August 2013,

Master's Degree Thesis

Acceleration of Advanced Encryption Standard Algorithm on GPU Using CUDA C

Graduate School of Chosun University

Department of Computer Engineering

Saifullah

Acceleration of Advanced Encryption Standard Algorithm on GPU Using CUDA C

GPU 병렬 컴퓨팅 기반 고속 AES 알고리즘 설계

August 23,2013

Graduate School of Chosun University

Department of Computer Engineering

Saifullah

Acceleration of Advanced Encryption Standard Algorithm on GPU Using CUDA C

Advisor: Dr. Inkyu Moon

A thesis submitted in partial fulfillment of the requirements for a Master's degree

April 2013

Graduate School of Chosun University

Department of Computer Engineering

Saifullah

사이플라흐의 석사학위논문을 인준함

위원장 조선대학교 교수 이 상 웅 (인)

위 원 조선대학교 교수 이 지 은 (인)

위 원 조선대학교 교수 문 인 규 (인)

2013 년 5 월

조선대학교 대학원

Table of Contents

LIST OF FIGURES

LIST OF TABLES

ABSTRACT

Acceleration of Advanced Encryption Standard Algorithm on GPU Using CUDA C

Saifullah

Advisor : Prof. Inkyu Moon, Ph.D.

Department of Computer Science

Graduate School of Chosun University

Application of image utilization and processing has exploded in the past few years. Progressively effortless access to unauthorized data and increasingly powerful digital media manipulation tools has made multimedia security a very important issue. Many complementary techniques have been developed to address content security and digital rights management. Some of them are multimedia encryption/scrambling, digital holographic encoding and etc. The key outcome of this work is to propose and validate a fast and robust encryption of large data size files particularly images by using symmetrical Advanced Encryption standard (AES) algorithm. The obstacle in encrypting an image with the above mentioned techniques is the size of the input data, as the size of the data continues to grow, the speed of encryption must increase to keep up or it will become a bottleneck. The recent developments in parallel industry by GPUs have

shown to offer performance improvements versus conventional CPUs for data intensive problems. Cryptography is the main mechanism to secure digital information data. The encryption of multimedia data are very time consuming so for the pursuit of achieving better performance in terms of execution for cryptographic process, many researchers tried to use the graphical processing unit as a cryptographic co-processor. The spotlight of the research is to explore the compatibility of symmetric key cipher for multimedia data on graphics processor. The proposed methodology for GPU based AES surpassed the fastest CPU based implementation. The versatility, easily availability of GPU and the increase in performance opens new door for the cryptographic encoding of heavy data. In this project, we illustrated the performance of AES for multimedia data on three different platforms i.e. MATLAB, Visual Studio (C++) and GPU. Experimental results validating our approach of parallel AES are obtained with a prototype based on GPU implementation of the AES algorithm using NVIDIA GPU GeForce 310 processor and test samples of image size 256*256 pixels with each pixel having depth of 32 bits. The achieved outputs reflect the superiority of GPU over CPU by significant figures. Based on our results, we present an application characteristic to accelerator platform mapping, which can aid developers in selecting appropriate target architecture for their chosen application.

한글 요약

GPU 병렬 컴퓨팅 기반 고속 AES 알고리즘 설계

사이플라흐

지도 교수 : 문 인규.

컴퓨터공학과

대학원, 조선대학교

이미지 활용 및 처리 응용 프로그램은 지난 몇 년 동안 폭발적으로 증가했다. 권한이 없는 데이터에 대한 접근이 점차 쉬워지는 문제와 점점 강력해지는 디지털 미디어 조작 도구들이 멀티미디어 보안을 매우 중요한 이슈로 만들었다. 많은 보완 기술이 콘텐츠 보안 및 디지털 권한 관리를 해결하기 위해 개발되었다. 그들 중 일부는 멀티미디어 암호화 / 혼합화, 디지털 홀로그램 인코딩 등이다. 연구의 주요 결과는 대칭 고급 암호화 표준 (AES) 알고리즘을 사용하여 대규모 데이터 파일, 특히 이미지의 빠르고 강력한 암호화를 제안하고 검증하는 것이다. 위에서 언급된 기술로 이미지를 암호화할 때의 장애물은 데이터의 크기가 지속적으로 증가하는 것 같이 입력 데이터의 크기인데 데이터의 크기 증가를 따라가기 위해 암호화의 속도를 증가시켜야 하거나 또는 이것으로 인해 병목현상에 빠진다. GPU에 의한 병렬 산업의 최근 발전은 데이터 집약적 인 문제에 대한 기존의 CPU에 비해 성능 향상을 제공하는 것을 보여준다. 암호화는 디지털 정보 데이터를 보호하는 기본 메커니즘이다. 멀티미디어 데이터의 암호화 연구는 암호화 프로세스에 대한 실행의 측면에서 더 나은 성

능을 달성 추구하기 위해 많은 시간을 소비하고 있고, 많은 연구자가 암호화 보조 프로세서 (co-processor)와 같은 그래픽 처리 장치를 사용하려고 노력했다. 연구의 중점은 그래픽 프로세서 상에서 멀티미디어 데이터를 위한 대칭 키 암호의 호환성을 살펴보는 것이다. GPU 기반의 AES에 대해 제안된 방법은 CPU 기반의 가장 빠른 실행 결과를넘어섰다. GPU의 다용도성과 쉬운 가용성과 성능의 증가는 대규모 데이터의 암호화 인코딩을 위한 새로운 문을 연다. 이 프로젝트에서 우리는 세 가지 플랫폼 즉, MATLAB, 비주얼 스튜디오(C++), GPU에서 멀티미디어 데이터를 위한 AES의 성능을 보여준다. 병렬 AES의 우리의 접근 방식의 유효성을 검사하는 실험 결과는 NVIDIA GPU 지포스 310 프로세서를 사용한 AES알고리즘의 GPU 수행을 기반으로 한 프로토 타입과 함께 각각의 픽셀이 32bit인 이미지 크기 256*256의 테스트 샘플들을 얻는다. 얻어진 결과들은 상당한 수치로 CPU에 비해 GPU의 우수성을 반영한다. 우리의 결과를 바탕으로, 우리는 자신이 선택한 응용 프로그램에 대한 적절한 대상 아키텍처를 선택하는 개발자들을 도울 수 있는 가속기 플랫폼 매핑 응용 프로그램 특성을 제시한다.

1. Introduction

Multimedia information availability has increased dramatically with the advent of information technology industry. Multimedia content can be text, audio, still images, animation and video. But with this availability comes problems of maintaining the security of information that is displayed in public. Addressing this issue, many techniques have been proposed that are profoundly based on cryptography or phase encoding. The purpose of all such techniques is to provide confidentiality, availability, message integration between senders and receivers, implement accountability and accuracy. Cryptography [1] is the art of keeping information secret by transforming it into an unreadable format (encryption) by using special keys, then rendering the information readable again for trusted parties by using the same or other special keys (Decryption).

Multimedia content encryption has drawn more and more researchers and engineers, owing to the challenging nature of the problem and its interdisciplinary nature in light of challenges faced with the requirements of multimedia communications, multimedia retrieval, multimedia compression and hardware resource usage. Multimedia encryption involves changing the multimedia data-stream itself to ensure secure transmission of video data between client and server (or two nodes). It can be accomplished by means of standard symmetric key cryptography where multimedia bit-stream is treated as a binary sequence and the whole data can be encrypted using conventional crypto-system such as DES [2] and AES [3].

However, traditional ciphers, such as DES, RSA or AES, are difficult to be used directly in multimedia data encryption, since multimedia data are often of large-volumes [4] with real-time requirement. Practically, multimedia data, such as image, video or audio, are often compressed [5] before transmission or storing. Scaling single-thread performance without excessive power dissipation

has faced many difficulties [6] in recent past, forcing CPU vendors to integrate multiple cores onto a single die. A good solution to this problem is recently released new technology of GPGPU (general purpose computing on graphics processing units) [7], which is software/ hardware co-design and is getting a lot of popularity in accelerating general purpose processors, performing complex and intensive computations on accelerator hardware. Accelerators range from general purpose processors optimized for throughput over single-thread performance, through programmable, domain-specific processors optimized for characteristics of a particular application domain, to custom, application specific chips which are possibly implemented with reconfigurable hardware such as FPGAs [8-9].

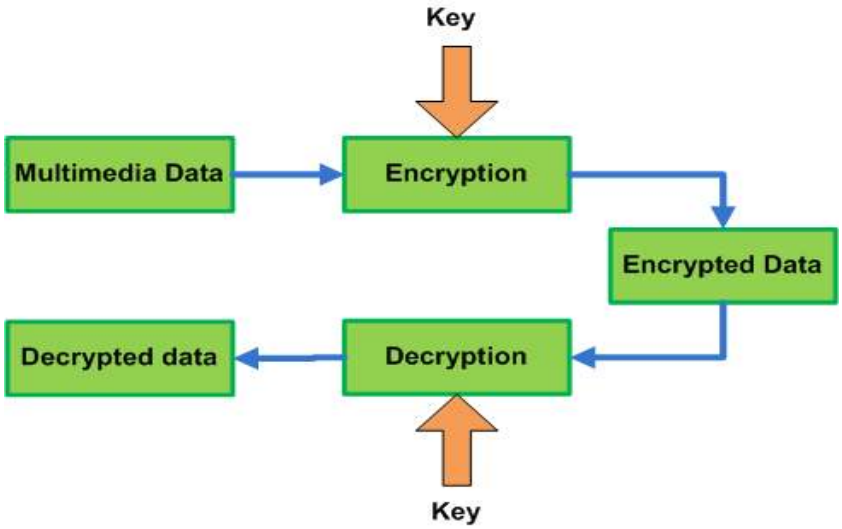

Figure.1. Simple Encoding Process

Accelerators' vast parallel computing resources and increasingly friendly programming environments make them good fits to accelerate compute-intensive and especially data parallel parts of applications. The process of executing AES in parallel manner can be divided into following five steps,

1. Carrying out the dependence analysis of a sequential source code in order to detect operations that can be processed in parallel loops
2. Finding the dependence vector for the loops
3. Selecting appropriate parallel methods and configuration of target hardware
4. Estimating the algorithm, GPU memory relations and the kernel requirements
5. Constructing the parallel forms of the source loops in accordance with the CUDA requirements.

The work explained in this research is concerned with the implementation of a robust AES which utilizes the Graphics Processing unit and shows improved performance over existing implementations. The Advanced Encryption Standard was chosen as a case study because the block cipher uses permutations and substitutions of data, rather than the arithmetic calculations which GPUs are known to excel in. The goal is to evaluate the potential of GPUs for encoding of image using AES and also to compare its performance with CPU implementation (C++ and MATLAB).

2. Related Work

Images are routinely used in diverse areas such as medical, military, science, engineering, art, entertainment, advertising, education as well as training. To encrypt digital images data, lots of encryption techniques have been proposed [10-11]. In most of the efficient image encryption techniques, many researchers utilized chaos systems to fulfill the demand of reliable and secure protection/storage/transmission of digital images over public networks. This is because of the fact that the chaotic signals have cryptographically desirable features such as high sensitivity to initial conditions/parameters, long periodicity, high randomness and mixing.

Accelerators such as FPGAs and GPUs, has demonstrated the ability to speed up a wide range of applications. Examples include image processing [12], data mining [13] and bio-informatics [14] for FPGAs, and linear algebra [15], database operations [16], K-Means [17], AES and DES encryption [18] and n-body simulations [19] on GPUs. Other work has compared GPUs with FPGAs for video processing applications [20], and similarly analyzed the performance characteristics of applications such as Monte-Carlo simulations and FFT [21]. NVIDIA's Compute Unified Device Architecture, or CUDA, and AMD's Compute Abstraction Layer, or CAL, are new language APIs and development environments for programming GPUs without the need to map traditional OpenGL and DirectX APIs to general purpose operations. Domain specific parallel libraries, such as a recent scan primitive's implementation [22] can be used as building blocks to ease parallel programming on the GPU.

Bielecki et al. [23] and Beletskyy et al. [24] used parallel programming as a way to increase the performance of the cryptographic algorithm, targeting at a series of algorithms like DES, 3DES, AES, IDEA, Blowfish, RC5, LOK191, GOST, and RSA. Focusing on the loop structures, they performed data dependency

analysis on loops and used loop parallelization technology with OpenMP. They observed that the execution time can be decreased significantly with the usage of symmetric multiprocessing (SMP). The research in [25] and [26] used a dedicated cryptographic coprocessor to alleviate the CPU from cryptographic workload. Although this way of implementation is several orders of magnitude faster than the software implementation, coprocessors lack the flexibility to support different parameters such as the key size or the mode of operations. Moreover, the silicon area will be increased and the system bus connecting the CPU and coprocessor forms a performance bottleneck. With the rapid development and increasing popularity of graphic processing unit (GPU), people tried to implement cryptographic applications on it due to the high-level parallelism this many-core structure provides. Harrison et al. [27] implemented AES Encryption ECB mode on GPU, taking advantage of its large number of simple processing units and stream processing. They mapped the AES algorithm onto GPU by implementing XOR using the Raster Operation Unit and fragment processor hardware. They showed that GPU can run AES with high efficiency and alleviate the cryptographic loads from CPU if used as a coprocessor.

3. Cryptography

Cryptography is generally understood to be the study of the principles and techniques by which information is converted into an encrypted version that is difficult (ideally impossible) for any unauthorized person to convert to the original information, while still allowing the intended reader to do so. In fact, cryptography covers rather more than merely encryption and decryption. It is, in practice, a specialized branch of information theory with substantial additions from other branches of mathematics. Cryptography is probably the most important aspect of communications security and is becoming increasingly important as a basic building block for computer security. The following four cryptographic goals form a framework from which other goals are derived:

1. Confidentiality is a service used to keep the content of information from all but those authorized to have it.
2. Data integrity is a service which addresses the unauthorized alteration of data.
3. Authentication is a service related to identification.
4. Non-repudiation is a service which prevents an entity from denying previous commitments or actions.

When disputes arise due to an entity denying that certain actions were taken, a means to resolve the situation is necessary. A fundamental goal of cryptography is to adequately address these four areas in both theory and practice.

3.1 Symmetrical Cryptography

In general, symmetric key algorithms [28] use a single, shared secret key. The same key is used for both encrypting and decrypting the data. There are two primary types of symmetric algorithms: block and stream ciphers. A block cipher is used to encrypt a text to produce a ciphertext, which transforms a fixed length of block data size into same length block of ciphertext in which a secret key and algorithm are applied to the block of data. For example, a block cipher might take a 64-bit block of plaintext as input, and output a corresponding 64-bit block of ciphertext. This transformation process should be conducted by a user providing a secret key and the decryption process is the inverse transformation to the ciphertext using the same key. AES, Blowfish, Data Encryption Standard (DES), Triple-DES, IDEA, Rijdael and RC2 are examples of symmetric block cipher.

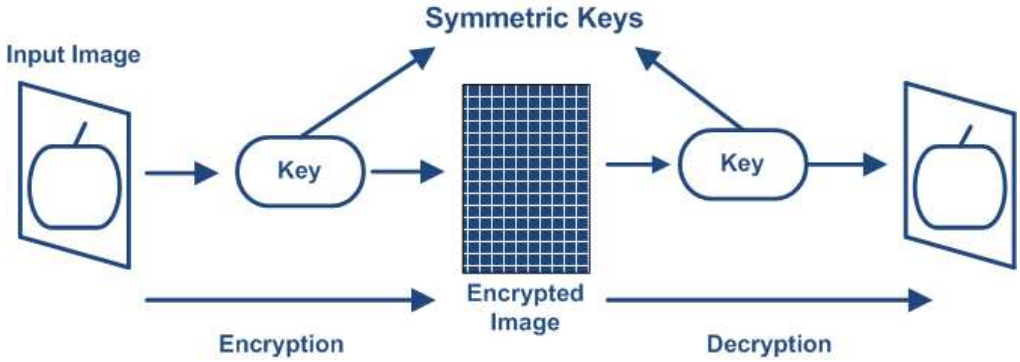

Figure.2. Symmetric Key Cryptography

3.2 Asymmetrical Cryptography

In the 1970s Martin Hellman, Whitfield Diffie, and, independently, Ralph Merkle invented a beautiful cryptographic idea [29]. Their idea was to solve

the key exchange and trust problems of symmetric cryptography by replacing the single shared secret key with a pair of mathematically related keys, one of which can be made publicly available and another that must be kept secret by the individual who generated the key pair. The advantages are obvious. First, no key agreement is required in advance, since the only key that needs to be shared with the other party is a public key that can be safely shared with everyone. Second, whereas the security of a symmetric algorithm depends on two parties successfully keeping a key secret, an asymmetric algorithm requires only the party that generated it to keep it secret. This is clearly much less problematic. Third, the issue of trusting the other party disappears in many scenarios, since without knowledge of your secret key, that party cannot do certain evil deeds, such as digitally sign a document with your private key or divulge your secret key to others.

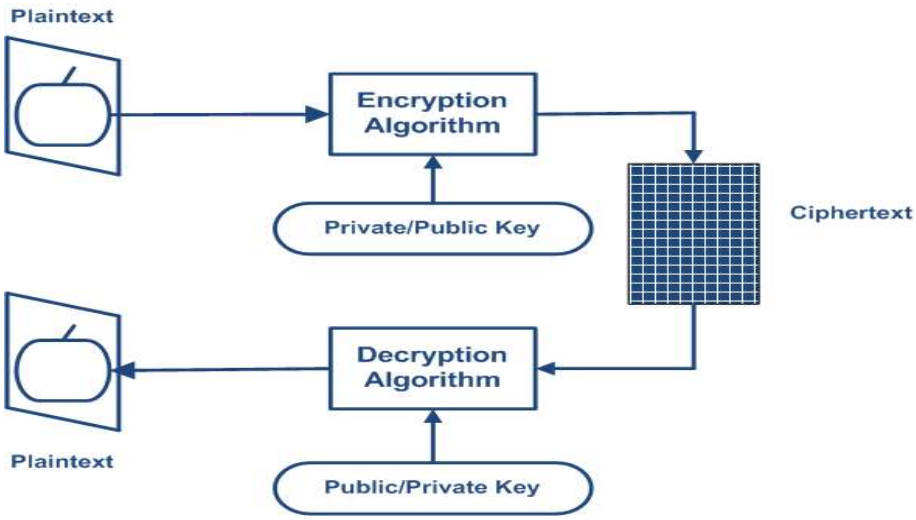

Figure.3. Asymmetric Key Cryptography

Asymmetric cryptography does not replace symmetric cryptography [30].

Rather, it is important to recognize the relative strengths and weaknesses of both techniques so that they can be used appropriately and in a complementary manner. Symmetric algorithms tend to be much faster than asymmetric algorithms, especially for bulk data encryption. They also provide much greater security than asymmetric algorithms for a given key size. On the down side, symmetric key cryptography requires that the secret key be securely exchanged and then remain secret at both ends. In a large network using symmetric encryption many key pairs will proliferate, all of which must be securely managed. Because the secret key is exchanged and stored in more than one place, the symmetric key must be changed frequently, perhaps even on a per-session basis. Finally, although symmetric keys can be used for message authentication in the form of a keyed secure hash, the full functionality of a digital signature requires asymmetric encryption techniques, such as RSA or DSA. As we shall see in the next chapter, a symmetric keyed secure hash algorithm can be used to implement a MAC (Message Authentication Code), which provides authentication and integrity but not non repudiation. In contrast, asymmetric digital signature algorithms provide authentication, integrity, and non repudiation, and enable the services of certificate authorities (CAs).

4. Advanced Encryption Standard

Joan Daemen and Vincent Rijmen urbanized a block cipher called Rijndael. In AES the span of each block and the key can be autonomously specified to be 128, 192, or 256 bits. In this paper we will only stress on block length and key length of 128 bits of AES. The AES arrangement exploits data of 128 bits and same three key size alternatives. This 128 bit data can be divided into four operation blocks, which are represented as a square matrix of bytes. These operation blocks are copied into a state array. The state array is organized as a 4×4 matrix. The data is conceded through Nr rounds (Nr = 10, 12, 14) for encryption [31]. These rounds are performed by the following transformations:

4.1.1. Byte-sub transformation

In this process 8-bit block is replaced with another 8- bit block, for substitution purpose we use S-box. This stage (known as SubBytes) is simply a table lookup using a 16×16 matrix of byte values called an **s-box**. This matrix consists of all the possible combinations of an 8 bit sequence. However, the s-box is not just a random permutation of these values and there is a well defined method for creating the s-box tables. The designers of Rijndael showed how this was done unlike the s-boxes in DES for which no rationale was given. We will not be too concerned here how the s-boxes are made up and can simply take them as table lookups.

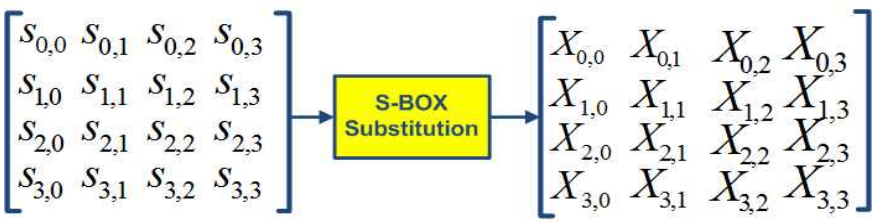

Figure.4. Sub-Bytes Transformation

The s-box is designed to be resistant to known cryptanalytic attacks. Specifically, the Rijndael developers sought a design that has a low correlation between input bits and output bits, and the property that the output cannot be described as a simple mathematical function of the input.

4.2 Shift-Rows Transformation

In this process we leave the first row of data, perform once shift left on 2nd row, two times shift left on 3rd row and three times shift left on 4th row. It is a simple Permutation.

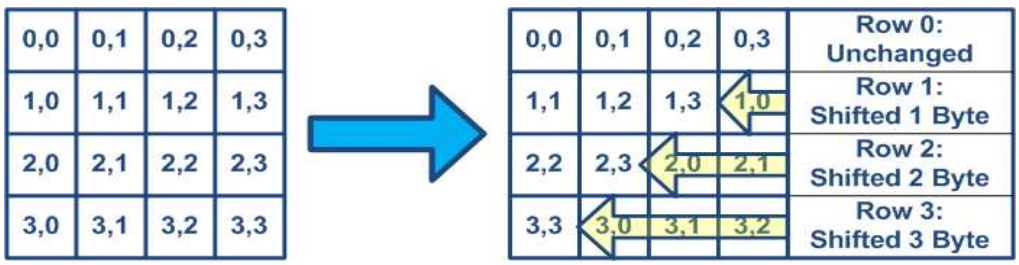

Figure.5. Shift-Rows Transformation

This operation may not appear to do much but if you think about how the bytes are ordered within **state** then it can be seen to have far more of an impact. Remember that **state** is treated as an array of four byte columns, i.e. the first column actually represents bytes 1, 2, 3 and 4. A one byte shift is therefore a linear distance of four bytes. The transformation also ensures that the four bytes of one column are spread out to four different columns.

4.3 MixColumns Transformation

This stage (known as MixColumn) is basically a substitution but it makes use of arithmetic of GF( ). Each column is operated on individually. Each byte of a column is mapped into a new value that is a function of all four bytes in the column. The transformation can be determined by the following matrix multiplication on **state**

Figure.6. Mix-Columns Transformation

4.3 AddRoundKey

In this stage (known as AddRoundKey) the 128 bits of **state** are bitwise XORed with the 128 bits of the round key. The operation is viewed as a columnwise operation between the 4 bytes of a **state** column and one word of the round key. This transformation is as simple as possible which helps in efficiency but it also effects every bit of **state**.

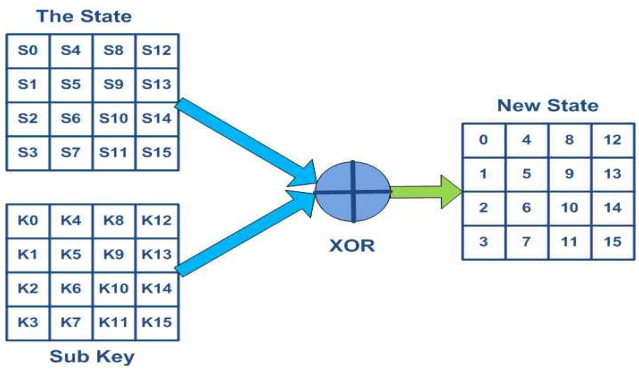

Figure.7. Add Round Key Transformation

5. Platform Overview

GPUs are inexpensive, commodity parallel devices with huge market penetration. They have already been employed as powerful co-processors for a large number of applications including games and 3-D physics simulation. The main advantages of the GPU as an accelerator stem from its high memory bandwidth and a large number of programmable cores with thousands of hardware thread contexts executing programs in a single program, multiple data (SPMD) fashion. GPUs are flexible and easy to program using high level languages and APIs which abstract away hardware details.

Compute Unified Device Architecture (CUDA) is an extension of C and an associated API for programming general purpose applications for all NVIDIA's architecture GPUs. CUDA has the advantage that is does not require programmers to master domain-specific languages to program the GPU. The GPU is treated as a coprocessor that executes data-parallel kernels with thousands of threads. Threads are grouped into thread blocks. Threads within a block can share data using fast shared-memory primitives and synchronize using hardware-supported barriers. Communication among thread blocks is limited to coordination through much slower global memory. The NVIDIA GeForce-310 GPU is comprised of 2 streaming multiprocessors (SMs). Each SM has 8 streaming processors (SPs), so there are total 16 CUDA cores.

5.1. Why GPGPU?

Commodity computer graphics chips, known generically as Graphics Processing Units or GPUs, are probably today's most powerful computational hardware. Researchers and developers have become interested in harnessing this power for general-purpose computing, an effort known collectively as GPGPU (for "General-Purpose computing on the GPU").

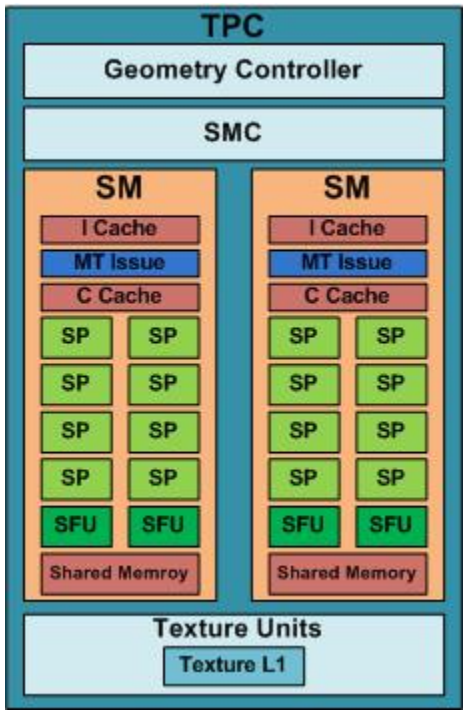

Figure.8. Texture Processor Cluster

5.2. Many-Core Architecture:

Most CPU has two or four cores on it, but the major GPU card has about 16 or even more cores on it. If we can find a way to divide a computation-sensitive problem to many parallel threads, it might get better performance to run on the GPU hardware. However the mapping is not straightforward. We may need to design some special data structures and modify the algorithm in the way we do CUDA programming. There are now two most famous general purpose GPU architecture, CUDA and Open-CL. The powerful compute capabilities of GPU stem from their vast availability of parallelism. CUDA is currently best suited for a SPMD programming style in which threads execute the same kernel but may communicate and follow divergent paths through that kernel. Designers have the

flexibility to trade-off performance for resources. For example, in massively parallel algorithms, hardware programmers might duplicate the same functional units many times, with only the die area limiting the level of parallelism.

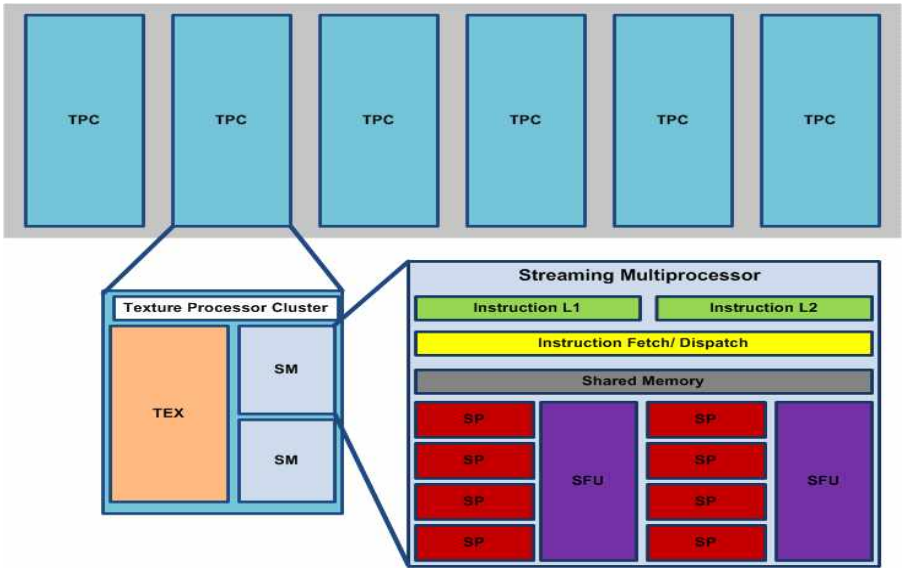

Figure.9. SM and SP of GPU

5.3. Parallel Architecture

Powerful and inexpensive: Recent graphics architectures provide tremendous memory bandwidth and computational horsepower. For example, the flagship NVIDIA GeForce 7900 GTX boasts 51.2 GB/sec memory bandwidth; the similarly priced ATI Radeon X1900 XTX can sustain a measured 240 GFLOPS, both measured with GPU Bench [BFH04a]. Nowadays GPU is not only the T&L (transform & lighting) and render hardware, but also the general purpose computation hardware.

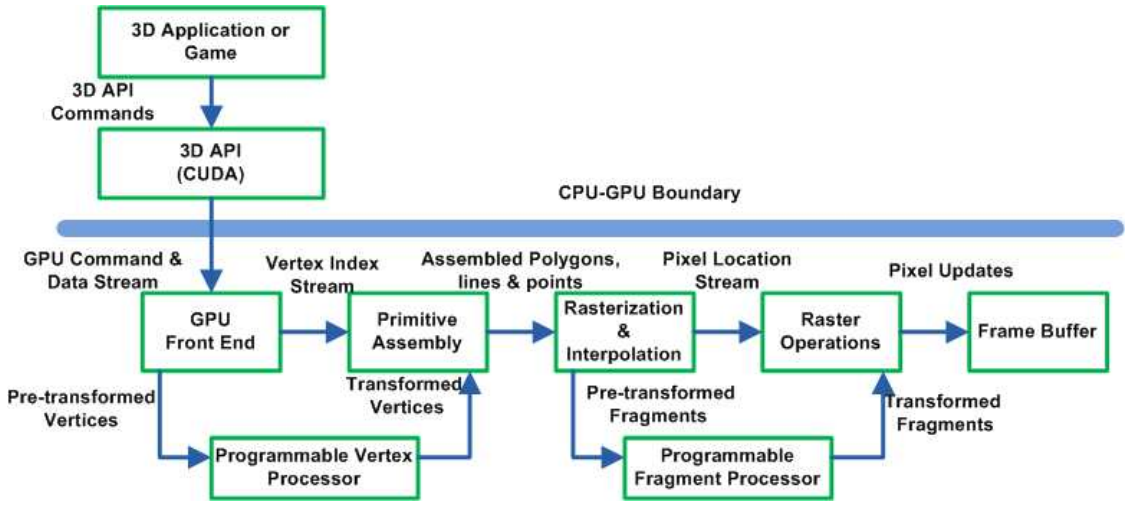

Figure.10. Graphics Pipeline of a GPU architecture

5.4. Compute Unified Device Architecture (CUDA)

The Compute Unified Device Architecture (CUDA) [32-33], proposed by NVIDIA for its graphics processors, exposes a programming model that integrates host (CPU) and GPU code in the same C++ source files. The main program introduced by the programming model is an explicitly parallel function invocation (kernel) which is executed by a user-specified number of threads. Every CUDA kernel is explicitly invoked by host code and executed by the device, while the host side code continues the execution asynchronously after instantiating the kernel.

The advent of multicore CPUs and manycore GPUs means that mainstream processor chips are now parallel systems. Furthermore, their parallelism continues to scale with Moore's law. The challenge is to develop application software that transparently scales its parallelism to leverage the increasing number of processor cores, much as 3D graphics applications transparently scale their parallelism to manycore GPUs with widely varying numbers of cores.

The CUDA parallel programming model is designed to overcome this challenge while maintaining a low learning curve for programmers familiar with standard programming languages such as C. At its core are three key abstractions - a hierarchy of thread groups, shared memories, and barrier synchronization - that are simply exposed to the programmer as a minimal set of language extensions. These abstractions provide fine-grained data parallelism and thread parallelism, nested within coarse-grained data parallelism and task parallelism. They guide the programmer to partition the problem into coarse sub-problems that can be solved independently in parallel by blocks of threads, and each sub-problem into finer pieces that can be solved cooperatively in parallel by all threads within the block.

This decomposition preserves language expressivity by allowing threads to cooperate when solving each sub-problem, and at the same time enables automatic scalability. Indeed, each block of threads can be scheduled on any of the available multiprocessors within a GPU, in any order, concurrently or sequentially, so that a compiled CUDA program can execute on any number of multiprocessors, and only the runtime system needs to know the physical multiprocessor count.

This scalable programming model allows the GPU architecture to span a wide market range by simply scaling the number of multiprocessors and memory partitions: from the high-performance enthusiast GeForce GPUs and professional Quadro and Tesla computing products to a variety of inexpensive, mainstream GeForce GPUs

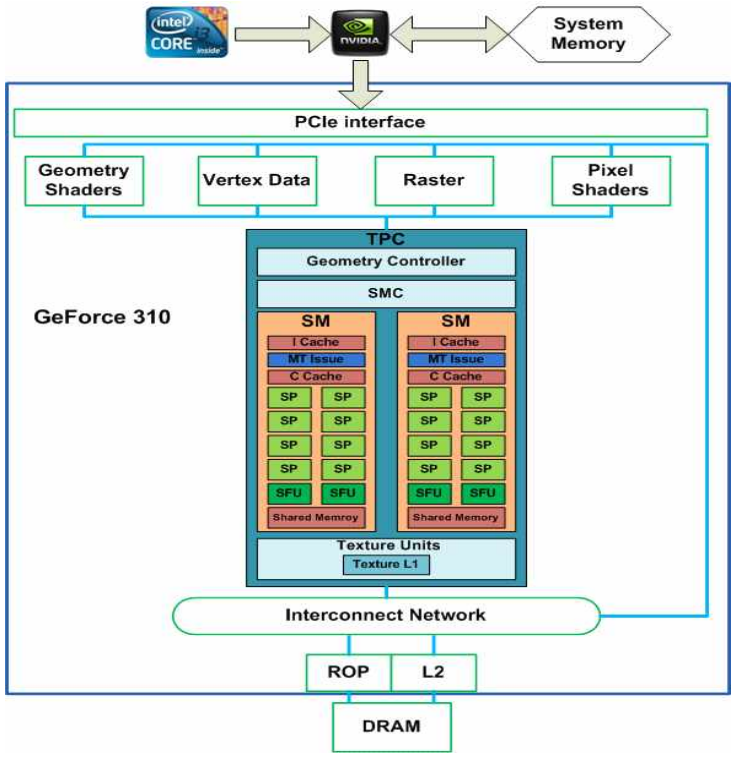

Figure.11.NVIDIA GeForce Graphical Interface

CUDA's runtime library provides programmers with a specific barrier statement, syncthreads(), but the limitation of this function is that it can only synchronize all the threads within a thread block. To achieve global barrier functionality, the programmer must allow the current kernel to complete and start a new kernel. This is currently fairly expensive, thus rewarding algorithms which keep communication and synchronization localized within thread blocks as long as possible. Fine-grained synchronization is also feasible so that execution units need only be synchronized with a select set of threads.

6. Parallel Advanced Encryption Standard Algorithm

6.1 Pre-processing

Block ciphers are one of the most important primitives in cryptography. They are based on well understood mathematical and cryptographic principles. Due to their inherent efficiency, these ciphers are used in many kinds of applications which require bulk encryption at high speed.

6.1.1. Data size

The input data is an image whose dimension is 256*256 pixels with each pixel having depth of 32 bit. So the total data size is 256*256*32 (N), equals to 2097152 bits. The image is further divided into corresponding blocks according to the algorithm which is discussed in the next section. The GPU exploited in the experiments has 16 CUDA cores and has the ability to accommodate maximum of 65536 blocks (B) in a grid (G) where each blocks can further accommodate maximum of 512 threads (T), so the total numbers of threads available in grid for processing is B*T (65536*512).

GPU configuration for AES: The input data size for AES will be N bits. According to the specification of the AES the input data size is 128 (s2) bits per block, so the number of block in the grid will be will be

$$\frac{N}{s2} = \frac{256*256*32}{128} = 16384 \text{ blocks}$$

where each block of AES have 128 bits or 4 pixels. For the GPU case there will be 16384 blocks in the grid where each blocks will further have 128 threads.

6.1.2. Electronic Codebook Mode (ECB)

The modes specify how data will be encrypted (protected) and decrypted (returned to original form). The modes included in this standard are the Electronic Codebook (ECB) mode, the Cipher Block Chaining (CBC) mode, the Cipher Feedback (CFB) mode, and the Output Feedback (OFB) mode. Out of these ECB is the basic mode so for our experiment this mode has been utilized.

The Electronic Codebook (ECB) mode is a basic, block, cryptographic method which transforms 128 bits of input to 128 bits of output. The analogy to a codebook arises because the same plain text block always produces the same cipher text block for a given cryptographic key. Thus a list (or codebook) of plain text blocks and corresponding cipher text blocks theoretically could be constructed for any given key. In electronic implementation the codebook entries are calculated each time for the plain text to be encrypted and, inversely, for the cipher text to be decrypted.

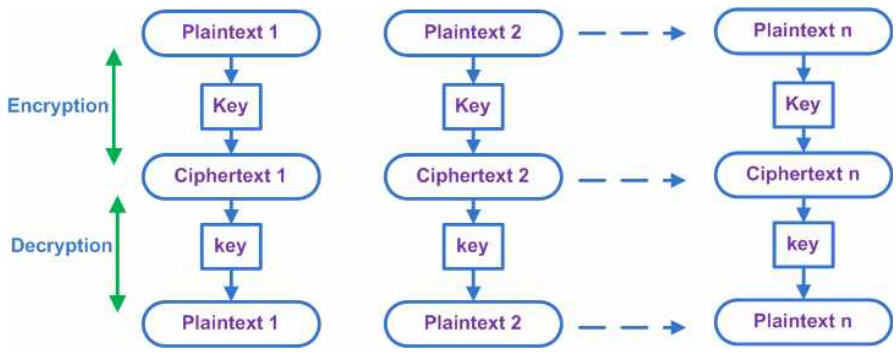

Figure.12. ECB Mode of Operation

6.2. Parallel Algorithm

6.2.1. Key Scheduling:

A key length of 128 bits involves 10 rounds, 192 bits entails 12 rounds and a key length of 256 bits entails 14 rounds. The key expansion algorithm must obviously generate a longer schedule for the 12 rounds required by a 192 bit key and the 14 rounds required by a 256 bit keys. Keeping in mind how we used the key schedule for the case of a 128 bit key, we are going to need 52 words in the key schedule for the case of 192-bit keys and 60 words for the case of 256-bit keys

For our project we are using 128-bit key, the key is also arranged in the form of a matrix of 4 × 4 bytes. As with the input block, the first word from the key fills the first column of the matrix, and so on. The four column words of the key matrix are expanded into a schedule of 44 words. Each round consumes four words from the key schedule. The figure below depicts the arrangement of the encryption key in the form of 4-byte words and the expansion of the key into a key schedule consisting of 44 4-byte words.

Key expansion takes place on a four-word to four-word basis, in the sense that each grouping of four words decides what the next grouping of four words will be. Let's say that we have the four words of the round key for the i^{th} round:

$$w_i, w_{i+1}, w_{i+2}, w_{i+3}$$

For these to serve as the round key for the i^{th} round, i must be a multiple of 4.These will obviously serve as the round key for the $(i/4)^{th}$

round. For example, w_4, w_5, w_6, w_7 is the round key for round 1, the sequence of words w_8, w_9, w_{10}, w_{11} the round key for round 2, and so on.

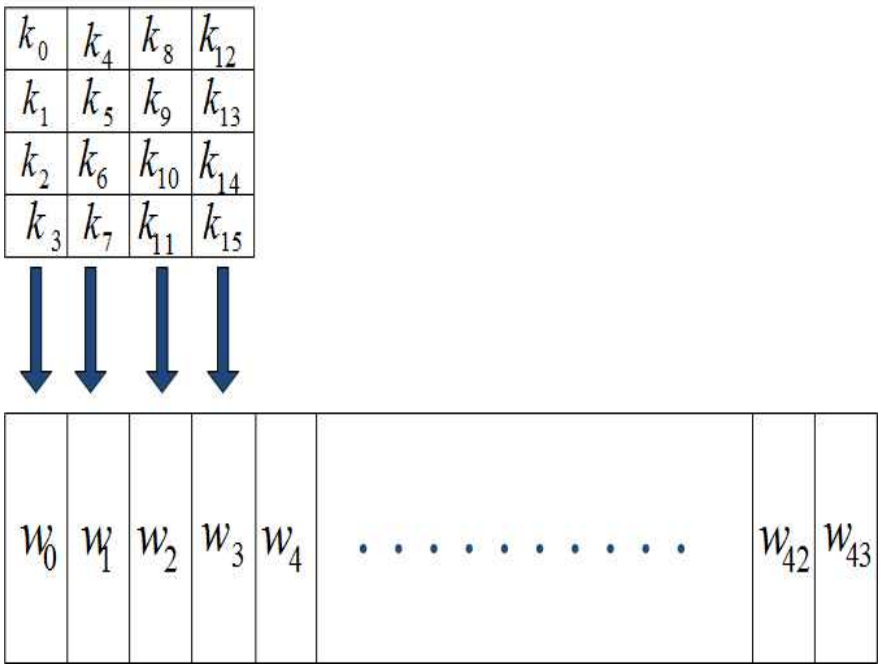

Figure.13. Key Scheduling

Now we need to determine the words $w_{i+4}, w_{i+5}, w_{i+6}, w_{i+7}$ from the words $w_i, w_{i+1}, w_{i+2}, w_{i+3}$.

$$w_{i+5} = w_{i+4} \otimes w_{i+1}$$
$$w_{i+6} = w_{i+5} \otimes w_{i+2}$$
$$w_{i+7} = w_{i+6} \otimes w_{i+3}$$

Note that except for the first word in a new 4-word grouping, each word is an XOR of the previous word and the corresponding word in the previous 4-word grouping.

So now we only need to figure out w_{i+4} This is the beginning word of each 4-word grouping in the key expansion. The beginning word of each round key is obtained by:

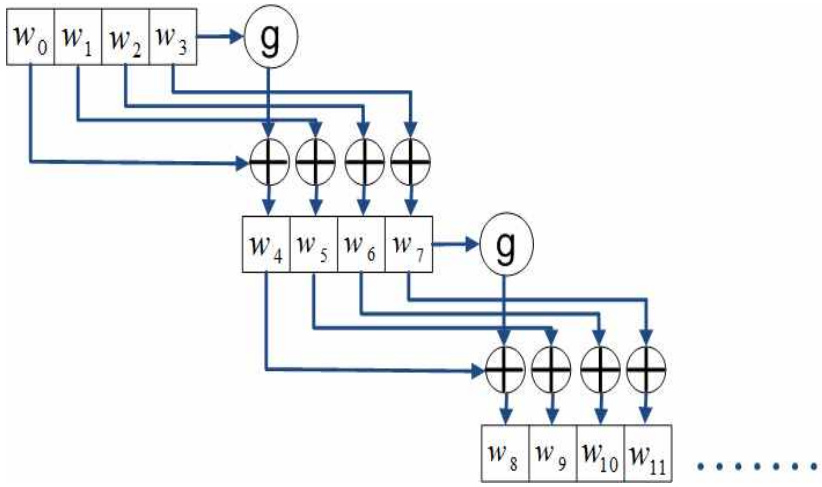

Figure.14. Round Key Expansion Algorithm

$$w_{i+4} = w_i \otimes g(w_i + 3)$$

That is, the first word of the new 4-word grouping is to be obtained by doing XOR'ing the first word of the last grouping with what is returned by applying a function g() to the last word of the previous 4-word grouping. The function g() consists of the following three steps: (i) Perform a one-byte left circular rotation on the argument 4- byte word, (ii) Perform a byte substitution for each byte of the word returned by the previous step by using the same 16 × 16 lookup table as used in the SubBytes step of the encryption rounds, and iii) XOR the bytes obtained from the previous step with what is known as a round constant. The round constant is a word whose three rightmost bytes are always zero. Therefore, XOR'ing with the round constant amounts to XOR'ing with just its leftmost byte.

The round constant for the i^{th} round is denoted Rcon[i]. Since, by specification, the three rightmost bytes of the round constant are zero, we can write it as shown below. The left hand side of the equation below stands for the round constant to be used in the i^{th} round. The right hand side of the equation says that the rightmost three bytes of the round constant are zero.

Rcon[i]=([RC[i],0,0,0)

The only non-zero byte in the round constants, RC[i],obeys the following recursion:

$$RC[1] = 1$$
$$RC[j] = 2 \times RC[j - 1]$$

The addition of the round constants is for the purpose of destroying any symmetry that may have been introduced by the other steps in the key expansion algorithm.

The key scheduling is performed on CPU before storing this into the constant memory of the GPU. As the same key will be used for the respective round of the algorithm, so the key for the ten rounds is scheduled and stored in the constant memory. Whenever the round will need the key for doing modulo-2 addition it will make a call to constant memory and after inspecting the round number, respective key will be provided to it.

6.2.2. Parallel Byte-Sub Transformation

The substitution process in symmetrical process plays an important role in both encrypting and decrypting the input data. This is a byte-by-byte substitution. The substitution byte for each input byte is found by using the same lookup table. The size of the lookup table is 16*16. To find the

substitute byte for a given input byte, we divide the input byte into two 4-bit patterns, each yielding an integer value between 0 and 15. (We can represent these by their hex values 0 through F.) One of the hex values is used as a row index and the other as a column index for reaching into the 16*16 lookup table.

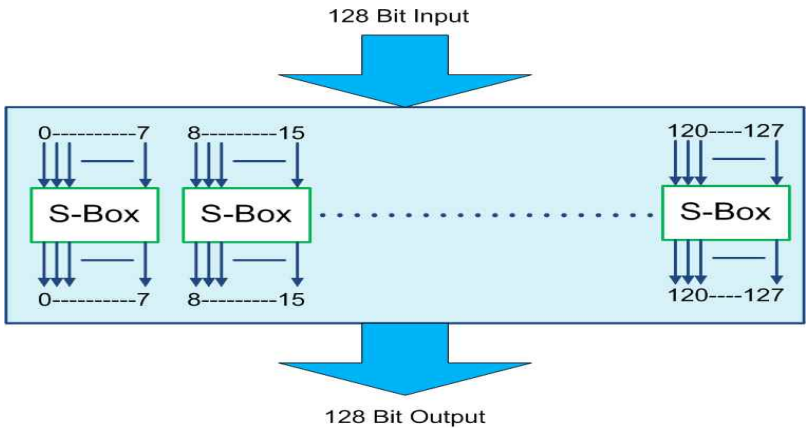

Figure.15. Parallel Byte-Sub Transformation

The goal of the substitution step is to reduce the correlation between input bits and output bits (at the byte level). The bit mangling part of the substitution step ensures that the substitution cannot be described in the form of evaluating a simple mathematical function. The total numbers of bits on which substitution operations will be performed are

$$\frac{N}{s2} * 128 = \frac{256 * 256 * 32}{128} * 128 = 2097152$$

Where N is the data size, s1 is the size of each block and 128 will be the bits in each block on which the operations will be performed. One thing to remember here that this is a parallel process and all the bits will take place

in the operations to substitute themselves with the new value chosen from the S-Box. The S-Box substitution will remain same for each round and also for each data set of 128 bit, so they will also be stored in the constant memory of the GPU and whenever their turn for contribution will arrive they will be called from the kernel and their execution will takes place.

6.2.3. Parallel Shift-Rows Transformation

This is where the matrix representation of the state array be- comes important. The ShiftRows transformation consists of (i) not shifting the first row of the state array at all, (ii) circularly shifting the second row by one byte to the left, (iii) circularly shifting the third row by two bytes to the left, iv) and circularly shifting the last row by three bytes to the left. This will be serial process. The function will be called whenever its turn for contribution will come. As the input block is written column-wise, so the first four bytes of the input block fill the first column of the state array, then next four bytes the second column, and so on. As a result, shifting the rows in the manner indicated scrambles up the byte order of the input block.

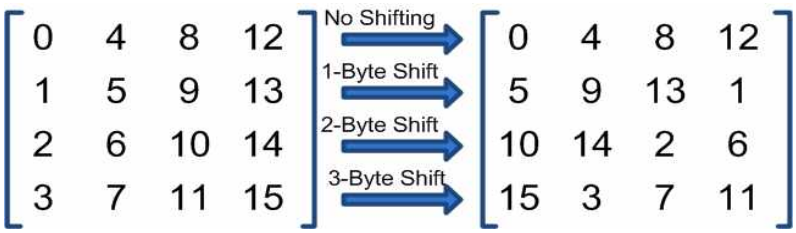

Figure.16. Parallel Shift-Rows Transformation

6.2.4. Parallel Mix-Columns Transformation

This step replaces each byte of a column by a function of all the bytes in the same column. More precisely, each byte in a column is replaced by two

times that byte, plus three times the next byte, plus the byte that comes next, plus the byte that follows. The words 'next' and 'follow' refer to bytes in the same column, and their meaning is circular, in the sense that the byte that is next to the one in the last row is the one in the first row. By 'two times' and 'three times', we mean multiplications in GF(28) by the bit patterns 000000010 and 00000011, respectively.

For the bytes in the first to fourth row of the state array, the operation can be stated as

$$
\begin{aligned}
s'_{0,j} &= (2 \times s_{0,j}) \otimes (3 \times s_{1,j}) \otimes s_{2,j} \otimes s_{3,j} \quad , \\
s'_{1,j} &= s_{0,j} \otimes (2 \times s_{1,j}) \otimes (3 \times s_{2,j}) \otimes s_{3,j} \\
s'_{2,j} &= s_{0,j} \otimes s_{1,j} \otimes (2 \times s_{2,j}) \otimes (3 \times s_{3,j}) \quad , \\
s'_{3,j} &= (3 \times s_{0,j}) \otimes s_{1,j} \otimes s_{2,j} \otimes (2 \times s_{3,j})
\end{aligned}
$$

Or in matrix form as

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

Figure.17. Parallel Mix-Columns Transformation

This portion is the most important part of the AES algorithm and if we see on the constant matrix then we can easily make a distinction that each row can be processed in parallel. This is a matrix multiplication whose parallel processing is achieved in a simple way.

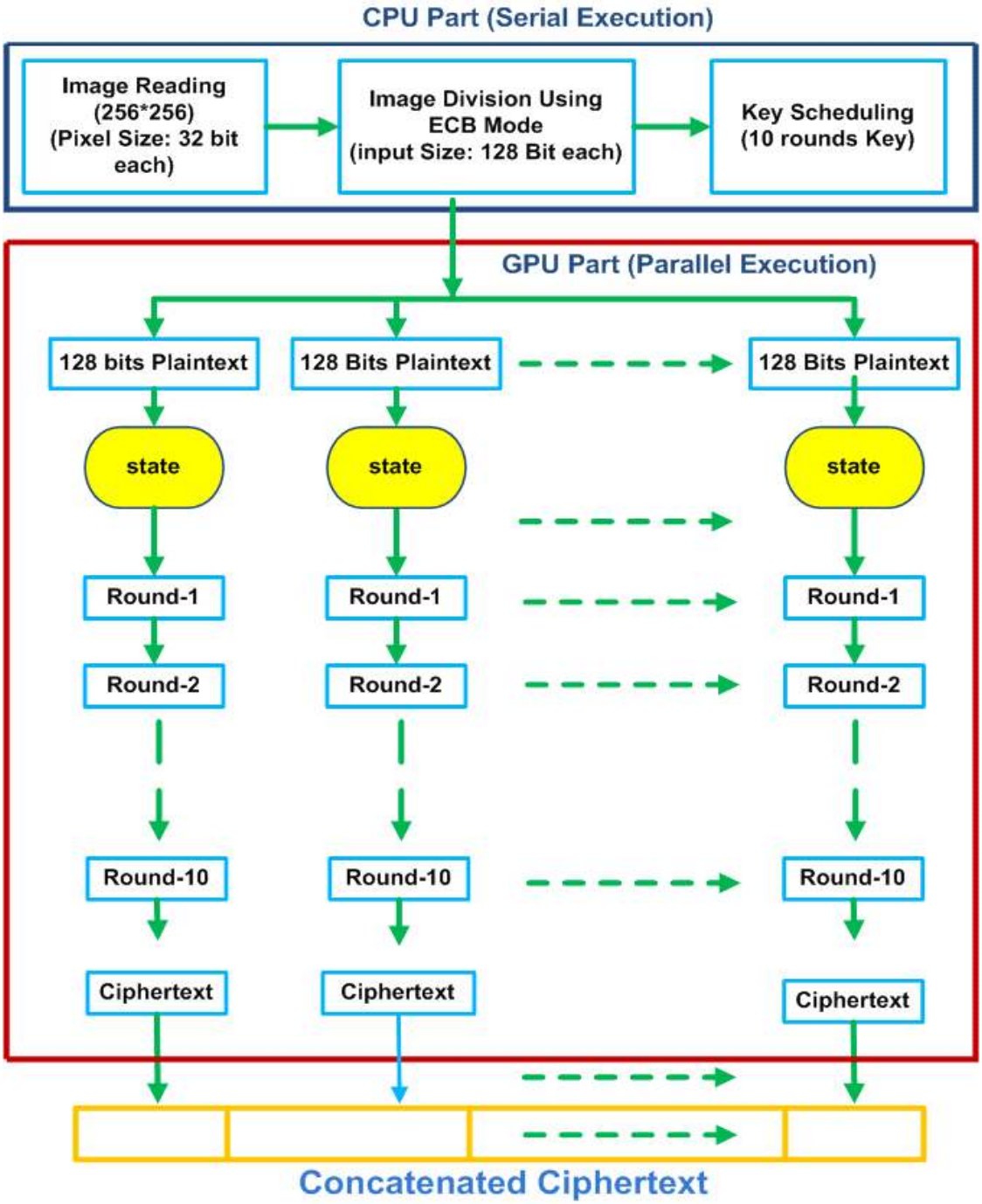

Figure.18. Parallel AES Algorithm

6.3. Summarize parallel Algorithm

1. Reading the input image using simple C++ language code (256*256).
2. Dividing the image into blocks of 128 bit each i.e. 1 pixel is 32 bit so 4 pixels will constitute one block of 128 bit.
3. Total no of input bits will be 256*256*32=2097152, therefore total no of input blocks will be 2097152/128=16384.
4. GPU Specification: there will be 16384 blocks (0-16383) with each block having 128 threads and each thread will contain one bits of input.
5. 128 threads in each block will constitute 4 columns of the state matrix as per AES specifications.
6. This strategy will help to fully utilize the concept of warps.
7. Allocating memory in GPU device and transferring data to GPU memory.
8. Round Key and S-Boxes data will also be transferred to GPU memory (constant).
9. After the initialization of kernel, all blocks will execute the first round in a parallel manner.
10. All the four process that is SubBytes, ShiftRows, Mixcolumns and AddRoundKey will be performed by all the blocks.
11. The remaining 9 rounds will be performed in a similar way.
12. The internal execution of each round is parallel but execution of 10 rounds will be serial as every new round depends on the output of its earlier round.

A multiprocessor is able to concurrently execute groups of 32 threads called warps. Since each thread in a warp may follow a different control flow, their execution paths may diverge due to the independent evaluation of conditional statements; in these cases the warp serially executes each path, disabling the computation for all threads that have not taken the one under execution. If

the control flow ever converges back, the warp is able to return to a single, parallel execution of all threads. Each multiprocessor executes warps much like the Single Instruction Multiple Data (SIMD) paradigm, as every thread is assigned to a different SP and every active thread executes the same instruction on different data. So now 16384 blocks will run in parallel for executing each round but the 10 rounds will be executed one after the other means serially.

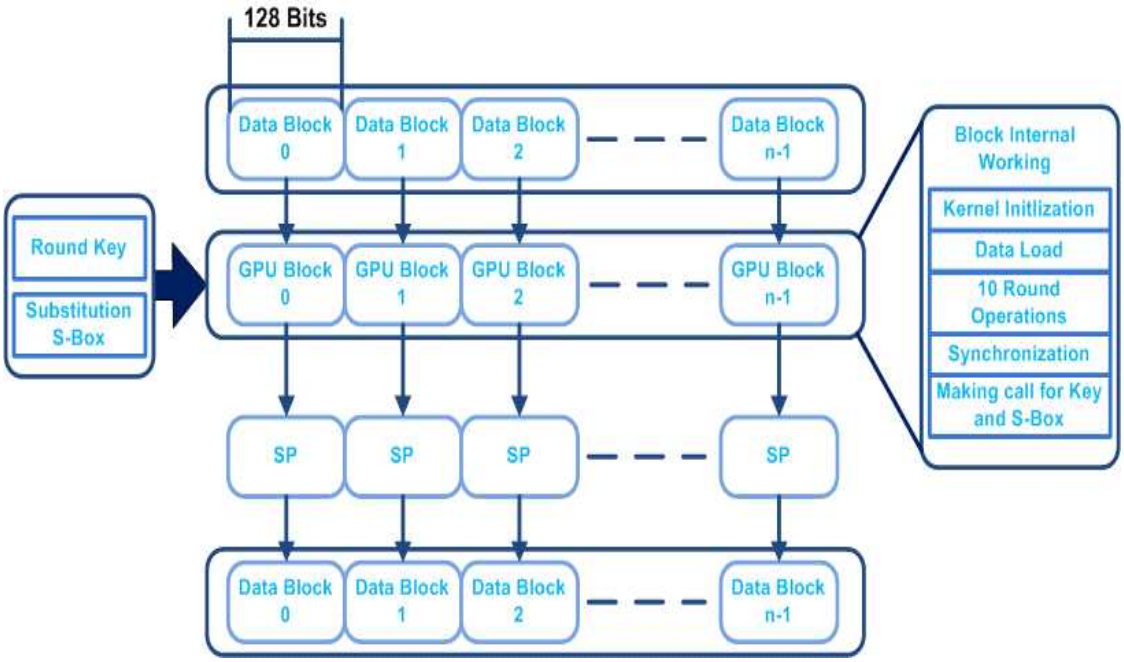

Figure.19. Parallel Flow of AES Algorithm in GPU Platform

7. Experimental Results:

The CUDA device accelerates the execution by harvesting a large amount of data parallelism. Data parallelism refers to the program property whereby many arithmetic operations can be safely performed on the data structures in a simultaneous manner. For the add round key, the output is generated by performing a XOR operaton between the state matrix and respective round key. As CUDA threads are of much lighter weight than the CPU threads, so these threads take very few cycles to generate and schedule due to efficient hardware support. This is in contrast with the CPU threads that typically require thousands of clock cycles to generate and schedule.

In CUDA, the host and devices have separate memory spaces. This reflects the reality that devices are typically hardware cards that come with their own dynamic random access memory (DRAM). In order to execute a kernel on a device, we allocated memory on the device and transfer pertinent data from the host memory to the allocated device memory, after device execution, the result is transferred from the device memory back to the host memory and free up the device memory that is no longer needed. The CUDA runtime system provides application programming interface (API) functions to perform these activities on behalf of the programmer.

In CUDA, a kernel function specifies the code to be executed by all threads during a parallel phase. Because all of these threads execute the same code, CUDA programming is an instance of the well known single-program, multiple-data (SPMD) parallel programming style [Atallah 1998], a popular programming style for massively parallel computing systems. As all the threads execute the same kernel code, there needs to be a mechanism to allow them to

distinguish themselves and direct themselves toward the particular parts of the data structure that they are designated to work on. So for this different threads will see different values in their threadIdx.x, threadIdx.y and threadIdx.z. Instead of having the loop increment like loop iteration, the CUDA threading hardware generates all of the threadIdx.x and threadIdx.y values for each thread. The code fragment uses the threadID=blockIDx.x * blockDim + threadIdx.x for one dimensional to identify the part of the input data to read from and the part of the output data structure to write on.

The objective of the experiments is to evaluate the performance of optimized multicore CPU implementation with a well designed GPU version and provide some insight into convincingly achievable speedups. All the experiments are conducted on multi core CPU, Intel Core i3 with installed memory of 3.00GB, endowed with an NVIDIA GeForce 310 with 512MB of global memory. This board has 16 computing cores clocked at 1.4GHz and is used on a PCI-Express 1.0 bus. The system is running windows 7 32bit and the CUDA Toolkit in use is version 4.0. For CUDA and C++ case all implementations were compiled in Visual Studio 2010 and for MATLAB case MATLAB 2010b release is utilized. All the collected results have been averaged over several trials for better results.

Implementing the GPU version carefully decrease the execution time. The comparative study proves the superiority of GPU due to its parallel architecture and show that GPU version of AES is approximately 8x faster than its counter C++ part, where as approximately 60x faster than its MATLAB counterpart. Regarding DRPE implementation on GPU, the GPU version is approximately 12x faster than C++ version whereas further 120x times faster than its MATLAB counterpart. The achieved results portray the dominance of parallel nature of GPU as compared to CPU.

S. NO	Process	AES Algorithm Operations		
		Add Round Key (Bits)	Mix Columns (Bits)	S-Box (Bits)
1	Parallel	$\frac{256*256*32}{128}*128 = 2097152$	$\frac{256*256*32}{128}*128 = 2097152$	$\frac{256*256*32}{128}*128 = 2097152$
2	Serial	Shift Rows		Total Operations
		$\frac{256*256*32}{128}*96 = 1572864$		(Parallel + Serial) * 10

Table.1. Total Number of Operations in AES Algorithm

S. No	Algorithm	Time (s)		
		MATLAB	C++	GPU
1	AES	1568.123	201.9	22.87

Table.2. Time Execution of AES Algorithm on Different platforms

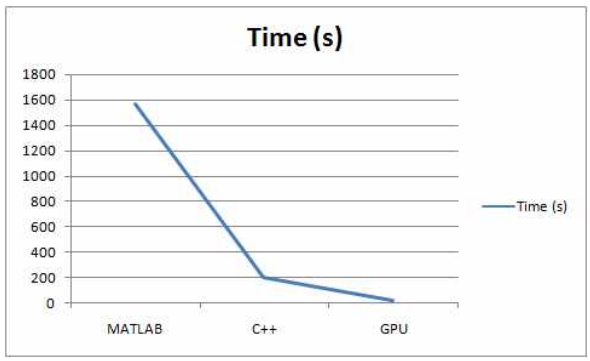

Figure.20. Pictorial View of Experimental Results

There has already been some work done for plaintext encryption using AES on GPU using CUDA [18] but their approach is different than us. They utilized the global memory and constant for storing round keys but in our approach we utilized only constant memory not only for storing round keys but also for S-boxes are stored in constant memory. In this way time of transferring data from host to device and vice versa will be saved and most importantly constant memory is fast as compared to global memory [34]. The second major advancement in our algorithm is that we used shared memory for storing the state matrix of the data, as state matrix is the only matrix which is being processed during the operations. In this way the data will be called directly from the shared memory. This approach gives us two advantages; (i) shared memory is fast [18], and (ii) global memory traffic will be slow.

Some other factors which can show the superiority of GPU are speedup, efficiency and redundancy. The speedup is found to b

$$S_p = \frac{T_1}{T_p} = \frac{1568.123}{22.87} \cong 68.6$$

where T_1 is execution time for serial processor and T_p is execution time for parallel process. The Efficiency is found to be

$$E_p = \frac{S_p}{p} = \frac{T_1}{pT_p} \cong \frac{68.6}{16} \cong 4.5$$

where p is number of core for the parallel process.

The results on the standard implementation of AES show a steady throughput regardless of the number of blocks per kernel because of the very low occupancy of the GPU memory resources. The results presented above show the maximum throughputs achievable, through calling the kernel with the 16384 number of CUDA-blocks (allowed 65535), in order to minimize the overhead induced by the kernel call to the system driver. The trend varying the number of thread per blocks shows that 128 threads are sufficient in order to completely utilize the GPU. Raising the number of threads per block does not yield significant performance advantages, even though the maximum throughput is reached at 512 threads per block because of algorithm specification of AES algorithm. Thus in addition to performance, development time is increasingly recognized as a significant component of overall effort to solution form the software engineering perspective.

8. Conclusion

Since AES was designed specifically for highly-optimized hardware implementations, its structure contains many operations which require computationally expensive adaptations in order to be executed by a general purpose CPU. Accelerators that are designed independently by different vendor's exhibit significant differences in hardware architecture, middleware support and programming models, which causes the processors designed for the same special task to favor differing subsets of applications. For example, programming methodologies range from direct hardware designs for FPGAs, through assembly and domain specific languages, to high level languages supported by GPUs [10]. These are widely different technologies and currently it is unclear which one is best suited to a given task.

Future computer systems will certainly include some accelerators, with the GPU and video processor the most common. Today, accelerators are primarily available as add-in boards. In the future they will probably be located on-chip with the CPU, thus reducing communication overhead. Different applications place unique and distinct demands on computing resources, and applications that work well on one processor will not necessarily map to another; this is even true for different phases of a single application.

From future perspectives a methodology for a quantitative comparison, especially one that distinguishes between fundamental organizational limits versus easily changed features, is an important area for further research. Also, GPUs are likely to migrate closer to the main CPU in future architectures. Issues related to this migration are also interesting areas for future work.

Reference

[1] B. Schneier, AppliedCryptography, 2-Edition, JohnWiley&son, Inc., NewYork, NY, 1996.

[2] National Institute of Standards and Technology (NIST), FIPS-46-3:Data Encryption Standard (DES)," http://www.itl.nist.gov/pspubs/, May 1999.

[3] J. Daemen, V. Rijmen, “AES Proposal: Rijndael”. Original AES Submission to NIST, 1999.

[4] Chen GR, Mao YB et al. A symmetric image encryption scheme based on 3D chaotic cat maps. Chaos, Solitons & Fractals 2004;21:749-61

[5] M. Droogenbroeck and R. Benedett, “Techniques for a selective encryption of uncompressed and compressed images,” in Proc. ACIVS, Ghent, Belgium, Sep. 2002.

[6] A. Iyer and D. Marculescu. Power-performance evaluation of globally asynchronous, locally-synchronous processors. In International SymposiumonComputerArchitecture, 2001

[7] S. Che, M. Boyer, J. Meng, D. Tarjan, J. Sheaffer, and K. Skadron, “A performance study of general-purpose applications on graphics processors using CUDA,” Journal of Parallel and Distributed Computing,
vol. 68, no. 10, pp. 1370–1380, October 2008.

[8] B. de Ruijsscher, G. N. Gaydadjiev, J. Lichtenauer, and E. Hendriks. FPGA accelerator for real-time skin segmentation. In Proceedings of the 2006 IEEE/ACM/IFIP Workshop on Embedded Systems for Real Time Multimedia, pages 93-97, 2006.

[9] Z. K. Baker and V. K. Prasanna. Efficient hardware data mining with the Apriori algorithm on FPGAs. In Proceedings of the 13th IEEE Symposium on

Field-Programmable Custom Computing Machines, pages 3-12, 2005.

[10] Zhang LH Liao XF, Wang XB. An image encryption approach based on chaotic maps, Chaos, Solitons & Fractals 2005;24:759-65

[11] M. S. Baptista, Cryptography with chaos, Phys. Lett. A 240 (1999) 50–54.

[12] H. Cheng and X. Li, "Partial encryption of compressed images and videos," IEEE Trans. Signal Process., vol. 48, no. 8, p. 2439, Aug. 2000.

[13] Z. K. Baker and V. K. Prasanna. Efficient hardware data mining with the Apriori algorithm on FPGAs. In Proceedings of the 13th IEEE Symposium on Field-Programmable Custom Computing Machines, pages 3-12, 2005.

[14] B. Harris, A. C. Jacob, J. M. Lancaster, J. Buhler, and R. D. Chamberlain. A banded Smith-Waterman FPGA accelerator for Mercury BLASTP. In Proceedings of the 2007 International Conference on Field Programmable Logic and Applications, pages 765-769, 2007.

[15] J. Kruger and R. Westermann. Linear algebra operators for GPU implementation of numerical algorithms. ACM Transactions on Graphics, 22(3):908-916, 2003.

[16] N. K. Govindaraju, B. Lloyd, W. Wang, M. Lin, and D. Manocha. Fast computation of database operations using graphics processors. In Proceedings of the 2004 International Conference on Management of Data, pages 215-226, 2004.

[17] S. Che, J. Meng, J. W. Sheaffer, and K. Skadron. A performance study of general purpose applications on graphics processors. In First Workshop on General Purpose Processing on Graphics Processing Units, 2007.

[18] Brandon p.Luken, Ming Ouyang, and Ahmed H.Desoky, AES and DES Encryption with GPU.

[19] L. Nyland, M. Harris, and J. Prins. Fast N-Body simulation with CUDA. GPU Gems 3, 2007.

[20] B. Cope, P. Y. K. Cheung, W. Luk, and S. Witt. Have GPUs made FPGAs redundant in the field of video processing? In Proceedings of the 2005 IEEE International Conference on Field-Programmable Technology, pages 111-118, 2005.

[21] L. W. Howes, P. Price, O. Mencer, O. Beckmann, and O. Pell. Comparing FPGAs to graphics accelerators and the Playstation 2 using a unified source description. In Proceedings of the 2006 International Conference on Field Programmable Logic and Applications, pages 1-6, 2006.

[22] S. Sengupta, M. Harris, Y. Zhang, and J. D. Owens. Scan primitives for GPU computing. In Proceedings of the Graphics Hardware 2007, pages 97-106, 2007

[23] Wnodzimierz Bielecki and Dariusz Burak, "Parallelization Method of Encryption Algorithms", Advances in Information Processing and Protection, 2008, pp. 191-204

[24] Vladimir Beletskyy and Dariusz Burak, "Parallelization of the IDEA Algorithm", Lecture Notes in Computer Science, VOL 3036, 2004, pp. 635-638

[25] Alireza Hodjat, "Interfacing a high speed crypto accelerator to an embedded CPU", Proceedings of the 38th Asilomar Conference on Signals, Systems, and Computers, 2004, pp. 488-492

[26] Patrick Schaumont , Kazuo Sakiyama , Alireza Hodjat , Ingrid Verbauwhede , "Embedded software integration for coarse-grain reconfigurable systems", Proceedings of the 18th International Parallel and Distributed Processing Symposium(IPDPS 2004), pp. 137-142

[27] Owen Harrison and John Waldron, "AES Encryption Implementation and Analysis on Commodity Graphics Processing Units", Proceedings of the 9th international workshop on Cryptographic Hardware and Embedded Systems, 2007, pp. 209-226

[28] P. Bilski, W. Winiecki, „Multi-core implementation of the symmetric cryptography algorithms in the measurement system," Measurement, No. 43, 2010, pp. 1049-1060.

[29] O. Baudron, D. Pointcheval, and J. Stern. Extended Notions of Security for Multicast Public Key Cryptosystems. In J. D. P. Rolim U. Montanari and E. Welzl, editors, Proceedings of the 27th International Colloquium on Automata, Languages and Programming (ICALP '2000), volume 1853 of Lecture Notes in Computer Science, pages 499{511, Geneva, Switzerland, 2000. Springer- Verlag, Berlin.

[30] W. Stallings, Cryptography and Network Security: Principles and Practices, 3rd edition, Prentice Hall, NJ, 2003.

[31] Federal Information Processing Standards Publication 197. United States National Institute of Standards and Technology (NIST). November 26, 2001. Retrieved October 2, 2012.

[32] J. Nickolls, I. Buck, M. Garland, and K. Skadron, Scalable parallel programming with cuda," ACMQueue,vol.6,no.2,pp.40{53,Mar.2008.

[33] NVIDIA Corporation, \CUDA Technology," http://www.nvidia.com/CUDA, Sep. 2008.

[34] D.B. kirk and W.W Hwu, Programming massively parallel processors: A Hands on Approach, Feb. 2010.

ACKNOWLEDGEMENTS

First and foremost, I offer my sincerest gratitude to my supervisor Prof. Inkyu Moon for his excellent guidance, caring and patience and providing me with comfortable research environment. His sincere encouragement to pursue my master's degree at 3DPIS (Three dimensional image processing system) Lab and his continuous support have been a great assistance to achieve a milestone in my career. Without his help, this thesis would not have been completed. He is the friendliest, easy going and encouraging advisor that anyone could wish for.

I would also like to show appreciation to all members of the thesis examining committee Prof. Sang Woong Lee and Prof. Ji-Eun Lee for their valuable advices and insight throughout my research. In addition, I would like to thank Department of Computer Engineering, Chosun University, to provide me the atmosphere to augment my knowledge.

My family members in Pakistan have always supported me with their distant love. The experience of being away from home helps me to realize the value and importance of family. Therefore, I am grateful to them and I would like to dedicate this thesis to my family. During my daily life in Korea, I have been supported by colleagues and cheerful friends, so I also want to extend my gratitude to all of them.