國立交通大學

資訊科學與工程研究所

碩 士 論 文

利 用 圖 形 處 理 器 加 速 3-DES 運 算

Using GPU to speed up 3-DES Computing

研 究 生：葉秀邦

指導教授：袁賢銘 教授

中 華 民 國 九 十 六 年 六 月

利用圖形處理器加速 3-DES 運算

研究生: 葉秀邦　　　　指導教授: 袁賢銘

國立交通大學資訊科學與工程研究所

摘要

加密與解密計算至今已經被開發出各式各樣的演算法，針對不同使用上的需求提供不同的計算方式，已經幾乎成爲網路傳輸中不可或缺的重要角色。然而，加密與解密的過程是需要龐大的 CPU 計算時間和資源，對一個支援加密傳輸的 web server 而言，往往 web server 本身會花 60%至 70%的資源在執行加解密的計算，對執行效率來說是很大的瓶頸。本研究探討這限制之最根本的原因，就是加解密過程耗費過多 CPU 資源，因此提出使用 GPU（圖形處理器）來加速這些加解密的計算，以減少 CPU 的負擔，讓 CPU 可以更專注於其他 web service 的服務。

本論文並非是目前爲止第一個利用了 GPU 的計算能力，來加速加解密運算之研究。然而加解密的密碼文件(cipher)種類相當多，我們針對計算量相當龐大的 3-DES 演算法進行研究，將用來原本該在 CPU 上之邏輯運算，轉換成可以在 GPU 上平行處理的演算法。在我們的實作中，隨著檔案大小的增加，我們觀察到在 GPU 提供了超過 5 倍目前 CPU 所能提供之運算能力，並大幅減少 CPU 所耗損的資源，有效的提供 web server 更好的服務品質。

Using GPU to speed up 3-DES Computing

Student: Hsiu-Pang Yeh Advisor: Shyan-Ming Yuan

Department of Computer Science and Engineering

National Chiao Tung University

Abstract

Various cryptography algorithms have been developed to date to provide different levels of data security for application domains, such as storage security, personal identification, and secure web browsing. Although these algorithms do a very good job of protecting your privacy, they consume massive amount of resource on the server-side while processing encrypting and decrypting requests from clients. Generally speaking, a web server supporting transport security (TLS/SSL) could spend up to 60%~70% of its computing resource in encrypting and decrypting data and leave 30%~40% for actual client request processing. Therefore in this research, we try to address the performance issue by using GPU (Graphics Processing Unit) to speed up the data encryption and decryption to reduce the computing resource spent on security and ultimately improve the web server throughput.

In this paper, we chose the widely-used 3-DES and implemented it on GPU. In our implementation, we observed the GPU cipher performs 5 times faster than the OpenSSL implementation on CPU. As a result, we show a promising direction for offloading the data encryption and decryption onto GPU.

Acknowledgement

首先我要感謝袁賢銘教授給我的指導，在我的研究領域裡給予我很多的意見，並且給予我最大的空間來發揮我的創意。也感謝所有幫助我的學長宋牧奇、林家峰，在我研究的過程中給我不少的指導跟建議。還有感謝我的同學羅國亨、徐俊傑等，在一起討論的過程中激盪出不少的想法。還有所有實驗室內提供電腦給我做實驗的同學，沒有你們我無法完成實驗。最後我要感謝我的爸媽，給予我這個良好的環境讓我求學生涯毫無後顧之憂，專心於學業，謹以這篇小小的學術成就來感謝您們的養育之恩。

Table of Contents

List of Figure

List of Tables

Chapter 1 Introduction

1.1. Preface

Cipher encryption could be the most important thing in internet security in recent ten years. As network technology evolved rapidly, the security problems need to be concerned seriously. Data Encryption Standard (DES) is proved as a standard method at 1976 [1]. Due to limitation of 64 bits key strength, DES has been superseded by 3-DES or the Advanced Encryption Standard.

3-DES could be seen as a transitional form between AES [2] and DES, it strengthen the DES's security, which is a stamp of approval possessing adequate security capability. The main function of 3-DES is manipulating DSE three times in a row. Theoretically, it can achieve more than 3 times of DES's security capability, and consumes CPU capability three times more vice versa. Therefore in many cases, it can't provide instant real time computing, which triggered emergence of the like AES highlighted the security and effectiveness capability.

Despite the newly generation of AES, the former computing methods still are irreplaceable. For instance, the newly version of OPENSSL still provide both DES and 3-DES for users [3]. The speed of 3-DES isn't fast enough in many circumstances though, it performs well in a none-instant-calculating condition. Many researchers still are aimed at improvements with these computing methods.

1.2. Motivation

The main streams of ciphers are DES, 3-DES and AES as before [3]. The lack of 64 bits key length of DES, a special effective way enables to decrypt within 24 hours allegedly, though it hasn't been proved, did alert the awareness of code safety. By increasing longer code or creating a more sophisticated computing method, we may guarantee the safety of coded data.

The advantages of AES, fast calculating speed, strong defense capability and easily parallelization, the latter one especially convenient for inventers, are the reasons made it popular.

The characteristics of 3-DES, 3 times more computing time and competitiveness of computing, are the inevitable result made its inferiority besides comparatively good quality of security as AES [4]. Most people turn to other ciphers which contain certain quality of speed and security. However, to invent new cipher involves quite long time. If focusing on improve the disadvantage of consumption of CPU resource in 3-DES, not only could fasten the DES but also provide more commodious user interface.

Chapter 2
Background and Related work

2.1. GPU (Graphics Processor Unit) introduction

Nowadays GPU is not only the T&L (transform & lighting) and render hardware, but also the general purpose computation hardware. This is the basic concept of general purpose computation on graphics hardware, also called GPGPU [5]. It means we can do lots of computing as CPU can do on GPU hardware. In fact, some kinds of computation have better performance on GPU than CPU, such as floating-point operation. The comparison of computation power between GPU and CPU in floating-points computation was depicted as Fig. 2-1. It is obviously that GPU and CPU do the same performance in floating-points computation in 2003. But in 2005, GPU do the twice performance than CPU in floating-points computation. However GPU can do triple or quadruple performance than CPU can do now.

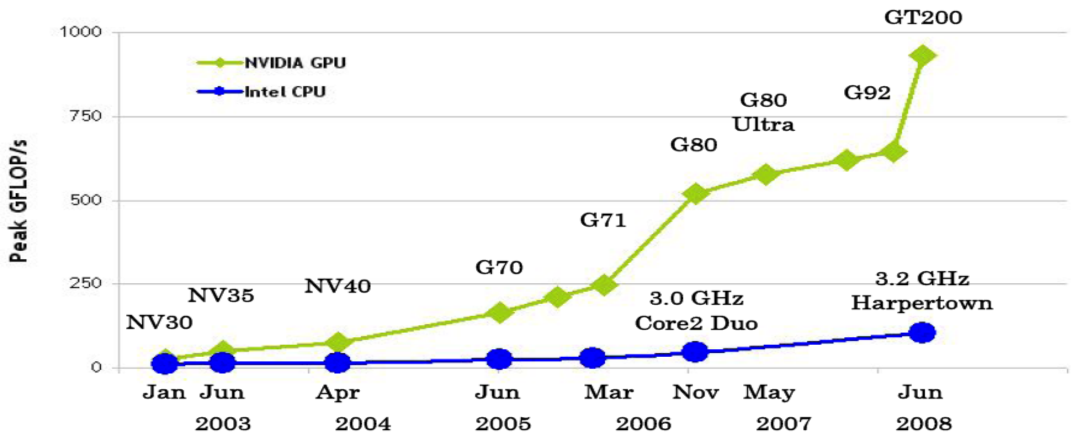

Fig. 2-1 Floating-Point Operations per Second for the CPU and GPU

Besides of floating-points computation, it is more easy and effective to do parallel processing on GPU hardware. The major CPU has two or four cores on it, but the major GPU card has about 128 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 has many constraints and is not straightforward. We may need to design some special data structures and modify the algorithm in the way we do graphics rendering. There are now two most famous general purpose GPU architecture, CUDA by NVIDIA and OpenCL by AMD/ATI.

2.2. CUDA (Compute Unified Device Architecture)

CUDA (Compute Unified Device Architecture) is the architecture to unify the general computation model on NVIDIA's graphics card devices [6]. Based on traditional C or C++ program language, CUDA is a new program language added some extension syntax and auxiliary libraries. The programmers can use the CUDA language to coding programming with ease just like C++ language programming [7]. Since we use CUDA to speed up our 3-DES encryption and decryption, we will discuss the advantage to CUDA architecture and enumerate main different between CUDA and OpenCL [8].

The objective of CUDA programming can be summarized as the following two parts:

- Easy to write program

CUDA programming is almost as easy as C++ programming. The main difference between CUDA and C++ programming is how to optimize your program. A programmer needs to know lots of hardware details about the graphic card while optimizing a CUDA program.

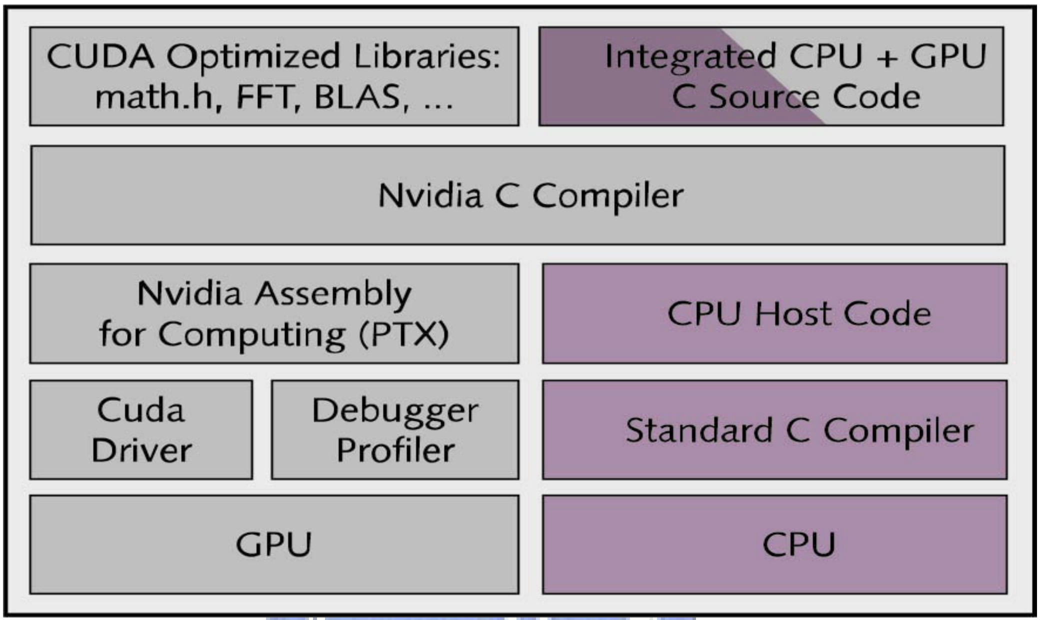

Fig. 2-2 CUDA platform for parallel processing on Nvidia GPUs

- Fit parallel processing

Unlike C++ programming for sequential processing, CUDA programming is suit for parallel processing. For instance, some loops need to be executed for hundreds or thousands times in sequential processing. But in CUDA programming, we can easily break a loop to hundreds or thousands threads. With parallel processing, sometimes we can get five to ten times performance than sequential processing [9].

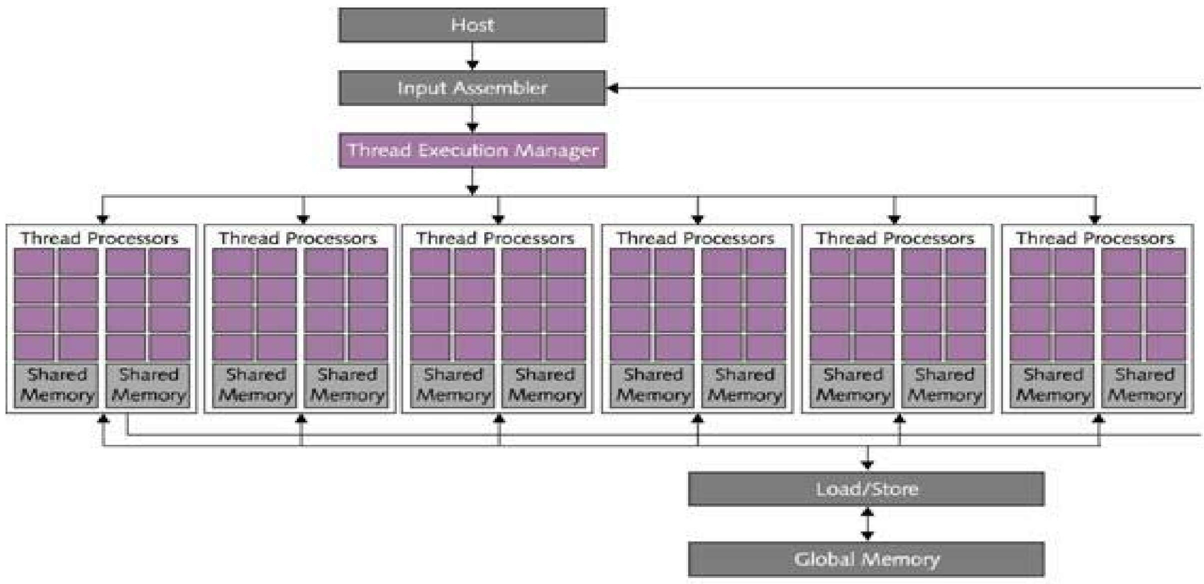

Fig. 2-3 CUDA Architecture from CUDA ZONE

- scattered write capability

Scattered write means that CUDA program codes can write to arbitrary addresses in GPU memory. In traditional GPU pipeline, it is impossible to assign any memory address by programmers. With the scattered write capability, it is more efficiently for us to build a parallel algorithm just like build an algorithm on C program [10].

- On chip shared memory

One special difference between NVIDIA and AMD/ATI is that NVIDIA GPU has shared memories on it [10]. With the shared memory, we can access data directly from shared memory without accessing the global memory on graphic card. Since it needs 4 hundred or 6 hundred cycles to access the global memory at a time, memory accessing is tone of key points to optimize parallel processing programs [11].

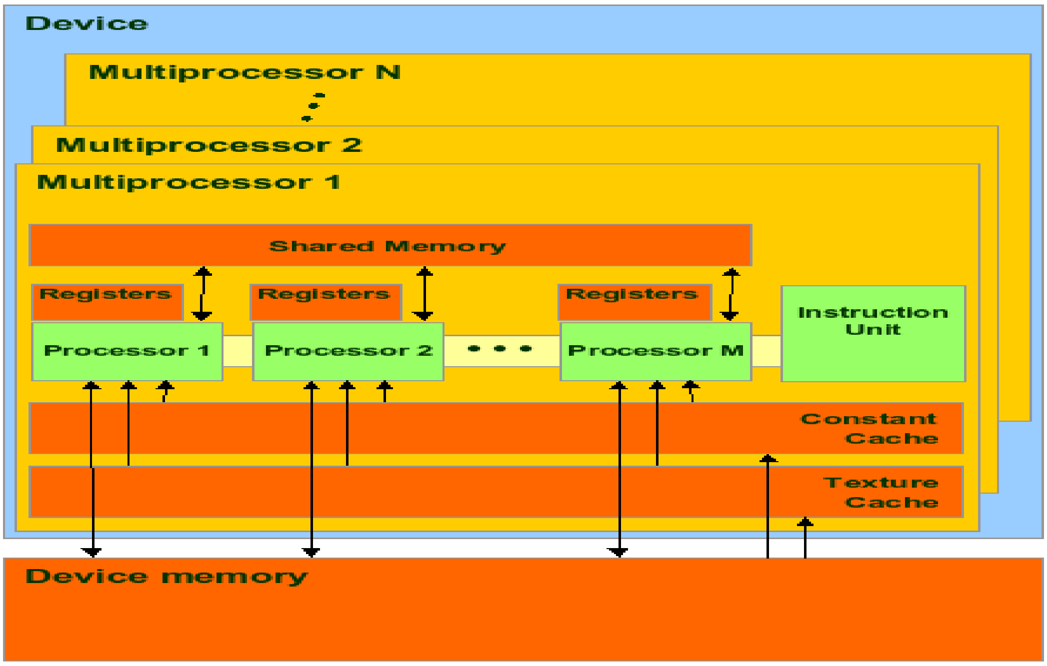

Fig. 2-4 CUDA Memory Architecture from CUDA ZONE

2.3. CUDA memory introduction

In fact, there are six different types of memory that can be access by programmer when writing a CUDA program.

Register's calculating speed is the fastest among all the other reservoirs. Most of the local variable of threads including array are manipulated by register spontaneously. In some conditions: Being "co-occupied" by too many variables that maxed out space of register breaching the limitation of compiler (using content N of --maxrregcount with N=128 presumed); using dynamic variables as indexes to access array (due to the need of order formation of array), it'll be replaced by local memory of compiler with lower speed [12]. The following we would talk about all kinds of memories on graphic card.

Shared memory's applied range is block, which only could be operated within the same block. The limitations of it are the scale and the necessity to synchronize threads, avoiding unexpected error of data filing. Its potency next only to register, is the focal point of optimization. Besides, we can't initialize it in the beginning process other than core implementation, nor access it in host machine. CUDA only provides API to assign its scale in current stage.

Local memory is the inevitable outcome while compiler automatically replacing data to global memory, similar to page swap in operating system. It has negative effect in efficiency, and the effect is hardly control. Therefore, when optimizing program, it has to be tracked by --ptxas-options to avoid it. Sometimes, to increase the number of blockDim in a block, we need --maxrregcount N to limit the usage amount of the maximum usable registers to each thread [13]. They have to compromise between the number of variables and the scale due to its necessity.

Other than device as a tag proclaim, reservoirs which are located by API directly through cudaMalloc also be seem as global memory. Global memory, which means all operating units could operate on it, and anything could be operate by threads in DRAM, threads in different block included. Its accessing without cache is differing from texture cache, which belongs to different ports in hardware and acquired coalesced read. (Coalesced read, is the constant block of reservoir in reading half the warp of thread, which synchronizes the controller of reservoir.)

Constant memory and texture cache belong to the same administrative level, but it has lower error rate cause of its size limitation. Despite the consuming of loading time in the very first time, its operating speed is as fast as shared memory. It can apply in all range and only can be access in the initial stage of setting a file or through the API in host machine.

Because of buffering of cache, texture memory doesn't acquire collaboration in accessing. The dozens computing cycles is the reason of slightly lower speed compare to global memory which access directly, but still much faster than global memory which doesn't collaborate while reading data. Therefore, texture memory is prior in a very sophisticated condition of collaborating accessing.

Texture memory is unrevised for cache's locality. CUDA provide not only 1D cache pattern, but also 2D and 3D texture cache which applies inclusively for the need of plotting.

The following table 2-1 and table 2-2 show the summary of GPU memory.

Type	Tag	Life Period	Access Scope	Hardware
Register	(none)	block	Thread R/W	On chip
Local memory	(none)	block	Thread R/W	DRAM
Shared memory	__shared__	block	Block R/W	On chip
Texture memory	(none)	program	Global R/W	DRAM + cache
Constant memory	__constant__	program	Global R	DRAM + cache
Global memory	__device__	program	Global R/W	DRAM

Table 2-1 Memory Types in CUDA

Type	Access time (clocks)	Performance Factor
Register	Immediately	none
Local memory	400 ~ 600	Compiler auto
Shared memory	4	Memory bank conflict
Texture memory	4, 400 ~ 600(miss)	Cache miss
Constant memory	4, 400 ~ 600(miss)	Cache miss
Global memory	400 ~ 600	Memory bank conflict

Table 2-2 Memory Access time in CUDA

2.4. Related Work

In recently years, more and more people try to speed up the compute-sensitive work in CUDA programming. This is because graphic card is much cheaper than other hardware device. User just needs to buy the NVIDIA graphic card and install their driver. In additional, users may get better performance by offering a good parallel algorithm in CUDA programming. Moreover, there is some commercial software supported to CUDA. For example, TMPGENC's video encoder software supports CUDA speed up in 2008. That is why some people say multi-cores is the final goal for CPU [14].

2.4.1. Related Work: AES speed up in CUDA

It is fully benefits of the most optimized known AES techniques in the CUDA-AES implementation. The CUDA-AES implementation is designed for 32-bit processors. Given the flexibility of memory model, it is possible to efficiently use the four T-look-up tables each one containing 256 entries of 32 bits each. The CUDA-AES implementation is based on combination of the round stages, which allows a very fast execution on processors with word length of 32 bits. In this paper, the author brings up a formula that stands for his AES architecture [15].

$$e_j = T_0\lfloor a_{0,j} \rfloor \oplus T_1\lfloor a_{1,j+1} \rfloor \oplus T_2\lfloor a_{2,j+2} \rfloor \oplus T_3\lfloor a_{3,j+3} \rfloor \oplus k_j$$

Where a is the round input, e is the round output of bytes, T[ ] is a look-up table, $\oplus$ means XOR and Kj is one column of the stage key. This solution takes only 4 look-ups and 4 XORs per column per round [16].

It is obviously that the CUDA-AES implementation takes few operation in a round rather than DES. That is the reason why AES encryption is as fast as DES encryption. Furthermore, AES is more suitable for CUDA programming because of using massive matrix. It can easily parallelize all the AES system by spreading up the linear operations in matrix to corresponding threads in GPU hardware [17]. This CUDA-AES implementation in this paper speeds up the about 5 times on G80. The detail data statistics will be analyzed with our results. We put all the data statistics in chapter four.

Chapter 3 System Architecture

In this chapter, we present the system overview of our framework that includes DES and 3-DES (Triple DES) architecture. Additionally, we not only do the encryption and decryption speed up work but also combine the 3-DES to web service. We will describe the whole system in the last subsection.

3.1. DES and 3DES Introduction

3.1.1. DES Architecture

Since that DES is the main core of 3-DES operation, so we first analysis the DES architecture to find out how many components in DES can be improved by CUDA [14]. DES is a block cipher system with a secret key. The principle of DES is that divide the plaintext into many 64-bits blocks. First, each of 64-bits blocks is subjected to an initial permutation IP. Then do sixteen times logic operation with sixteen sub-keys. Finally we do IP^{-1} operation to this text to get the final cipher text, so we can send the 64-bits cipher text to other people with DES encryption [18].

In traditional DES architecture is as the Fig. 3-1, all the operations including input, initial permutation, permuted input, sub-key generation, IP^{-1} operation, and output are all achieved by CPU hardware [19]. Therefore, all logic operations in CPU waste too many CPU time and CPU recourses. Our job is to assign some operations form CPU to GPU to speed up the whole system and decrease the CPU recourses wastage.

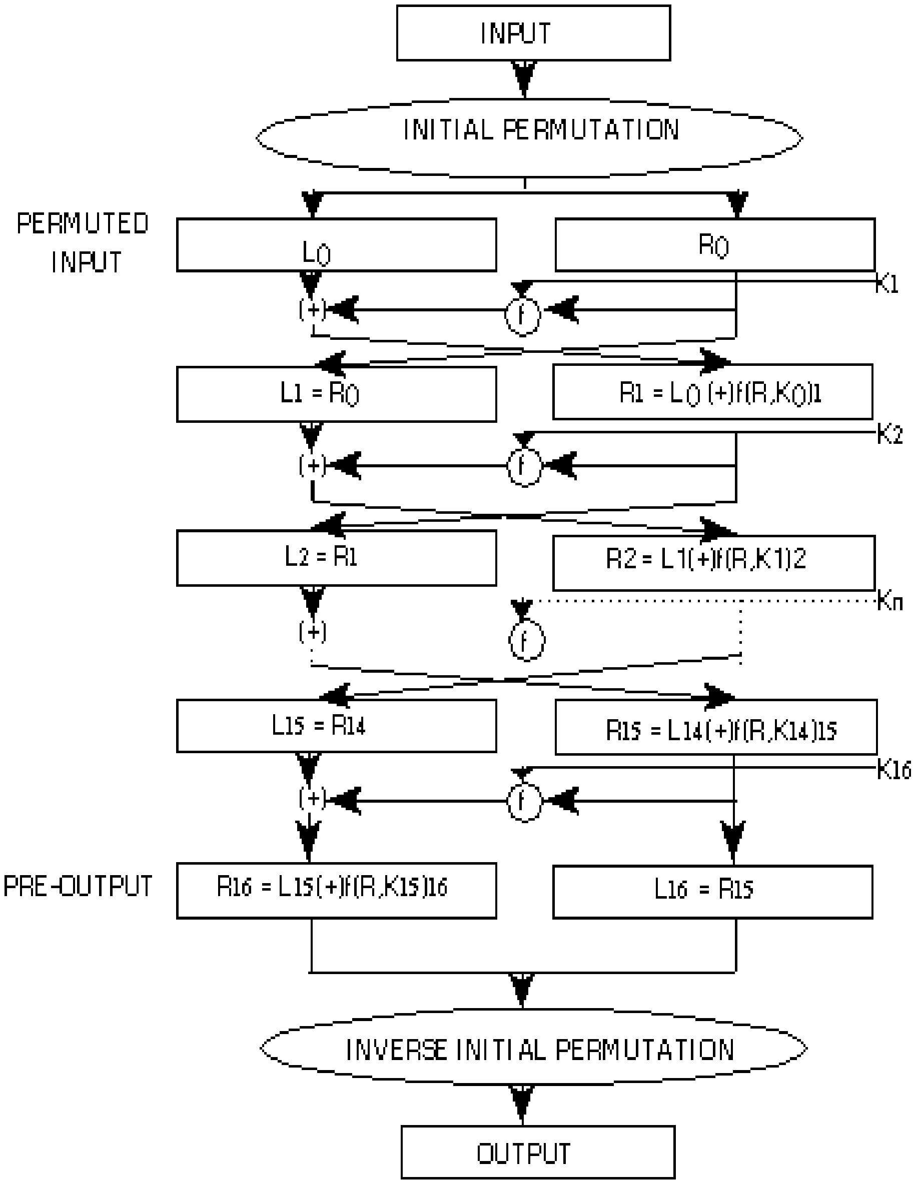

Fig. 3-1 Traditional Enciphering Computation

3.1.2. 3-DES Architecture

The secret key DES only has 64 bits (including 8 bits for error detection), people believe that DES may easily be cracked by modern computer because of the short secret key. However 3-DES is one of the methods to improve the weakness of the short secret key problem. By 3-DES, we can improve the secret key from 64 bits to

192 bits [20]. Anyway we don't need to worry about the secret problem in 3-DES operation, but the computation time is 3-DES is such a big problem for user.

3-DES is based on DES architecture. In order to create a longer secret key, 3-DES tries to do DES encryption and decryption three times to get triple length as the secret key in DES [21]. As Fig. 3-2, we can see that 3-DES needs triple more CPU time to finish a job.

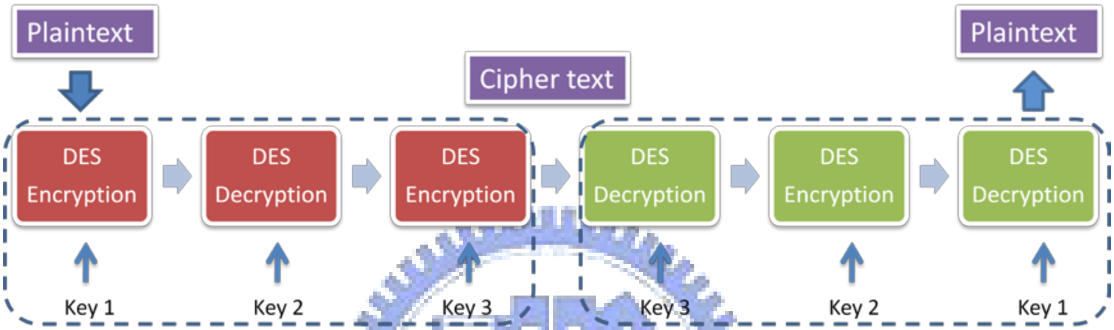

Fig. 3-2 DES Architecture

In detail, 3-DES's encryption can divide to three parts. First, we use the first key to do DES encryption to plaintext. Then we use the second key to do DES decryption and finally do the DES encryption with the third key to accomplish the 3-DES encryption. Therefore the 3-DES decryption is inverse operation to 3-DES encryption.

It is obviously that 3-DES is a computation-sensitive work. It may cost too much CPU time and CPU recourse [22]. What we need to do is using CUDA programming to speed up the 3-DES computation in GPU.

3.2. Implement 3-DES in CUDA

3.2.1. Facing problems in CUDA program

A trivial idea to implement 3-DES system in CUDA programming is put the encryption and decryption operation on graphic card with keep the tradition 3-DES architecture. However it may face several problems in CUDA programming.

- Parallel algorithm is not easy to build

When we try to rewrite the DES architecture for CUDA programming, one of the main problems in CUDA programming is how to parallelize 3-DES architecture. Since DES architecture is a sequential execution program as in Fig. 3-2, it is so difficult to parallelize the internal flow of DES. In fact, AES uses massive matrixes to solve this problem. The restriction of DES is not easy to crack because there is no good matrix solution to DES.

To solve this problem, we modify the traditional DES algorithm to fit CUDA program. Traditional DES deals with 64-bits plaintext once. In our idea, we input about 64 * N bits plaintext once [23]. N means how many threads we use on graphic card. By inputting 64 *N bits plaintext, we can easily create a new DES architecture with parallel algorithm. Nevertheless, there are still many problems when you try to input 64 * N bits plaintext once, we will talk about this later.

- How to optimize your system using GPU memory

Another troublesome problem is how to perfect assign the GPU memory. While we try to input 64 * N bits plaintext once, we find out that the system efficiency goes down quickly. The main reason is that we did not well assign the GPU memory [24].

In our original idea, we try not to rewrite the whole 3-DES architecture and want to get benefits by CUDA programming. In fact, this is a terrible mistake that we waste too much time in data moving from host memory (CPU memory) to device memory (GPU memory). Although copying data form host memory to device memory can use Direct Memory Access (DMA) to optimize the device's high-performance, we still have to reduce the frequency of data moving [25].

Traditional 3-DES encryption contains three components, including two DES encryptions and one DES decryption [26]. While we rewrite the 3-DES encryption and decryption form C code to CUDA code, it is necessary to reduce the frequency of data moving. For example that we first write two function call named __global__DES_encryption() and __global__DES_decryption(), "__global__" means this function call will be executed by GPU core. In Fig. 3-3, it is obviously that it needs too much data moving to complete 3-DES encryption once [27]. However what we need is to combine three operations into one operation, so we can reduce the six data moving operations to two data moving operation.

Algorithm 3-DES Encryption ((plaintext, ciphertext) ;
Input: *plaintext* (64 * N bits plaintext).
Output: *ciphertext* (64 * N bits ciphertext).

// N means how many threads we use in device.

begin
    declare integer *plaintext[64 * N], temptext[64 * N], ciphertext[64 * N];*
    declare integer *key1[64], key2[64], key3[64] ;*

    input *(*plaintext, temptext*);   //put plaintextin to temp buffer*
    key_schedule (key1, key2, key3);   // generate subkeys

    copy *temptext* form host memory to device memory
    __global__DES_encryption (temptext, key1);
    copy *temptext* form device memory to host memory

```
        copy temptext form host memory to device memory
        __global__DES_decryption (temptext, key2);
        copy temptext form device memory to host memory

        copy temptext form host memory to device memory
        __global__DES_encryption (temptext, key3);
        copy temptext form device memory to host memory

        swap (ciphertext , temptext);
        output (ciphertext);
end
```

Fig. 3-3 Traditional 3-DES algorithm with too much data moving

3.2.2. 3-DES Architecture for CUDA programming

A new 3-DES architecture is a special design for CUDA programming. In our design, we try many kinds of method to optimise the architercture. The final answer we get is the following flow chart as Fig. 3-4. The implement details of the flow chart will be discuss in next chapter, we just need to know how this architecture work and compare this new 3-DES architecture with traditional 3-DES architecture. In fact, Fig. 3-4 only show how 3-DES encryption work. The 3-DES decryption's flow chart is almost like 3-DES encryption flow chart but still exist some differents. We will also talk about the differents on next chapter.

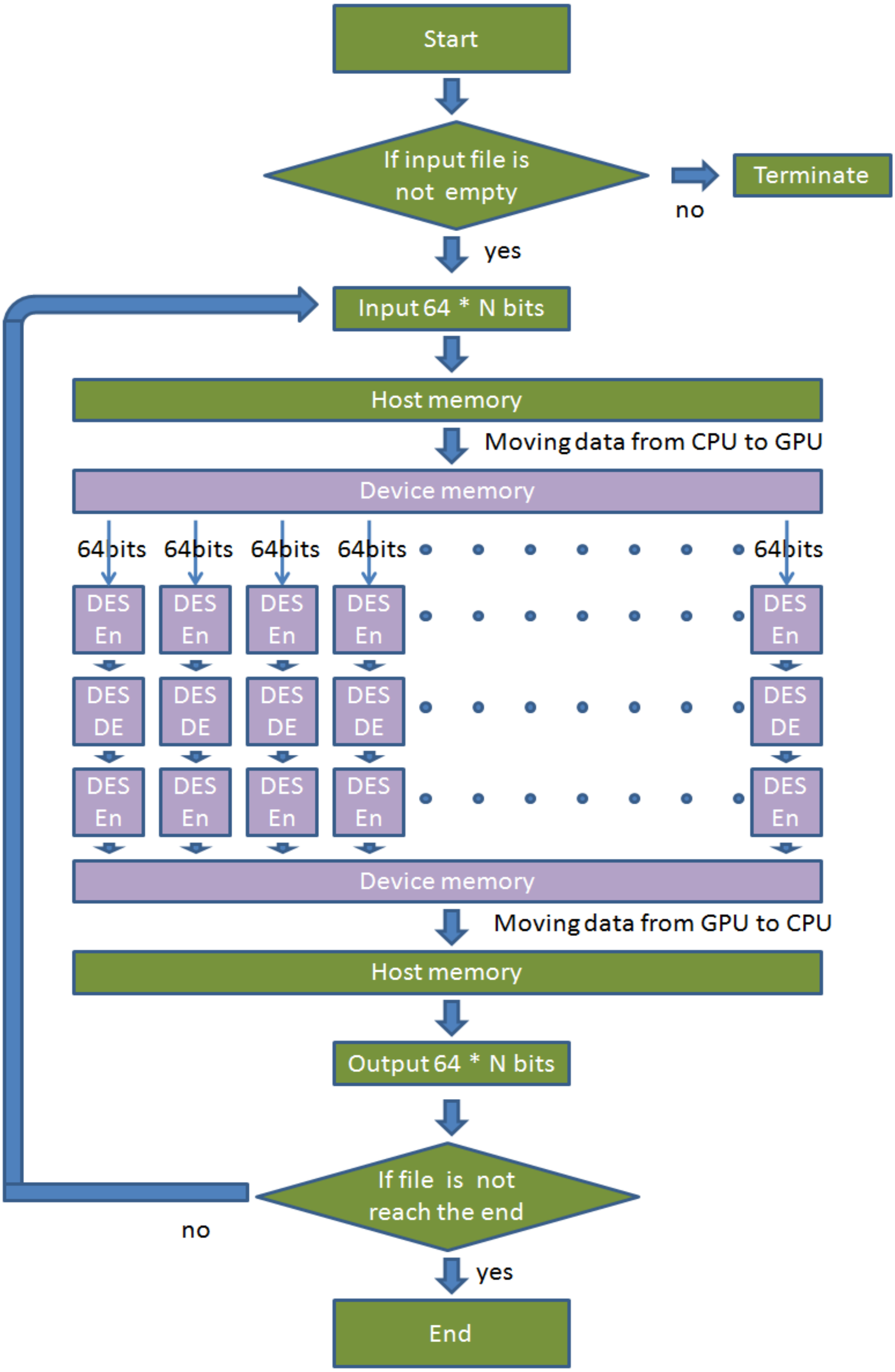

Fig. 3-4 New 3-DESencryption Flowchart

3.3. Integrate 3-DES with web server

To analysis the efficiency of 3-DES encryption and decryption, we construct a website that offers 3-DES encryption and decryption service. The website is used for ensuring internal security. All data stored in the website is encrypted. While users want to download the data form website, the website will immediately decode the encryption data and send the result to users. On the other hand, when a user wants to upload a file to web server, the website will encode the file first and then put the cipher text into web server.

In Fig. 3-5, it is conspicuous to realize how this data server works. After the user upload a file to server, the website will refresh the download page and list a new cyber link to the new upload file. That is because we want to test the 3-DES speed up performance through the environment of integrating 3-DES with web service. In traditional 3-DES encryption and decryption, the efficiency is too low to tolerate. After we implement 3-DES in CUDA programming, the 3-DES encryption and decryption become fast enough for general purpose. Moreover, users would not know that web server do the 3-DES decryption when they push the download button.

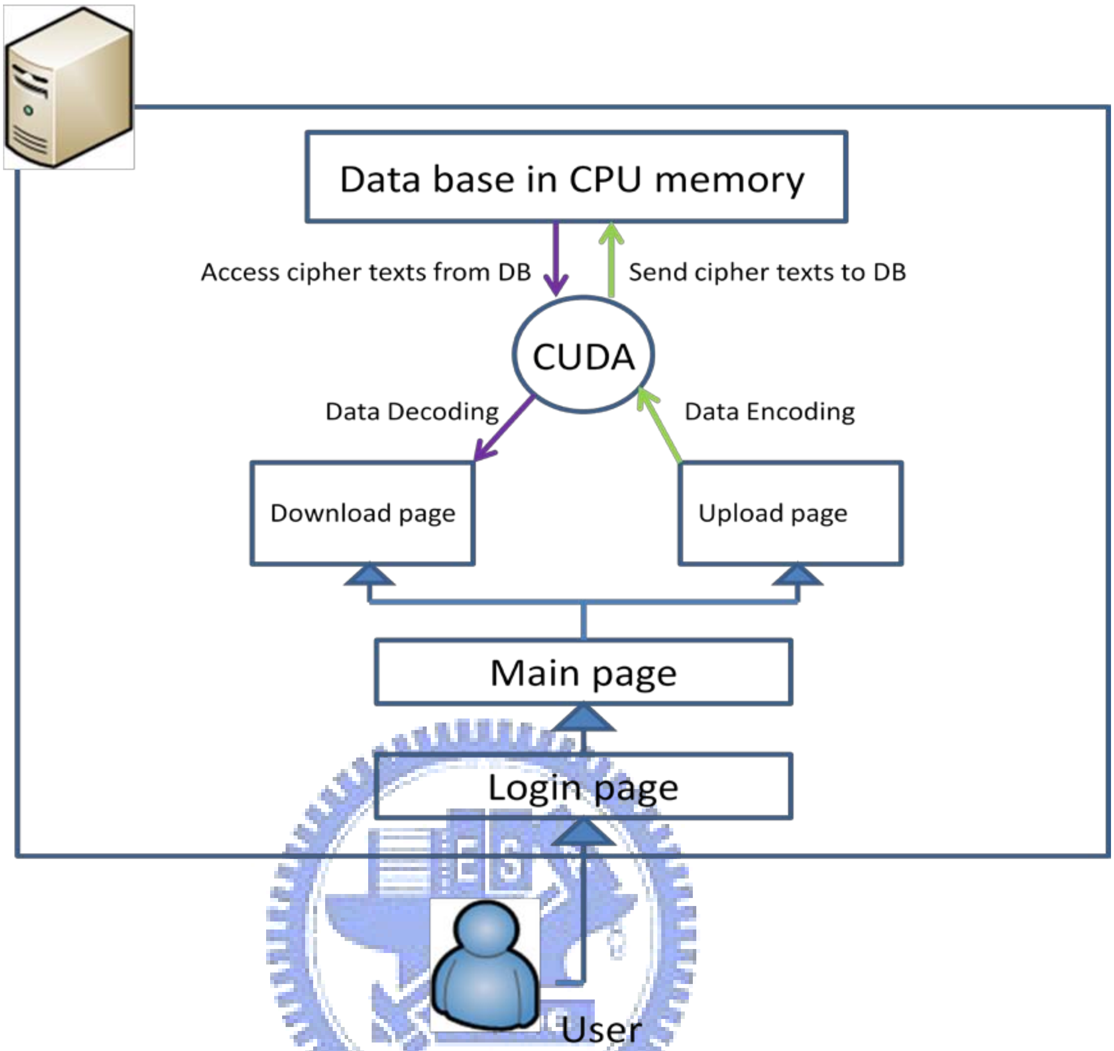

Fig. 3-5 New Web service architecture

Unfortunately we run into many troubles in implement the whole environment. The most important trouble is the security problem. While a user tries to download a file from this server, the CPU in the web server would send a signal to call DMA for moving data form host memory to device memory ideally in our theory. But it is not working actually. Final we find out that there is some security measures for avoiding any unsuitable access to host computer. In other hand, the host computer would not accept the GPU access request that is sent from apache server. We will discuss how we solve this problem in next chapter.

Chapter 4
Experimental Results and Analysis

In this chapter we will discuss our implement methods in detail. Based on the framework we talked in chapter three, we would introduce that how we construct each of these components. Therefore, since that there are too much components in our architecture, we just introduce the important or particular design in our implementation and some exceptional troubles programmers may face.

4.1. 3-DES implementation in GPU

4.1.1. Move data from CPU to GPU

The First step of DES encryption and decryption is moving plaintexts or cipher texts from host to device. Host means CPU and device means graphic card. First, the host will instruct the device to malloc same size of data in GPU memory. Second, the host would send data from host memory to device memory with DMA. Finally GPU receive the data, the host can further call GPU to do computing on shaders.

While GPU receive the data and the instruction from CPU, we need to map the data to threads in shaders. In DES implementation, a plaintext is 64-bits. We use a unsigned char array with 8 elements to store the 64-bits and we store all the 64 * N bits in the unsigned char array named block. First GPU reads all elements in array block and send each element to different thread. We use J = (blockIdx.x*blockDim.x+ threadIdx.x) to indicate each thread. J means absolutely thread ID, blockIdx means how many blocks in a grid, blockDim mean how many threads in a block, and threadIdx means the relative thread ID in a block. Second we divide the plaintext to two parts named left and right. Each of Left and right are contains 32 bits, so we can combine four unsigned char elements individually to left and right like Fig. 4-1.

Algorithm __golbal__DES_EN (block, key) ;   // Same as 3-DES_DE
<u>Input:</u> *block* (64 * N bits plaintext)
 *Key(64 * 3 keys)*
<u>Output:</u> *block* (64 * N bits cipher)
// N means how many threads we use in device.
begin

```
    unsigned long left,right;
    int j=blockIdx.x*blockDim.x+threadIdx.x;

    left = ((unsigned long) block[0 + 8 * j] << 24)
      | ((unsigned long) block[1    + 8 * j] << 16)
      | ((unsigned long) block[2    + 8 * j] << 8)
      | (unsigned long) block[3    + 8 * j];
    right = ((unsigned long) block[4 + 8 * j ] << 24)
      | ((unsigned long) block[5    + 8 * j] << 16)
      | ((unsigned long) block[6    + 8 * j] << 8)
      | (unsigned long) block[7    + 8 * j];

    Permutation
    Do 64 times of round function
    Inverse permutation

    block[0 + 8 * j] = right >> 24;
    block[1 + 8 * j ] = right >> 16;
    block[2 + 8 * j ] = right >> 8;
    block[3 + 8 * j ] = right;
    block[4 + 8 * j ] = left >> 24;
    block[5    + 8 * j] = left >> 16;
    block[6 + 8 * j ] = left >> 8;
    block[7    + 8 * j] = left;
```

end

Fig. 4-1 3-DES Pseudo code with moving data details

4.1.2. Initial permutation

The permutations employed by the cipher are described using bit numbers. The numbering used in the standards documents is enumerating the bits from left to right, starting at 1. When displayed as a matrix, row major order is used. This is best illustrated by the identity transform shown in Table 4.1.

1	2	3	4	5	6	7	8
9	10	11	12	13	14	15	16
17	18	19	20	21	22	23	24
25	26	27	28	29	30	31	32
33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48
49	50	51	52	53	54	55	56
57	58	59	60	61	62	63	64

Table 4-1 Bit number

The Pentium processor reads its memory using the opposite bytes (row) order, giving the bit number matrix shown in Table 4.2. We have here divided the matrix in upper and lower halves. On the Pentium we need one 32-bit register to store each half, and hence swapping the halves amounts to swapping the roles of those two registers.

57	58	59	60	61	62	63	64
49	50	51	52	53	54	55	56
41	42	43	44	45	46	47	48
33	34	35	36	37	38	39	40
25	26	27	28	29	30	31	32
17	18	19	20	21	22	23	24
9	10	11	12	13	14	15	16
1	2	3	4	5	6	7	8

Table 4-2 Input / output bit ordering on Pentium

The initial permutation of the DES has a very simple structure, and can be performed as a series of bit block swaps known as Hoey's Initial Permutation Algorithm. This algorithm is shown in Table 4.1. Note that there is also an implicit swap of the upper and lower halves at the beginning. To optimize the algorithm for Pentium, Richard Outerbridge's C code implementing the algorithm was analyzed, providing a set of bit exchanges between the two halves of the input block. Though not the same implementation, the idea for how to code each swap came from Eric Young's libdes. IP.1 is applied by performing the swaps of IP in reverse order.

58	50	42	34	26	18	10	2
60	52	44	36	28	20	12	4
62	54	46	38	30	22	14	6
64	56	48	40	32	24	16	8
57	49	41	33	25	17	9	1
59	51	43	35	27	19	11	3
61	53	45	37	29	21	13	5
63	55	47	39	31	23	15	7

Table 4-3 Initial permutation

In our CUDA programming, the initial permutation will be computed on GPU device.

4.1.3. Round function

DES needs 16 times of round function to complete encryption and decryption. Generally we make a DES function and recursive call this function to complete 3-DES encryption and decryption. In fact, this may waste lots of CPU time to move data between CPU memory and GPU memory. The solution to reduce the unnecessary memory access is to do 64 times of round function continuously. Thus program only needs two memory accesses to finish one 3-DES encryption or decryption.

The main important things to rewrite the Round function are to care about the on chip memory access. We use a constant unsigned long memory to store our spbox. Constant memory's speed is usually faster than global memory and would not occupy the space of shared memory. After assign the accuracy values to the spbox, we do the round function for 64 times just like the following Fig. 4-2.

```
While (I < 64)
    {
    //phase i
        work = ((right >> 4) | (right << 28)) ^ ks[i];
        left ^= Spbox[6][work & 0x3f];
        left ^= Spbox[4][(work >> 8) & 0x3f];
        left ^= Spbox[2][(work >> 16) & 0x3f];
        left ^= Spbox[0][(work >> 24) & 0x3f];
        work = right ^ ks[i+1];
        left ^= Spbox[7][work & 0x3f];
        left ^= Spbox[5][(work >> 8) & 0x3f];
        left ^= Spbox[3][(work >> 16) & 0x3f];
        left ^= Spbox[1][(work >> 24) & 0x3f];
    }
```

Fig. 4-2 Round function details

4.2. 3-DES Encode implementation

Since we want to design a 3-DES encryption program, we need to consider all kinds of file type. For example in the word files like PowerPoint or WordPad, it uses '\0' to be its end symbol. In this situation that we just need to determine what the input word is. But in streaming data like music or movie files, the '\0' is not the end symbol. This means we need to adjust our program to fit all the file types.

Moreover, we input 64 * n bits plaintext once rather than 64 bits plaintext. This may make a mistake for us to determine where the end symbol is. For instance, if we input 1000 words of plaintext once and the end symbol '\0' is the 1250th word of the plaintext, we must need to do the input loop for two times to input all the plaintext. However after we do the loop for two times, we read the 2000th word of the plaintext. Because the 2000th word is a empty word that compiler could not recognize, the loop would not be stopped.

For the two reasons we describe above, we bring up follow pseudo codes to explain how we solve these problems. First is the pseudo code of 3-DES encryption in Fig. 4-3.

```
Algorithm 3-DES Encryption ((plaintext, ciphertext) ;
Input: plaintext (64 * N bits plaintext).
Output: ciphertext (64 * N bits ciphertext).

// N means how many threads we use in device.

begin
    declare integer plaintext[64 * N], temptext[64 * N], ciphertext[64 * N];
    declare integer key1[64], key2[64], key3[64], key[192] ;
    declare integer plaintextsize = 64 * N;

    While(not reach the end of file)
    {
        input_EN (plaintext, temptext); //put plaintext tinto temp buffer
        key = key_schedule (key1, key2, key3);  // generate subkeys

        copy temptext form host memory to device memory
        __global__3DES_encryption (temptext, key);
        copy temptext form device memory to host memory

        swap (ciphertext , temptext);
        output_EN (ciphertext);
    }
end
```

Fig. 4-3 3-DES Encryption Pseudo code

In the figure above, it is obviously that only the function named __global__3DES_encryption () would be executed in GPU. This is because of parallel factor. Besides the __global__3DES_encryption function, only the key_schedule function can be parallelized. But the key_schedule function has few computation that it is not worth executing in CUDA, its memory access time is larger than its computation time.

Another interesting thing deserves to mention is the input function. Since we want to design a mature code to support all the file types, we need to devise new end symbol judgment methods. Our method is to design a new end symbol for all file types. If input 1000 words which equals to 8000 bits of plaintext once, the design details are follows:

I. Design a new end symbol

End symbol is a unique word that only recognized by program, so we cannot choose the common word to be the end symbol. We choose the word "/nend/ab" for our end symbol. This can distinguish the end symbol from other words.

II. Add end symbol to plaintext every 1000 words

The program will check if the plaintext reaches the end every 1000 words. We use fread() to determine if we had already read in all the plaintext. If fread() return the value of 8000, which means input 8000 bits or 1000 words. This means the file may not reach the end. We should add the end symbol to this point and go on search for the end of the plaintext. So we accurately output 1008 words (1000 words + 8 words end symbol) at one time.

III. Find out the end of the plaintext

While the value returned form fread() is less than 8000, this means we have found the end of the plaintext. Since we process 1000 words once, we need to fill it to reach 1000 words. The end symbol "/nend/ab" is only 8 words. After the end symbol, we fill the vacancy with "\0" to reach 1008 words.

IV. The stop condition

After we find out the end of the plaintext, next round the fread() would return the value of 0. 0 means nothing left in the plaintext, so the program will break the loop to stop 3-DES encryption.

```
Algorithm 3-DES input_EN (plaintext, *temptext) ;
Input: plaintext (64 * N bits ciphertext).
Output: ciphertext (64 * N bits plaintext).

// N means how many threads we use in device.
begin
    int trueinputsize = inputsize -8;
    int len = fread(cp , sizeof(unsigned char), trueinputsize, fileinput);

    if(len equals to the size of input file)
    { // add end symbol after the end
        temptext[len] = '/';                temptext [len+1] = 'n';
        temptext [len+2] = 'e';             temptext [len+3] = 'n';
        temptext [len+4] = 'd';             temptext [len+5] = '/';
        temptext [len+6] = 'a';             temptext [len+7] = 'b';
    }
    if( len is less than the size of input file and len doesn't equals to 0)
    { // add end symbol after the end and fill 1008 words with '\0'
        temptext [len] = '/';               temptext [len+1] = 'n';
        temptext [len+2] = 'e';             temptext [len+3] = 'n';
        temptext [len+4] = 'd';             temptext [len+5] = '/';
        temptext [len+6] = 'a';             temptext [len+7] = 'b';
        for (i = (len+8); i < inputsize ; i++)
            temptext [i] = '\0';
    }
    if(file reach the end or len is 0)
        break;
end
```

Fig. 4-4 Input Details of 3-DES Encryption Pseudo Code

4.3. 3-DES Decode implementation

3-DES decode implementation is similar to 3-DES encode implementation. But 3-DES decryption needs more decision operation, which means that needs more CPU time to determine when to stop. Unlike 3-DES encode implementation, the input of 3-DES decryption is all cipher texts. We need to decode the cipher texts before we analysis it.

```
Algorithm 3-DES Decryption (plaintext, *temptext) ;
Input: ciphertext (64 * N bits ciphertext).
Output: plaintext (64 * N bits plaintext).

// N means how many threads we use in device.

begin
    declare integer plaintext[64 * N], temptext[64 * N], ciphertext[64 * N];
    declare integer key1[64], key2[64], key3[64], key[192] ;
    declare integer ciphertextsize = 64 * N;

    While(not reach the end of file)
    {
        Input_DE (ciphertext, temptext, ciphertextsize); //put ciphertext
        into temp buffer
        key = key_schedule (key3, key2, key1);  // generate subkeys

        copy temptext form host memory to device memory
        __global__3DES_encryption (temptext, key);
        copy temptext form device memory to host memory

        swap (plaintext , temptext);
        output_DE (plaintext);
    }
end
```

Fig. 4-5 3-DES Decryption Pseudo code

The main implementation difference between encryption and decryption is the decision condition. In 3-DES encryption program, system can use feof() to see if the file reaches the end. But in 3-DES decryption, all inputs are cipher texts. If we just decode and output, we would get the plaintext with many end symbols inside. The following figure is our implement details.

I. Input 1008 words and decode

While this program starts, it would input 1008 words form cipher text. After inputting 1008 words, the program will send the cipher text to GPU for decoding. This phase will continue until there are no words of cipher text to input.

II. Analysis after decoding

After decoding, the program would analysis the last eight words of the plaintext. If the eight words is "/nend/ab", it means the input file does not reach the end. The program will output the result without "/nend/ab" and return to stage one. Else if the eight words is not "/nend/ab", it means the input file reaches the end. The program should trace the 1008 words to find out where the end symbol is. When the program finds out the position of the end symbol, it would send the result without "/nend/ab\0\0\0……" to plaintext. After that, the program will break from the loop and finish all the jobs.

```
Algorithm 3-DES output_DE ((plaintext, *temptext) ;
Input: ciphertext (64 * N bits ciphertext).
Output: plainrtext (64 * N bits plaintext).

// N means how many threads we use in device.

begin
    int countertemp = 0;

    if (there is nothing inputs form file)
        return; // reach the file end
    if( (temptext[len-8] = '/' ) && ( temptext [len-7] = 'n' ) &&
      (temptext [len-6] = 'e') && ( temptext [len-5] = 'n ) &&;
      (temptext [len-4] = 'd') && ( temptext [len-3] = '/' ) &&
      (temptext [len-2] = 'a') && ( temptext [len-1] = 'b')
    }
        Countertemp= inputsize - 8;   // set the size for output
    esle
    {
        unsigned char *temp = temptext;
        for( int i = 0; i < inputsize; i ++ )
        {
            if( (temp [i] = '/') && (temp [i+1] = 'n') &&
              (temp [i+2] = 'e')&& (temp [i+3] = 'n') &&
              (temp [i+4] = 'd')&& (temp[i+5] = '/') &&
              (temp [i+6] = 'a')&& (temp[[i+7] = 'b') )
                Break;
            else
                countertemp ++; // set the size for output
        }
    }
    fwrite(temptext, size of (unsigned char), countertemp, Decodeoutput);
    // Decodeoutput is a pointer to the output file.
end
```

Fig. 4-6 Output function of 3-DES Decryption Pseudo code

4.4. Configuration

4.4.1. Hardware configuration

To support CUDA computing, we have the following hardware configuration:

CPU	Intel® Core™2 Quad Processor Q6600 (2.4GHz, quad-core)
Motherboard	ASUS P5E-VM-DO-BP, Intel® X38 Chipset
RAM	Transcend 2G DDR-800
GPU	NVIDIA 9800GT 521MB (ASUS OEM)
HDD	WD 320G w/ 8MB buffer

Table 4-4 Hardware Configuration

Since we want to compare the performance of CPU versus GPU, we list the specification of the GPU in detail as follows:

Model	EN9800GT TOP/HTDP/512M
Graphics Engine	GeForce 9800GT
Bus Standard	PCI Express 2.0
Video Memory	512MB DDR3
Engine Clock	650 MHz
Shader Clock	1.625 GHz
Memory Clock	1.94 GHz (970MHz DDR3)
Memory Interface	256-bit
DVI Max. Resolution	2560 * 1600
D-Sub Output	Yes x 1 (via DVI to D-Sub adaptor x 1)
DVI Output	Yes x 2 (DVI-I)
HDTV Output(YPbPr)	YES, via HDTV Out cable
HDCP compliant	Yes
Adaptor/Cable Bundled	1 x DVI-to-D-Sub adapter 1 x HDTV-out cable 1 x Power Cable
Software Bundled	ASUS Utilities & Driver
Note	The card size is 4.376 inches x 9 inches

Table 4-5 NVIDIA 9800GT Hardware Specification

4.4.2. Software Configuration

OS	OPEN SUSE 11.1 (32bit version)
GPU Driver Version	97.73
CUDA Version	2.0 beta

Table 4-6 Software Configuration

4.5. Evaluation and Analysis

The following is our statistics of experimentation. We got the data by testing the CPU time in OpenSSL and implementing DES and 3-DES on graphic card. Besides DES and 3-DES cryptography, AES cryptography had been implemented on CUDA in 2007 by Svetlin A. Manavski [28]. We will discuss how many differences between our results.

cipher \ compute method		**OpenSSL on CPU**	**Implement DES on CUDA in our work**
DES (64bits)	4M	60ms	32ms
	8M	108ms	41ms
	109.8M	2s 100ms	712ms
	697M	14s 313ms	4s808ms

Table 4-7 Comparison DES in CPU and in GPU

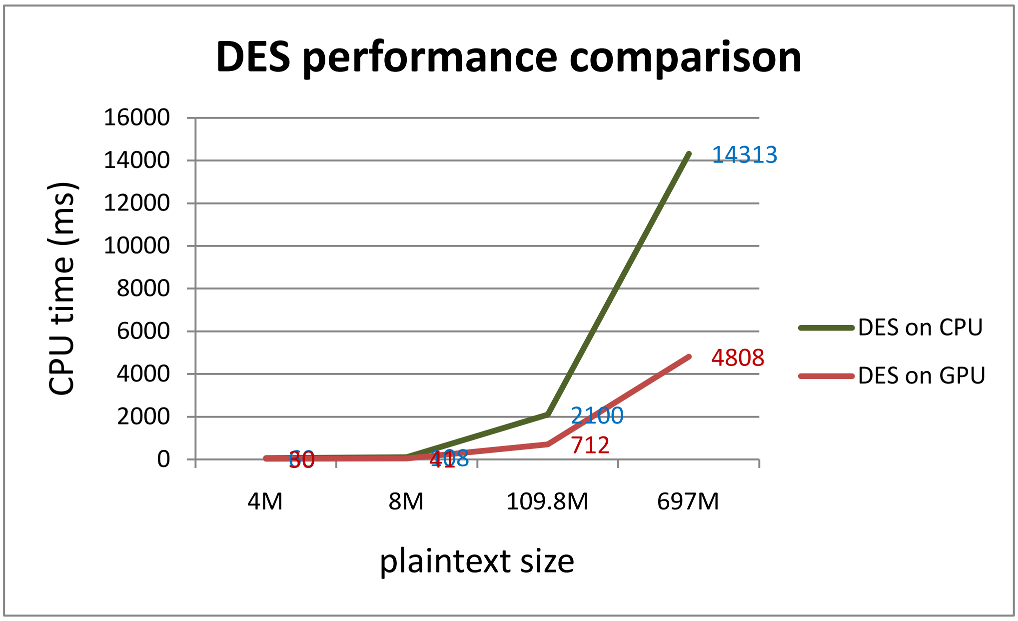

Fig. 4-7 DES performance comparison

There are two parts in table 4-7. One is about the CPU time to implement DES in CPU and the other one is about the CPU time to implement DES in CUDA programming. We test the DES's original CPU time in OpenSSL. It is obviously that we get better performance in CUDA programming. In plaintext of 4M byte, DES in CUDA programming has about 1/2 CPU time to DES of OpenSSL in CPU. In plaintext of 697M, we even get about three to four times of performance in CUDA programming. This means we would get more benefits in larger plaintext size to fit parallel algorithm.

cipher \ compute method		3-DES on CPU	Implement 3-DES on CUDA in our work
3-DES (192bits)	4M	220ms	44ms
	8M	244ms	56ms
	109.8M	6s 936ms	1s 160ms
	697M	47s 15ms	8s 225ms

Table 4-8 Comparison 3-DES in CPU and in GPU

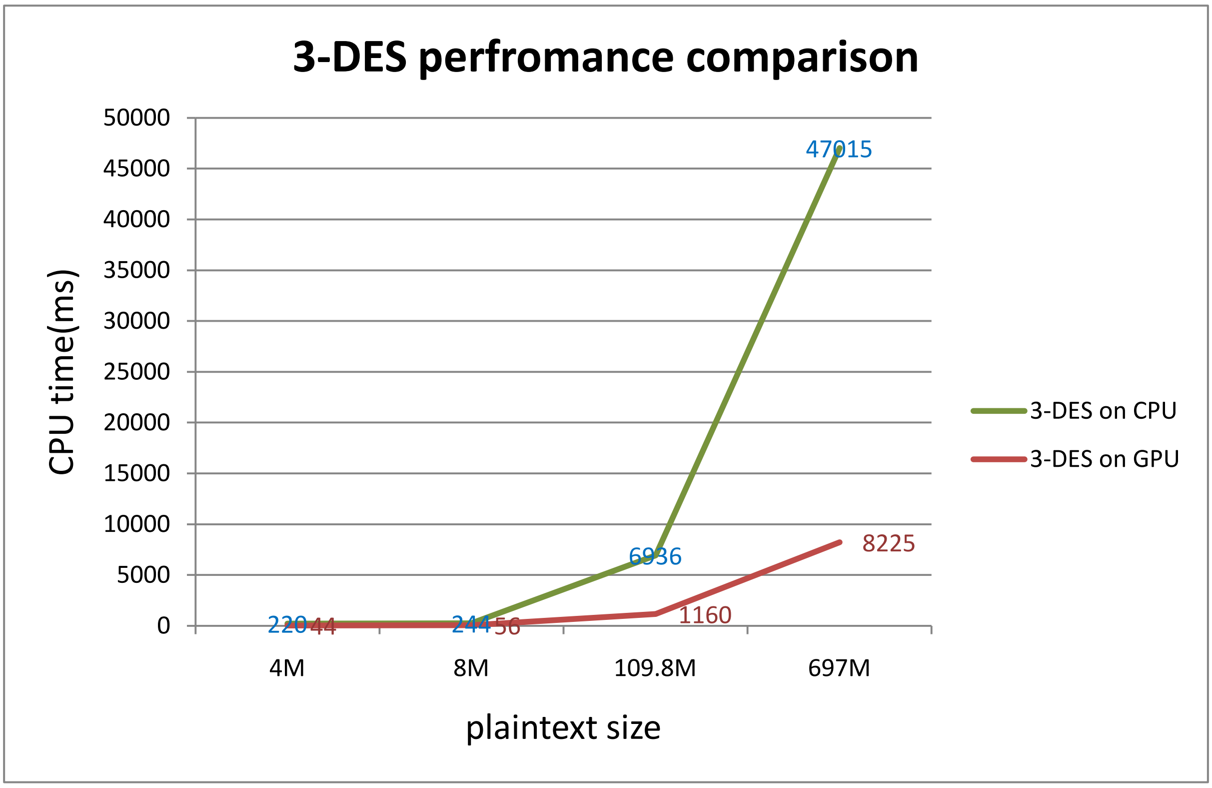

Fig. 4-8 3-DES performance comparison

In 3-DES, we can see that it would get about five times performance in GPU than in CPU while the file size is near to 4M bytes. Moreover, it would get more than six times performance if the file size is larger than 700M bytes. The new 3-DES on

GPU is faster than AES on CPU in all the situations. This means that we can do the same encryption and decryption in 3-DES to get the similar performance. The following is relationship between AES on CPU and 3-DES on GPU.

In the related work about speeding up AES with CUDA programming by Svetlin A. Manavski, a comparison has been made between GPUs and a CPU implementation based on the OpenSSL [29] library. Both the internal GPU elaboration time is shown and the total time, that includes the time needed for the download and read-back operations (that means copying data from the host memory to the GPU device and back). The CUDA-AES [30] implementation on NVIDIA card is faster than the CPU on every input size with reference both to the internal GPU and to the total time [31]. A peak throughput rate of 8.28 Gbit/s is achieved with an input size of 8MB. In that case the GPU is 19.60 times faster than the CPU [32]. So we have both have about six times in GPU than in CPU when the file size is more than 512M bytes.

Chapter 5 Conclusion and Future works

5.1. Conclusion

Parallel programming nowadays becomes more and more important to programmer. This is because multi-cores is the trend to both CPU and GPU architecture development. However not all the programs are parallel program and this paper provide a method to solve this program. Find out the main points to parallelize a program is the most important thing to programmer.

CUDA code is not easy to optimize if the user do not understand the architecture of CUDA. Sometimes this makes programmer to abuse the graphic memory. Then the new CUDA program may have worse performance than original program on CPU. So the final performance is decided by programmer's effort.

Although DES and 3-DES is an old cipher, there still many people use them to do encoding and decoding. This is because it needs a long time to verify the security of a cipher. DES and 3-DES however both have the trusty security. 3-DES is a compute-sensitive program, so it usually is used for off-time computing. For example, many finance institutions and banks use 3-DES to protect their system while making an inventory. Each cipher has its advantages and disadvantages, what we have to do is to properly use the advantages and keep of the disadvantages. It is better to ameliorate a defective cipher rather than abandon it rashly.

In the result we can see that CUDA programming is a cheap way to speed up a compute-sensitive program. Rather than implement a system on chip or on FPGA for about more than one hundred thousand, use just need to pay about five to eight thousand to get CUDA. Maybe a program gets better performance on FPGA than on graphic card. But this is a cheap and easy method for a program to learn how to speed up a program or a system in parallel program.

5.2. Future work

(1) Speed up more cipher in CUDA programming

Except for DES and 3-DES, there are still many ciphers that we can try to speed them up in CUDA programming. But some cipher is not easy to implement on graphic card. For example the RSA cipher is very hard to get better performance in CUDA programming. RSA is an algorithm for public-key encryption. The encryption fun of RSA is $c \equiv m^e \pmod{n}$. C means cipher text, m and n are integers that 0 < m < n. It is obviously that is very hard to parallelize the encryption function of RSA. Although there is still some methods like squaring algorithm can improve this situation. However the performance is still too bad compared with programming in CPU and CUDA unless those programmers can design a new parallel algorithm to RSA. Entirely more high performance ciphers bring us more convenient life. It is necessary that programmers try to speed up more cipher in CUDA programming.

(2) Add the new 3-DES architecture to SSL

Secure Sockets Layer (SSL), are cryptographic protocols that provide security and data integrity for communications over networks such as the Internet. SSL encrypt the segments of network connections at the transport layer end-to-end. SSL contains three parts of cipher system, public-key cryptography, symmetric -secret system, and one-way hash function. 3-DES is one kind of public-key cryptography. If programmers furthermore implement the new 3-DES architecture to SSL, it may improve the all the transport layer performance.

(3) Improve the new 3-DES-website's security

Our new 3-DES-website only offers data encryption and decryption. It could be safer if adding more safety mechanisms. However website security design is not our professional specialty. After this drawback has been reformed, the new 3-DES-websitr can be used to ensure internal management of enterprises.

Bibliography

[1] E.Biham and A. Shamir, “ Differential Cryptanalysis of DES-like Cryptosystems.” Journal of Cryptology, Vol.4, pp.3-72, (1911).

[2] National Institute of Standards and Technology (NIST), “FIPS 197: Advanced Encryption Standard (AES)”, 2001.

[3] E.Biham and A.Shamir, “ Differential Cryptanalysis of FEAL and N-Hash, “ Advance in Cryptology – EUROCRYPT’91, Lecture Note in Computer Science, Vol.547, p.1-16, (1991).

[4] E.Biham and A.Shamir, “Differential Cryptanalysis of the full 16-round DES, “ CRYPTO’92 Extended Abstracts, p.12-1-12-5, (1992).

[5] General Purpose Computation Using Graphics Hardware, http://www.gpgpu.org.

[6] NVidia CUDA , http://developer.NVidia.com/object/CUDA.html.

[7] OpenGL Shading Language Specification, Version 1.20 http://www.opengl.org/registry/doc/GLSLangSpec.Full.1.20.8.pdf

[8] AMD CTM, http://ati.amd.com/companyinfo/researcher/documents.html.

[9] NVIDIA CUDA Programming Guide, Version 0.8.2 http://developer.download.nvidia.com/compute/cuda/0_81/NVIDIA_CUDA_Programming_Guide_0.8.2.pdf

[10] I. Buck, A. Lefohn, P. McCormick, J. Owens, T. Purcell, R. Strodka, "General Purpose Computation on Graphics Hardware". IEEE Visualization 05, Minneapolis, USA, 2005.

[11] OpenGL Architecture Review Board, M. Woo, J. Neider, T. Davis, D. Shreiner, "The OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 2", 5th edition. ISBN 0321335732, Addison-Wesley, New York, 2005.

[12] D. L. Cook, A. D. Keroymytis, "Cryptographics: Exploiting Graphics Cards for Security", Advancements in Information Security series, Springer, 2006.

[13] Svetlin A. Manavski, CUDA Compatible GPU as an Efficient Hardware Accelerator for AES Cryprography, Springer-Verlag, New York, 2007.

[14] A.Tardy-Corfdir and H.Gilbert, "A Known Plaintext Attack of FEAL-4 and FEAL-6, " Advances in Cryptology – CRYPTO'91, Lecture Notes in Computer Science, Vol.576, (1991).

[15] M. Matsui and A. Yamagishi, "A New Method for Known Plaintext Attack of FEAL Cipher, " Advances in Cryptology – EUROCRY'92, Lecture Notes in Computer Science, Vol.658 , (1992).

[16] A. Shamir, " On the Security of DES, " Advances in Cryptology –CRYPTO'85, Lecture Notes in Computer Science, (1985).

[17] R.A.Rueppel, "Analysis and Design of stream Cipher, "Springer Verlag, (1986).

[18] Federal Information Processing Standards Publication (FIPS PUB) 46-2, Data Encryption Standard (DES), National Institute of Standards and Technology, Washington, DC, 1993.

[19] M. J. Wiener, "Efficient DES Key Search," *Technical* Federal Information Processing Standards Publication (FIPS PUB) 46-2, Data Encryption Standard (DES), National Institute of Standards and Technology, Washington, DC, 1993.

[20] M. J. Wiener, "Efficient DES Key Search," *Technical Report TR-244*, School of Computer Science, Carleton University, Ottawa, Canada, May 1994. Presented at the rump session of Crypto'93.

[21] F. Hendessi and M. R. Aref, "A Successful Attack Against the DES," *Information Theory and Application, Third Canadian Workshop Proceedings*, 1994, pp. 78-90.

[22] Frank Rubin, "Foiling an Exhaustive Key-Search Attack," *CRYPTOLOGIA* **11,** No. 2, 102-107 (April 1987).

[23] M. Hellman, "A Cryptanalytic Time-Memory Trade-off," *IEEE Trans. Info. Theory* **IT-26,** No. 4, 401-406 (1980).

[22] E. Biham and A. Shamir, *Differential Cryptanalysis of the Data Encryption Standard*, Springer-Verlag, New York, 1993.

[24] E. Biham and A. Shamir, *Differential Cryptanalysis of the Data Encryption Standard*, Springer-Verlag, New York, 1993.

[25] M. Matsui, “The First Experimental Cryptanalysis of the Data Encryption Standard,” *Lecture Notes in Computer Science* **839,** *Advances in Cryptology-Crypto* ***’94*** *Proceedings,* Springer-Verlag, New York, 1994.

[26] D. Coppersmith, “The Data Encryption Standard (DES) and Its Strength Against Attacks,” *IBM J. Res. Devebp.* **38,** NO. *3,* 243-250 (1994).

[27] B. ***S.*** Kaliski, Jr. and M. J. B. Robshaw, “Multiple Encryption: Weighing Security and Performance,” *Dr. Dobb’s Journal* **21,** No. 1, 123-127 (1996)

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

[29] OpenSSL Open Source Project, http://www.openssl.org.

[30] C. Su, T. Lin, C. Huang, C. Wu, “A High-Throughput Low-Cost AES processor”. IEEE Communications Magazine, vol. 41, no. 12, pp. 86-91, 2003.

[31] J. Wolkerstorfer, E. Oswald, M. Lamberger, “An ASIC Implementation of the AES Sboxes”. RSA Conference 02, San Jose CA, 2002.

[32] A. Hodjat, I. Verbauwhede, “Minimum Area Cost for a 30 to 70 Gbits/s AES Processor”. IEEE Computer Society Annual Symposium on VLSI (ISVLSI 2004), Eerging Trends in VLSI System Design, pp 83-88, 2004.