1T	14.78	0.25
	192	8T	4.54	0.80
		1T	17.59	0.21
	256	8T	5.27	0.69
		1T	20.35	0.18

4.3. File Encryption

File encryption is important since it is one of the most basic usages of AES. In this research, CTR mode of operation is used for encrypting files. It is one of the officially published mode of operations by NIST. Its main advantage is that it is parallelizable which makes it perfect for multithreaded situations. CTR section of this research forms the basis for file encryption implementations. The difference between them is that in CTR implementations; resulting ciphertext is not stored anywhere while for

file encryption implementations; ciphertexts must be stored in RAM in order to be written as files to disks. For the case of GPU, data is stored in VRAM. This situation favours the CPU implementations since in order to write some data as a file, it must exist in RAM. However, since GPU uses VRAM, which is different than RAM, encrypted data is stored there and must be moved to RAM. This causes a lot of time considering how fast CUDA encrypts, which kind of blocks the overall performance output of GPU. Duration of this movement is totally bound by I/O operations and affected by motherboard, CPU and even the speed of GPU socket connected to motherboard. The data writing speed, which is the speed for fetching data from RAM and sending it to disk system, is also bound by I/O speeds of the system and availability to write. Ergo, these duration values do not specify whether CUDA or CPU is faster.

The basic difference between CTR and file encryption implementations is how each threads process data. In CTR implementations, each thread knows which interval of total data will be processed. In other words, total encrypted data is equally separated to each thread. This is also for the case of exhaustive search. Since their data is artificially created, it is done in such a way that every thread gets equal shared. For instance, if there are 2048 keys to be tried for exhaustive search and 1024 threads, each thread process 2 different keys. This is illustrated in Figure 4.15 on the left. On the other hand, for file encryption; a different data sharing mechanism is implemented, which is shown on the right. The fundamental idea behind this design lies in sharing the data equally between the threads in the case of whole data cannot be divided in equal portions. Considering there are n threads, each thread gets its next task by adding n to its data process index. If there is no data after this adding process, this means that thread is reached its end of life cycle and stopped. After every thread is stopped, overall process is finished. In this way, every thread equally takes a task from whole data. This approach is so much easier to be implemented than sharing irregularly varied data to equal chunks according to dynamic thread count.

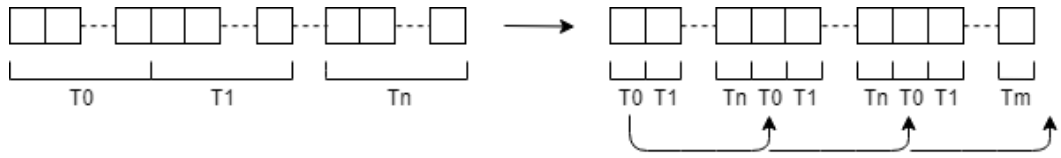


Figure 4.15: Thread allocation difference between exhaustive search, CTR and file encryption

For this part of the research, a file is selected with the size of 2.27 GB (2,447,461,582 bytes) and encryption iterations are carried out by CUDA, AES-NI and C++ versions. Each encryption is run for 25 times with 10 seconds of intervals, which is the same case as exhaustive search and CTR implementations. Obtained results are recorded.

In each implementation, only counter values are sent to execution according to file size. So, when the file size is read as 2.27 GB, counter values with respect to that size will be encrypted. After encryption, resulting ciphertexts are XORed with the data stream of file. For CUDA case, counter values are encrypted and stored in global memory. After that, data is fetched to RAM. For CPU case, since data is already in RAM, there is no need for additional operations like CUDA.

GTX 970 is connected to Z87 motherboard via PCIe v3.0 x16 slot. It has 4 GB GDDR5 VRAM. RAM used in the system is 8x2 DDR3 SDRAM with 1600 MHz bus speed, Kingston ASU16D3LU1KBG/8G. For read/write operations, RBU-SC400S371128GB SSD of Kingston is used. This SSD is fairly an average one considering the performance and according to userbenchmark.com; it has 427 MB/s reading and 148 MB/s writing sequential speeds for operating on big files [44]. These measurements are taken from 579 benchmark results done by real people across the world. So, system is bound by these values, meaning that no matter which speed the encryption can be done, it cannot cross the 150MB/s threshold while writing data to a file. When data is in RAM, file is read with 1024 bytes of chunks and each chunk is XORed with the relevant encrypted counter section. The result is written to disk with 1024 bytes of chunks as well. For each implementation, code used for file read and write sections are the same in order not to create any inequity between them.

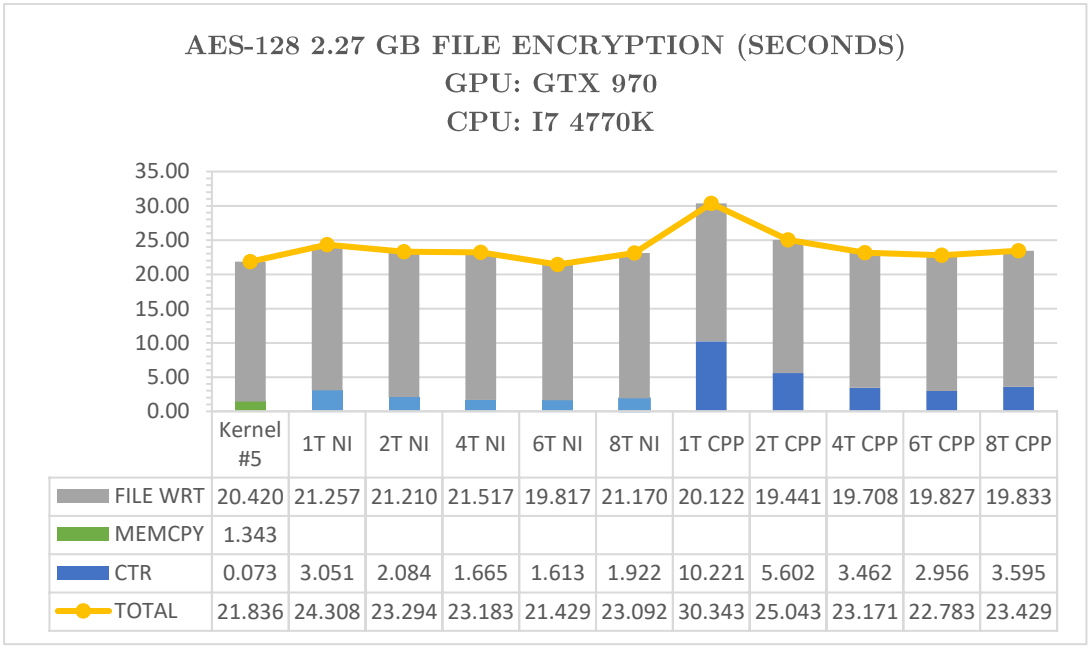
4.3.1. AES-128 Comparison

According to the results of file encryption experiment for AES-128, which can be found in Figure 4.16, CUDA is still the fastest option. It is 22 times faster than 6 threads AES-NI implementation and 40 times than 6 threads C++ without considering file write operation. In other words, using CUDA for file encryption gives 95% more efficiency in terms of duration. After the data is fetched to RAM, file is written to disk around 20 seconds. This duration is not relevant for the scope of this research as only file encryption speeds are important. Nevertheless, they are shown in Figure 4.16 for each implementation. It must also be noted that file writing speeds are erratic and subjected to the state of the system in which the writing operation is being carried out.

CUDA version of AES-128 uses 39 registers, which slows down the kernels according to Figure 3.8, and 41,136 bytes of shared memory. It is basically the same implementation of kernel #5 but only data sharing mechanism is changed according to Figure 4.15. Before the kernel is started, void data is allocated in global memory with respect to file size for encrypting counter values. This allocation is done by cudaMalloc command. So, unified memory is not used for encrypted file data since it is slower. Each thread populates its own encrypted values in global memory and when the overall data population is finished, data is fetched to RAM by cudaMemcpy

function which transfers block of data from global memory to RAM. After that, only writing the encrypted data as a file operation remains. So, actual file encryption process is stopped when the full encrypted data is reached to RAM. According to this approach, CUDA shows the most promising speed in terms of CTR encryption as it can produce 22 times more encrypted data as throughput. However, this produced data sits on global memory and needs to be transferred to RAM in order to be written as a file. Traversing of this data greatly hinders the performance of CUDA as 95% of the file encryption process is spent on moving the data. This is what bottlenecking CUDA performance for file encryption considering CPU implementations.

AES-NI outperforms C++ just like exhaustive search and CTR implementations. It shows 70%, 63%, 52%, 45% and 47% more efficiency with respect to thread counts. AES-128 file encryption also has 6 threaded versions of CPU implementations. The reason behind this is that 8 threads C++ implementation performs worse than 4 threads. This means that increasing threads no longer results in more performance as more threads stall the overall performance while storing data to RAM. So, the performance of 6 threads is also included for AES-128 in order to check whether there is a point between 4 and 8 threads that increases performance. According to Figure 4.16, 6 threads implementation performs better than both 4 threads and 8 threads by 3% and 16% respectively for AES-NI. For C++ implementation, performance difference of 6 threads with respect to 4 and 8 threads are 14% and 18%. 6 threads.

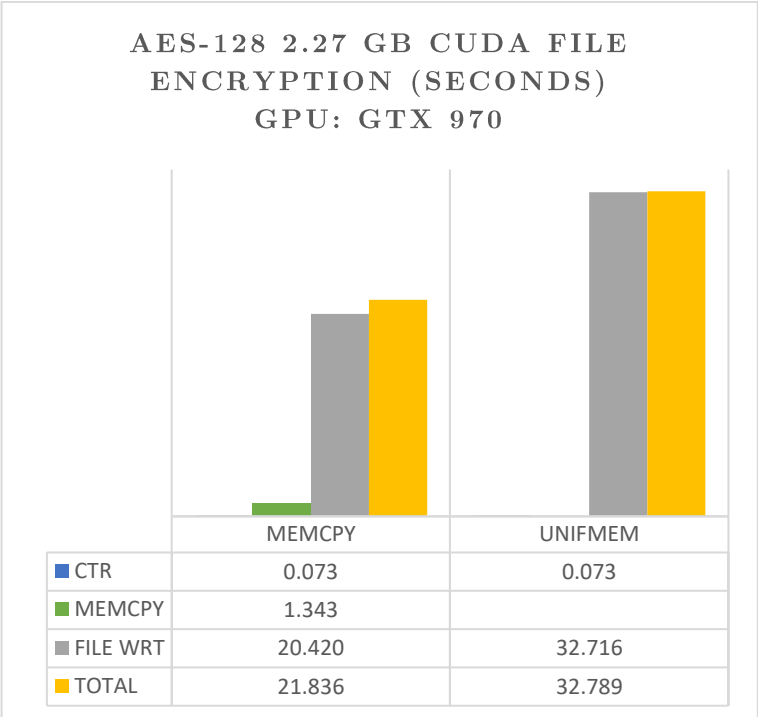


	Kernel #5	1T NI	2T NI	4T NI	6T NI	8T NI	1T CPP	2T CPP	4T CPP	6T CPP	8T CPP
FILE WRT	20.420	21.257	21.210	21.517	19.817	21.170	20.122	19.441	19.708	19.827	19.833
MEMCPY	1.343										
CTR	0.073	3.051	2.084	1.665	1.613	1.922	10.221	5.602	3.462	2.956	3.595
TOTAL	21.836	24.308	23.294	23.183	21.429	23.092	30.343	25.043	23.171	22.783	23.429

Figure 4.16: CUDA, AES-NI and C++ AES-128 file encryption durations

CUDA version of AES-128 is experimented with two different data copying techniques which are called unified memory and memory copy. The obtained results are shared

in Figure 4.17. According to it, cudaMemcpy approach is by far the superior among two. In both attempts, encrypting data in CTR mode of operation takes the same time but the real difference is shown in file write operation. Unified memory example, in which CUDA handles the transfer of data from global memory to RAM, spends almost 60% more time to write the file to disk. However, memory copy example, which uses cudaMemcpy CUDA function to transfer data, takes so much less time than unified memory case. It can be said that memory copy case is 33% efficient than unified memory approach in terms of overall duration. Ergo, if performance is important, data must be carried with memory copy approach, not with unified memory. This is the reason that kernel #5 of this experiment uses cudaMemcpy approach for every file encryption implementation.



	MEMCPY	UNIFMEM
CTR	0.073	0.073
MEMCPY	1.343	
FILE WRT	20.420	32.716
TOTAL	21.836	32.789

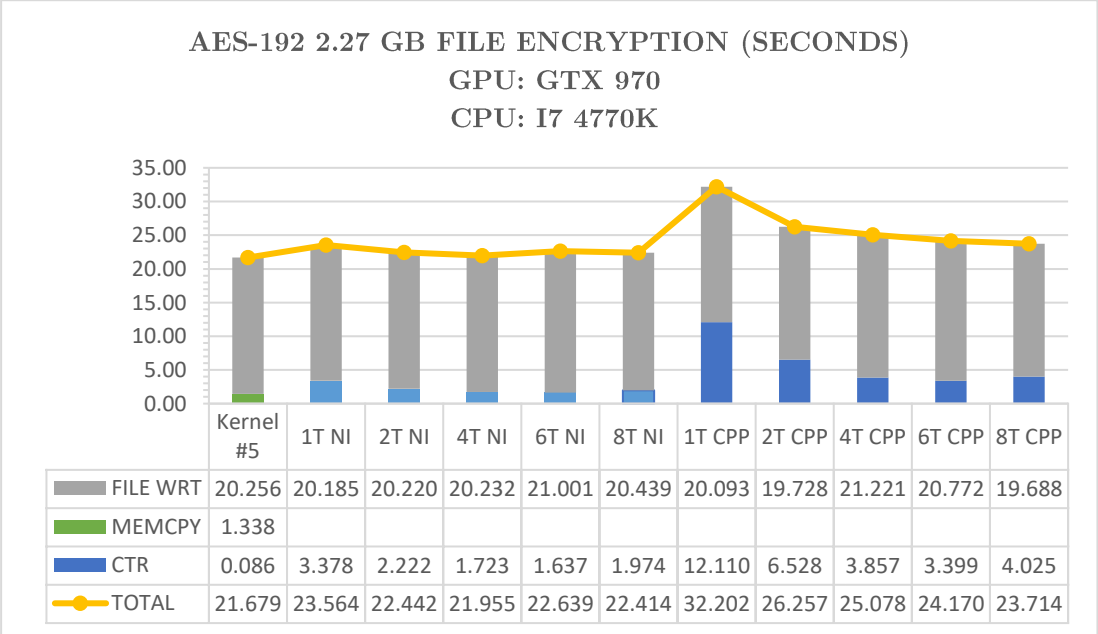
Figure 4.17: CUDA memcpy and unified memory difference on AES-128 file encryption

4.3.2. AES-192 Comparison

CUDA version of AES-192 file encryption uses 41,168 bytes of shared memory and 39 registers like AES-128. This is not the ideal case for CUDA since any value more than 32 registers results in badly affected occupancy ratios, as can be seen in Figure 3.8. However, CUDA still offers the best possible solution when it comes to file encryption with AES-192. GPU is 19 times faster than the fastest AES-NI implementation and 40 times faster than C++. CUDA memory copy duration matches the one for AES-

128 since the data transferred is of the same size. Only the complexity of CTR operation is changed.

AES-NI is still better than C++ implementation with efficiency ratios of 72%, 66%, 55%, 52% and 51% with respect to increasing thread counts. Just like AES-128 case, 6 threads versions of AES-192 are calculated and the results show that they produce the best performance. 6 threads version of AES performs 5% and 17% better than 4 and 8 threads versions while for C++, these ratios are 12% and 16%. Efficiency ratios of increasing threads to 6 are 34%, 22% and 5% for AES-NI and 46%, 41% and 12% for C++. This shows that while AES-NI is faster, C++ implementations gain more efficiency from increased thread counts considering AES-NI.



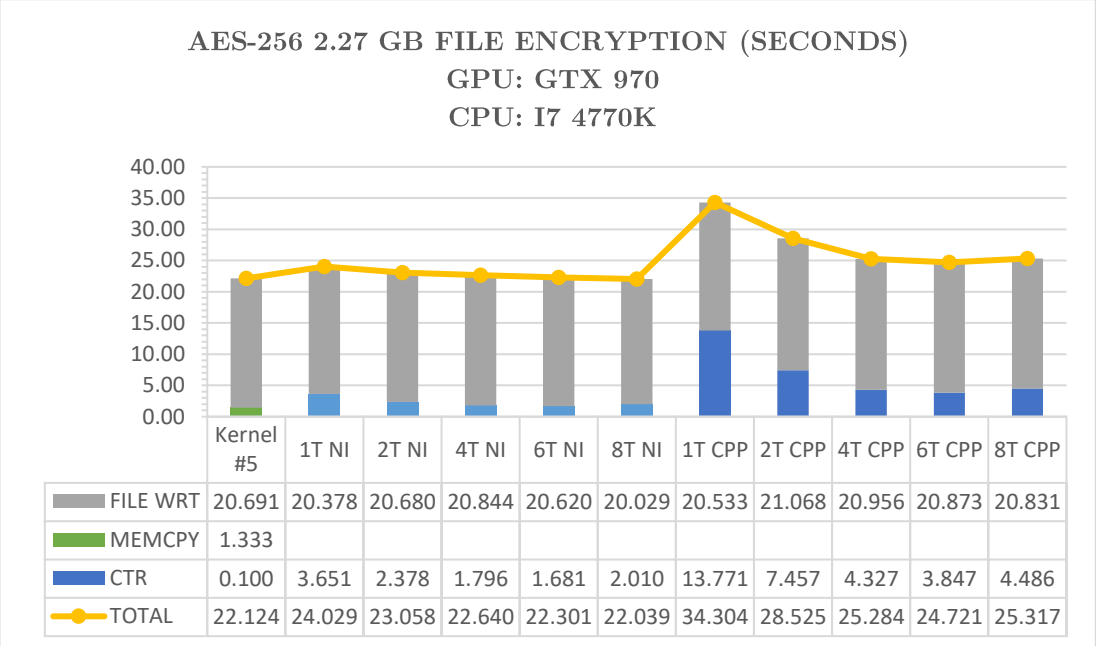
	Kernel #5	1T NI	2T NI	4T NI	6T NI	8T NI	1T CPP	2T CPP	4T CPP	6T CPP	8T CPP
FILE WRT	20.256	20.185	20.220	20.232	21.001	20.439	20.093	19.728	21.221	20.772	19.688
MEMCPY	1.338										
CTR	0.086	3.378	2.222	1.723	1.637	1.974	12.110	6.528	3.857	3.399	4.025
TOTAL	21.679	23.564	22.442	21.955	22.639	22.414	32.202	26.257	25.078	24.170	23.714

Figure 4.18: CUDA, AES-NI and C++ AES-192 file encryption durations

4.3.3. AES-256 Comparison

File encryption version of kernel #5 uses 39 registers per thread as well. It also uses 41,200 bytes of shared memory. Each implementation is run for 25 times with 10 seconds of intervals and results are recorded to Figure 4.19. According to it, CUDA is 17 times faster than 6 threads AES-NI implementation, which makes it 94% more efficient in terms of duration. It is also 38 times faster than C++ implementation. 6 threads versions, which produce the best CPU performance, are implemented like AES-128 and AES-192 cases since both 8 threads versions perform worse than their 4 threads equal.

AES-NI is the efficient one among CPU implementations. With respect to thread counts, NI has 74%, 68%, 59%, 56% and 55% more efficiency than C++ in terms of duration. 6 threads version of AES-NI performs 6% and 16% more efficient than 4 and 8 threads in terms of duration. These ratios are 11% and 14% for 6 threads C++ implementation. With each increased thread count, AES-NI produces 35%, 24% and 6% better results in terms of duration while these ratios are 46%, 42% and 11% for C++.



	Kernel #5	1T NI	2T NI	4T NI	6T NI	8T NI	1T CPP	2T CPP	4T CPP	6T CPP	8T CPP
FILE WRT	20.691	20.378	20.680	20.844	20.620	20.029	20.533	21.068	20.956	20.873	20.831
MEMCPY	1.333										
CTR	0.100	3.651	2.378	1.796	1.681	2.010	13.771	7.457	4.327	3.847	4.486
TOTAL	22.124	24.029	23.058	22.640	22.301	22.039	34.304	28.525	25.284	24.721	25.317

Figure 4.19: CUDA, AES-NI and C++ AES-192 file encryption durations

4.3.4. Overview

File encryption experiments are conducted for 128, 192 and 256 bits of AES keys. A file with the size of 2.27GB is selected and it is encrypted for 25 times with 10 seconds of intervals between them. Results of CUDA, AES-NI and C++ versions are recorded. According to them, CUDA is by far the superior implementation for file encryption. It can encrypt the selected file 22, 19 and 17 times faster than the best CPU implementation with respect to AES key length. These ratios are around 4 times for CTR implementation. So, it can be said that CUDA is more efficient than CPU implementations when it comes to file encryption.

There is another advantage of CUDA on CPU implementations that almost all versions after 4 threads have %100 CPU utilization values. Reaching %100 utilization means that every available CPU resource is being used at that moment. After reaching this cap, increasing thread counts does not affect the performance of executions since

there is not any available resource. This creates a disadvantage in which users cannot be able to work on computers while a file is being encrypted. However, since GPUs exist inside the systems as a coprocessor, any kind of process working on them does not fully affect the users. For instance, this example can be applied to servers with GPUs in which every encryption process goes through CUDA greatly reduces the workload of CPUs on the server.

For this part of research, CUDA is tried with unified memory for file encryption. Unified memory concept is easy to use but creates a lot of workload to both RAM and global memory. Therefore, it slows down the encryption process. For this reason, another data transfer technique, which is direct copy from global memory, is used. Obtained results are compared and they show that kernel with direct memory copy operation is able to increase its efficiency by 33%. This comparison is done only for AES-128 and after finding out that it is faster, all CUDA experiments are run with this approach for file encryption. It is currently the best implemented choice.

AES-NI is still the superior CPU implementation as for it increases average efficiency of file encryption by 72%, 66%, 55%, 51% and 51% with respect to thread counts. However, CPU utilization cap creates an odd situation for file encryption experiment. After 4 threads, there exists a point in which increasing thread count does not reflect on increased performance. In fact, 8 threads versions experience a dropdown in performance. This was not the case for CTR and exhaustive search implementations. They also share the same type of suffering from CPU utilization gap, but they are able to increase their performance. This situation can be explained with the fact that basic difference of writing encrypted data to memory causes implementations with higher threads to slow down. In order to show this, 6 threads versions are also implemented for both AES-NI and C++. 6 threads versions are less affected than 8 threads as they are able to increase overall performance while 8 threads implementations fail to do so. In order to justify this as it only happens to file encryption implementations and not exhaustive search or CTR, additional experiments are carried out for them with 6 threads. Results indicate that 6 threads increase efficiency of exhaustive search and CTR both for AES-NI and C++ implementations considering 4 threads but not so much as 8 threads. In other words, they stand between 4 and 8 threads in terms of performance. This proved that this case only happens for file encryption due to memory allocations.

Figure 4.20 shows the performance difference of implementations with different key lengths in terms of how many GBs of data each implementation can encrypt per second. CUDA outperforms other implementations by a big margin like CTR case. However, the advantage of CPU for CTR case seems to decrease considering how efficient CUDA is for file encryption. For instance, when 128 bits of key is selected,

kernel #5 can encrypt 31.24 GBs of data per second without considering I/O operations and 6 thread AES-NI can only encrypt 1.41 GB. These values are 37.52 and 8.64 GBs for CTR. Therefore, CPU implementations are affected more severely than CUDA for file encryption. Results of other key lengths show similar outcomes in favour of CUDA. The reason behind CTR implementations showing higher throughput lies in the fact that every ciphertext is recorded to VRAM or RAM in the case of file encryption implementations. On the other hand, CTR versions only produces ciphertext without any record operation. That is why there is approximately 6 GBs of speed difference per second between CUDA versions of CTR and file encryption. This difference can be minimized with memory optimizations and better hardware equipment. Similar cases can be observed in other key lengths as well.

Table 4.14: CUDA, AES-NI and C++ file encryption results

Architecture	**AES**		**Cycles/byte**	**GB/sec**
NVIDIA GTX 970, 1.4 GHz	128		0.04	31.24
	192		0.05	26.59
	256		0.06	22.90
Intel i7 4770K, 3.9 GHz AES-NI	128	6T	2.57	1.41
		1T	4.86	0.75
	192	6T	2.61	1.39
		1T	5.38	0.67
	256	6T	2.68	1.36
		1T	5.82	0.62
Intel i7 4770K, 3.9 GHz CPP	128	6T	4.71	0.77
		1T	16.29	0.22
	192	6T	5.42	0.67
		1T	19.30	0.19
	256	6T	6.13	0.59
		1T	21.94	0.17

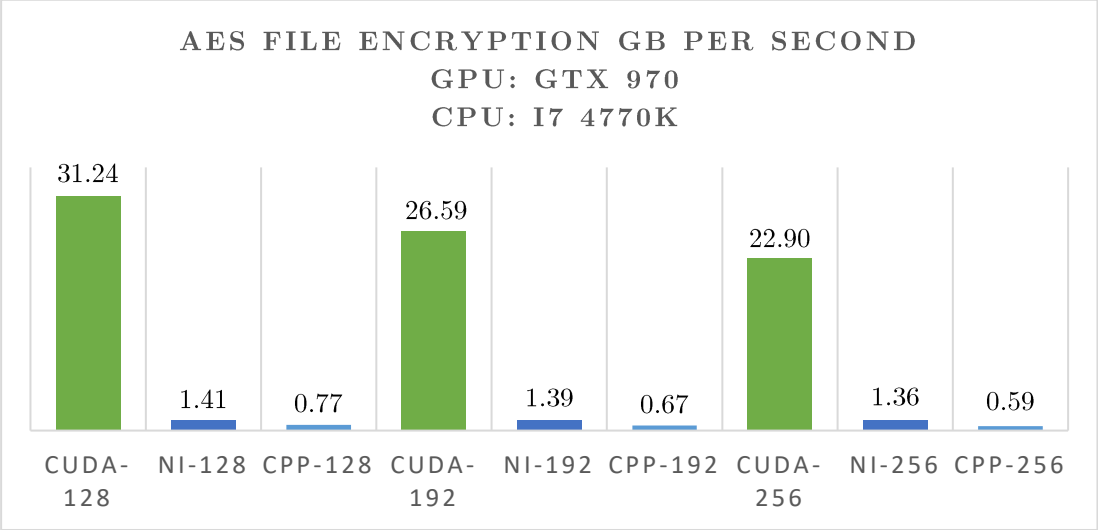


Figure 4.20: CUDA, AES-NI and C++ file encryption performance difference

CHAPTER 5

CONCLUSIONS AND FUTURE WORK

5.1. Conclusions

In summary, GPU usage has been increasing in the past years with the advancements in computer graphics and machine learning. They are widely used in embedded systems, mobile phones, personal computers and workstations. Its capability to harness power with parallel processing approach is very promising and it is used for gaming, image processing, deep learning training etc. Using GPUs for cryptographic purposes brings about a lot of performance throughput since these cards are basically designed to do simultaneous processing. Block ciphers are quintessential in terms of extracting power from parallel processing as they are easily adapted to be run in parallel. For this reason, AES is selected to be implemented in CUDA for experimenting how much performance gain can be achieved considering CPU implementations.

One of the fields that GPUs are used is full disk encryption. By using GPU, file encryption is greatly accelerated which shortens the time spent on full disk encryption. This is crucial for systems that constantly uses fully encrypted disks like militaristic organizations. Another advantage of using GPUs for file encryption is that it eases the workload of CPU. Since GPU exists in the system as coprocessor and does all the encryption work, CPU is free to be used by operation system. In this way, user experience is not affected by ongoing full disk encryption at all. Another advantage of GPU usage in cryptography can be considered as the same case in SSL servers. If all encryption is done on GPU for a busy SSL server, then the workload of that server can be spared to do something else in the meantime.

GPUs can also be used in implementations of AES based cryptographic algorithms like small-AES [45] and RECTANGLE. SHA-3 competition winner, Keccak, is also an AES based hashing algorithm. CAESAR competition winners are mostly AES based designs as well. Moreover, an alternative cryptocurrency which is named as X11 uses multiple rounds of 11 different hashes and some of them are AES based hashing algorithms like Keccak. All of these AES based cryptographic algorithms can be implemented for GPUs in order to increase the efficiency of output according to the results of this research.

Table 5.1: AES experiment results

	Architecture	AES	Cycles/byte	GB/sec
Exhaustive Search	NVIDIA GTX 970, 1.4 GHz	128	0.05	26.31
		192	0.06	21.95
		256	0.07	18.40
	Intel i7 4770K, 3.9 GHz AES-NI	128	2.96	1.23
		192	3.10	1.17
		256	3.59	1.01
	Intel i7 4770K, 3.9 GHz CPP	128	4.92	0.74
		192	6.77	0.54
		256	7.58	0.48
CTR	NVIDIA GTX 970, 1.4 GHz	128	0.03	37.52
		192	0.04	30.92
		256	0.05	26.25
	Intel i7 4770K, 3.9 GHz AES-NI	128	0.42	8.64
		192	0.47	7.71
		256	0.52	6.99
	Intel i7 4770K, 3.9 GHz CPP	128	3.81	0.95
		192	4.54	0.80
		256	5.27	0.69
File Encryption (CTR)	NVIDIA GTX 970, 1.4 GHz	128	0.04	31.24
		192	0.05	26.59
		256	0.06	22.90
	Intel i7 4770K, 3.9 GHz AES-NI	128	2.57	1.41
		192	2.61	1.39
		256	2.68	1.36
	Intel i7 4770K, 3.9 GHz CPP	128	4.71	0.77
		192	5.42	0.67
		256	6.13	0.59

GPUs can also be used in experiments like exhaustive search in this research. Since GPU offers a lot of power considering CPU, results of these experiments can be verified in practice. Distinguishers used in cryptanalysis are found in theoretical ways. In order to check whether the assumptions on these distinguishers are true or not, practical

experiments must be done. Therefore, GPUs can be used in these practical experiments since they have got higher computing capabilities. For instance, the first collision for SHA-1 is found with the help of GPUs [14]. In that research, GPU provided 65 times more performance than CPU which enabled this attack to be carried out in practice. Therefore, it is important to use GPU for cryptographic purposes.

Exhaustive search, CTR and file encryption experiments were carried out for 128, 192 and 256 bits of AES. For exhaustive search, 6 different CUDA kernels are implemented with different structures and they are compared with each other. According to Nsight results, kernel #5 is the fastest CUDA implementation. So, it is selected as the primal CUDA kernel for comparison with CPU implementations for every other case. CUDA is 21, 19 and 18 times faster than the best CPU implementation which is 8 threads version of AES-NI on exhaustive search case with respect to AES key length. This basically means that only 21 concurrently working CPUs can outmatch the power GPU has for AES-128 exhaustive search. Secondly, CTR experiment is done for both GPU and CPU implementations and according to the results, CUDA is 4 times faster than the best CPU implementation. This shows that CUDA lost most of its performance difference considering exhaustive search. However, it is still the best possible choice for CTR by a big margin. Final experiment is done on file encryption in which CUDA kernels perform just like exhaustive search. They are 22, 19 and 17 times faster than the best CPU implementations with respect to AES key length. While calculating file encryption durations, I/O operations, e.g. copying data from global memory of GPU to RAM, are not considered as only storing encrypted data to RAM for CPU and VRAM for GPU is important for indicating the performance difference between CPU and GPU. As a result of these experiments, it can be said that exhaustive search and file encryption is where CUDA outshines CPU by a big margin. Ergo, it is better to use CUDA for those kinds of situations.

The best results of cycles per byte and throughput (GB/sec) values for each implementation are shared in Table 5.1. According to it, CUDA implementations have fewer cycle per byte values with respect to CPU implementations. CUDA implementations are run on 1664 cores while CPU has only 4 cores. Considering the core number difference between implementations, CPU cores are much advanced and efficient than CUDA ones. However, when a lot of CUDA cores come together to work as a whole, they are more efficient in terms of overall cycle per byte values. Hardware implementation of AES-NI also shows fewer cycle per byte values with respect to C++ versions. This indicates the efficiency of AES-NI in which every round of AES is computed in hardware as special instructions.

Another advantage of CUDA is that GPUs can be installed on systems as a coprocessor next to CPU. So, when any operation is being processed by CPU, GPU

can do exhaustive search or file encryption on the background without affecting CPU workload. This is very advantageous as when 4 or more threads of CPU implementations are working, utilization becomes so high that the system could not be used since there is not resource available to user activities. This is not the case for CUDA. As a result of these findings, using CUDA for AES encryptions is advised since it greatly increases performance output especially on exhaustive search case.

5.2. Future Work

Both CUDA and GPU implementations can be improved further to increase their performance. The main problem for CUDA implementations is bank conflict issue on shared memory. This problem is partly resolved with extending the table, but S-box could not be fully extended, which is the case for kernel #5, due to the shortage on shared memory. This can be improved by trying a different GPU that has enough shared memory to extend every look-up table. This improvement will most probably increase efficiency as extending look-up tables shows that it decreases bank conflict values. Furthermore, exhaustive search versions of AES-192 and AES-256 along with every CUDA implementation of file encryption use more than 32 registers per thread. This case creates a disadvantage since occupancy ratio lowers when registers per thread value goes more than 32. So, there is a potential of occupancy increase in these kernels when they are optimized in terms of performance.

CPU implementations also can be improved in terms of performance, especially C++ ones. Results show that C++ implementations suffer more than AES-NI versions with thread counts bigger than 4 since Intel heavily optimized hardware implementations. In order to fairly compare C++ software and AES-NI hardware implementations, C++ versions need more optimization for better CPU execution. Furthermore, CPU used in this research only has 4 physical cores, thus, CPU implementations with more than 4 threads suffers from utilization cap. This cap can also be removed by using a better CPU with more cores. Furthermore, current file encryption implementation only allows files up to a certain size like 3 GB to be encrypted. This is because global ram of GTX 970 only has 4 GB of VRAM. Additional improvements can be done here so that bigger files can be encrypted by using 1 GBs of chunks. Moreover, file write speeds can be improved as well by using a better SSD with NVM express since SSD used in this experiment is fairly an old one. Moreover, the main bottleneck of CUDA file encryption comes from transferring data from global memory to RAM. Improving this transfer speed will greatly increase the performance of CUDA.

REFERENCES

[1] A. J. Menezes, P. C. van Oorschot, and S. a Vanstone, "Chapter 01: Overview of Cryptography," *Handb. Appl. Cryptogr.*, pp. 1–48, 1996.

[2] NIST., "FIPS 46-3: Data Encryption Standard (DES)," *Natl. Inst. Stand. Technol. NIST*, vol. 3, 2009.

[3] J. Katz and Y. Lindell, "Introduction to Modern Cryptography," 2007.

[4] E. Barker, W. Barker, and W. Burr, "Recommendation for Key Management," *NIST Spec. Publ. 800-57*, pp. 1–142, 2007.

[5] R. Salz, "The SWEET32 Issue, CVE-2016-2183 - OpenSSL Blog," *OpenSSL Blog*, 2016. [Online]. Available: https://www.openssl.org/blog/blog/2016/08/24/sweet32/. [Accessed: 11-Mar-2019].

[6] M. J. Dworkin, "Recommendation for block cipher modes of operation : National Inst of Standards and Technology Gaithersburg Md Computer Security Div," 2007.

[7] "Federal Information Processing Standards (FIPS) Publication 197 Announcing the ADVANCED ENCRYPTION STANDARD (AES)," *US Dep. Commer. Natl. Inst. Stand. Technol.*, 2001.

[8] National Institute of Standards and Technology, "Special Publication (NIST SP) - 800-21 2nd ed: Guideline for Implementing Cryptography in the Federal Government [Second Edition]," no. December, p. 97, 2005.

[9] J. Daemen and V. Rijmen, "The Design of Rijndael," *Springer, New York*, p. 255, 2002.

[10] S. Gueron, *Intel Advanced Encryption Standard (AES) New Instructions Set: White Paper*, no. May. 2006.

[11] K. Krewell, "What's the Difference Between a CPU and a GPU? | The Official NVIDIA Blog." [Online]. Available: https://blogs.nvidia.com/blog/2009/12/16/whats-the-difference-between-a-cpu-and-a-gpu/. [Accessed: 13-Mar-2019].

[12] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet Classification with Deep Convolutional Neural Networks," 2012.

[13] K. Powell, "Deep Learning and NVIDIA TITAN X, DIGITS DevBox | NVIDIA Blog." [Online]. Available: https://blogs.nvidia.com/blog/2015/03/17/digits-devbox/. [Accessed: 13-Mar-2019].

[14] M. Stevens, E. Bursztein, P. Karpman, A. Albertini, and Y. Markov, "The first collision for full SHA-1," 2017.

[15] C. Tezcan, "Brute Force Cryptanalysis of MIFARE Classic Cards on GPU," no. Icissp, pp. 524–528, 2017.

[16] G. Leurent and T. Peyrin, "From collisions to chosen-prefix collisions application to full SHA-1," *Lect. Notes Comput. Sci. (including Subser. Lect. Notes Artif. Intell. Lect. Notes Bioinformatics)*, vol. 11478 LNCS, pp. 527–555, 2019.

[17] R. Peddie, Jon; Dow, "Due to the collapse of Crypto-mining Q4 2018 goes down as worst fourth quarter for add-in boards in over 10 years." [Online]. Available: https://www.jonpeddie.com/store/add-in-board-report. [Accessed: 13-Mar-2019].

[18] "Steam Hardware & Software Survey: February 2019." [Online]. Available: https://store.steampowered.com/hwsurvey.

[19] "GeForce GTX 970 | Specifications." [Online]. Available: https://www.geforce.com/hardware/desktop-gpus/geforce-gtx-970/specifications. [Accessed: 18-Mar-2019].

[20] A. Biryukov and J. Großschädl, "Cryptanalysis of the full AES using GPU-like special-purpose hardware," *Fundam. Informaticae*, vol. 114, no. 3–4, pp. 221–237.

[21] D. A. Osvik, J. W. Bos, D. Stefan, and D. Canright, "Fast software AES encryption," *Lect. Notes Comput. Sci. (including Subser. Lect. Notes Artif. Intell. Lect. Notes Bioinformatics)*, vol. 6147 LNCS, pp. 75–93, 2010.

[22] K. Iwai, N. Nishikawa, and T. Kurokawa, "Acceleration of AES encryption on CUDA GPU," *Int. J. Netw. Comput.*, vol. 2, no. 1, pp. 131–145.

[23] N. Nishikawa, K. Iwai, H. Tanaka, and T. Kurokawa, "Throughput and power efficiency evaluations of block ciphers on Kepler and GCN GPUs," *Proc. - 2013 1st Int. Symp. Comput. Networking, CANDAR 2013*, pp. 366–372, 2013.

[24] J. Yang and J. Goodman, "Symmetric Key Cryptography on Modern

Graphics Hardware," *Adv. Cryptol. – ASIACRYPT 2007*, pp. 249–264, 2007.

[25] J. Gilger, J. Barnickel, and U. Meyer, "GPU-acceleration of block ciphers in the OpenSSL cryptographic library," *Lect. Notes Comput. Sci. (including Subser. Lect. Notes Artif. Intell. Lect. Notes Bioinformatics)*, vol. 7483 LNCS, pp. 338–353, 2012.

[26] G. Agosta, A. Barenghi, F. De Santis, A. Di Biagio, and G. Pelosi, "Fast disk encryption through GPGPU acceleration," *Parallel Distrib. Comput. Appl. Technol. PDCAT Proc.*, pp. 102–109, 2009.

[27] K. Akdemir, M. Dixon, W. Feghali, P. Fay, V. Gopal, J. Guilford, E. Ozturk, G. Wolrich, and R. Zohar, "Breakthrough AES Performance with Intel® AES New Instructions (White Paper). Intel Corp."

[28] L. Knudsen and M. Robshaw, "The Block Cipher Companion," 2011.

[29] J. Daemen and V. Rijmen, "AES Proposal : Rijndael," no. December, 2012.

[30] C. Nvidia, "Cuda c programming guide," *Changes*, 2012. [Online]. Available: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html. [Accessed: 18-Mar-2019].

[31] "An Even Easier Introduction to CUDA | NVIDIA Developer Blog." [Online]. Available: https://devblogs.nvidia.com/even-easier-introduction-cuda/. [Accessed: 19-Mar-2019].

[32] "CUDA Occupancy Calculator - Nvidia." [Online]. Available: https://developer.download.nvidia.com/compute/cuda/CUDA_Occupancy_calculator.xls. [Accessed: 20-Mar-2019].

[33] NVIDIA, "CUDA C Programming Guide (https://docs.nvidia.com/cuda/cuda-c-programming-guide/)," no. February, 2015.

[34] "CUDA Programming: BANK CONFLICTS IN SHARED MEMORY IN CUDA." [Online]. Available: http://cuda-programming.blogspot.com/2013/02/bank-conflicts-in-shared-memory-in-cuda.html. [Accessed: 25-Mar-2019].

[35] M. S. Nobile, P. Cazzaniga, D. Besozzi, D. Pescini, and G. Mauri, "cuTauLeaping: A GPU-powered tau-leaping stochastic simulator for massive parallel analyses of biological systems," *PLoS One*, vol. 9, no. 3, 2014.

[36] R. Landaverde, T. Zhang, A. K. Coskun, and M. Herbordt, "An investigation of Unified Memory Access performance in CUDA," *2014 IEEE High Perform. Extrem. Comput. Conf. HPEC 2014*, 2014.

[37] "NVIDIA Nsight Systems | NVIDIA Developer." [Online]. Available:

https://developer.nvidia.com/nsight-systems. [Accessed: 14-May-2019].

[38] "Achieved Occupancy | NVIDIA Nsight Systems." [Online]. Available: https://docs.nvidia.com/nsight-visual-studio-edition/5.6/Nsight_Visual_Studio_Edition_User_Guide.htm#Analysis/Report/CudaExperiments/KernelLevel/AchievedOccupancy.htm. [Accessed: 14-May-2019].

[39] "NVIDIA Nsight Visual Studio Edition User Guide - Instruction Statistics." [Online]. Available: https://docs.nvidia.com/nsight-visual-studio-edition/5.6/Nsight_Visual_Studio_Edition_User_Guide.htm#Analysis/Report/CudaExperiments/KernelLevel/InstructionStatistics.htm. [Accessed: 15-May-2019].

[40] "NVIDIA Nsight Visual Studio Edition User Guide - Branch Statistics." [Online]. Available: https://docs.nvidia.com/nsight-visual-studio-edition/5.6/Nsight_Visual_Studio_Edition_User_Guide.htm#Analysis/Report/CudaExperiments/KernelLevel/BranchStatistics.htm. [Accessed: 15-May-2019].

[41] "NVIDIA Nsight Visual Studio Edition User Guide - Issue Efficiency." [Online]. Available: https://docs.nvidia.com/nsight-visual-studio-edition/5.6/Nsight_Visual_Studio_Edition_User_Guide.htm#Analysis/Report/CudaExperiments/KernelLevel/IssueEfficiency.htm. [Accessed: 16-May-2019].

[42] "NVIDIA Nsight Visual Studio Edition User Guide - Achieved IOPs." [Online]. Available: https://docs.nvidia.com/nsight-visual-studio-edition/5.6/Nsight_Visual_Studio_Edition_User_Guide.htm#Analysis/Report/CudaExperiments/KernelLevel/AchievedIops.htm. [Accessed: 16-May-2019].

[43] "NVIDIA Nsight Visual Studio Edition User Guide - Pipe Utilization." [Online]. Available: https://docs.nvidia.com/nsight-visual-studio-edition/5.6/Nsight_Visual_Studio_Edition_User_Guide.htm#Analysis/Report/CudaExperiments/KernelLevel/PipeUtilization.htm. [Accessed: 16-May-2019].

[44] "UserBenchmark: Kingston RBU-SC400S37128G 128GB." [Online]. Available: https://ssd.userbenchmark.com/SpeedTest/15616/KINGSTON-RBU-SC400S37128G. [Accessed: 24-May-2019].

[45] C. Cid, S. Murphy, and M. J. B. Robshaw, "Small Scale Variants of the AES," pp. 145–162, 2010.

TEZ İZİN FORMU / THESIS PERMISSION FORM

ENSTİTÜ / INSTITUTE

Fen Bilimleri Enstitüsü / Graduate School of Natural and Applied Sciences ☐

Sosyal Bilimler Enstitüsü / Graduate School of Social Sciences ☐

Uygulamalı Matematik Enstitüsü / Graduate School of Applied Mathematics ☐

Enformatik Enstitüsü / Graduate School of Informatics ☐

Deniz Bilimleri Enstitüsü / Graduate School of Marine Sciences ☐

YAZARIN / AUTHOR

Soyadı / Surname : .....
Adı / Name : .....
Bölümü / Department : .....

TEZİN ADI / TITLE OF THE THESIS (**İngilizce** / English) : .....
.....
.....
.....
.....

TEZİN TÜRÜ / DEGREE**:** **Yüksek Lisans** / Master ☐ **Doktora** / PhD ☐

Tezin tamamı dünya çapında erişime açılacaktır. / Release the entire work immediately for access worldwide. ☐

Tez iki yıl süreyle erişime kapalı olacaktır. / Secure the entire work for patent and/or proprietary purposes for a period of **two year. *** ☐

Tez altı ay süreyle erişime kapalı olacaktır. / Secure the entire work for period of **six months**. ***** ☐

***** *Enstitü Yönetim Kurulu Kararının basılı kopyası tezle birlikte kütüphaneye teslim edilecektir. A copy of the Decision of the Institute Administrative Committee will be delivered to the library together with the printed thesis.*

Yazarın imzası / Signature ..... **Tarih** / Date .....