2013

# Use of general purpose graphical processing units in Blowfish encryption algorithm

Cankat Duman

# USE OF GENERAL PURPOSE GRAPHICAL PROCESSING UNITS IN BLOWFISH ENCRYPTION ALGORITHM

---

A Thesis

Presented to the

Faculty of

California State University,

San Bernardino

---

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

in

Computer Science

---

by

Cankat Duman

September 2013

# USE OF GENERAL PURPOSE GRAPHICAL PROCESSING UNITS IN BLOWFISH ENCRYPTION ALGORITHM

---

A Thesis

Presented to the

Faculty of

California State University,

San Bernardino

---

by

Cankat Duman

September 2013

Approved by:

[redacted]

Keith Evan Schubert, Advisor, School of Computer Science and Engineering

Aug 30, 2013
Date

[redacted]

Ernesto Gomez

[redacted]

Dick Botting

# ABSTRACT

The purpose of this thesis is to present the findings of the work that has been completed on exploring the possibilities of speedup gained when using General Purpose Graphical Processing Units for the Blowfish encryption algorithm with the use of the CUDA (Compute Unified Device Architecture) programming language and architecture. Current Graphical Processing Units (GPU) contains massive amounts of cores that can be used to work in massive parallelization. While task of massively parallel computation was a big challenge until recent years, the CUDA architecture and programming language is a very useful and relatively easier method to explore and experiment the possibilities of massive parallelization in GPU. In this thesis, both CPU and GPU version of the Blowfish algorithm has been implemented in C and CUDA programming languages respectively and compared by running number of tests in the same computing environment to fairly compare the results and to find out if predicted massive speed up can be achieved by the GPU with the use of CUDA. Using this enormous power of parallelization with the use of GPGPU has proved that a great deal of speedup can be gained specifically for the Blowfish encryption algorithm where it is costly in time and computation power when it is processed with the use of traditional CPU computations. Results of this thesis clearly show massive amounts of speed up in processing time when GPU is used over CPU with the use of CUDA and more importantly, results of this thesis help identify the speedup that can be accomplished to make this algorithm usable and relevant for the everyday use for everyone. Furthermore, with all of the data and results from experiments done for this thesis, further study in similar algorithms that utilize block ciphers for encryption, more

precisely, the Feistel networks could provide similar results and may be used in same fashion to allow people to take advantage of cryptography to secure their data in a fast and efficient way.

# ACKNOWLEDGEMENTS

I would like to thank my advisor Dr. Schubert, and the committee members Dr. Botting and Dr. Gomez for their help and guidance. I also would like to thank my wife Secil, who has given me her support and encouragement through out completing this work.

# DEDICATION

To my big brother Dogan, whom I always saw as a beacon of light and inspiration to do better.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

# 1. INTRODUCTION

This thesis will explore the performance speed up possibilities of Blowfish encryption algorithm with the utilization of Nvidia CUDA programming that take as advantage of massive amounts of cores that GPU has versus the standard CPU processing with original C version of the algorithm. General Purpose Graphical Processing Units (GPGPU) are high performance, many-core processors that are capable of handling very high level computation and throughput. Due to the massive parallelization capabilities of GPUs, when an algorithm designed for CPU processing gets re-tailored for the use of GPU computation, there is a possibility of achieving tremendous amount of speedup. Using GPU for parallelizing an encryption algorithm is especially new and there has not been an extensive studies in theses, particularly of the sub-field of GPGPU computation of Feistel ciphers. This thesis will explore the use of GPGPU in Feistel cypher encryption, specifically the Blowfish encryption algorithm. Utilizing the massive parallelization power of GPGPU to make the Blowfish encryption algorithm work much faster compared to CPU utilized process with the use of NVIDIA CUDA GPU Computing SDK 4.0 and NVIDIA CUDA tool kit. Furthermore, to figure out the speed up that will be achieved with the GPU computing, I will compare the clock cycles that it takes to compute a given set of problem with both CPU computed and GPU computed versions of the same block cipher algorithm. There are

number of different programming languages for utilizing the power of GPU such as CUDA, OpenCL, DirectCompute, and CUDA Fortran. CUDA is NVIDIA's parallel computing architecture and CUDA is a C/C++ like programming language to utilize the massive parallel computing power of NVIDIA GPU's.

Use of GPGPU in encryption, especially with block cipher algorithms, shows tremendous speed up on the process and improves the results. Furthermore, use of GPGPU with block ciphers opens up the possibility of using some of the inefficient encryption algorithms due to the computational power limits caused by sequential CPU computation, whereas GPGPU allows the algorithms to execute in massive scale parallelism.

### *1.1 Background*

Improvements in computing power of GPUs in recent years has championed GPUs compared to CPUs. Their massive computing power (Most of the recent GPUs have excess of Teraflop computing power) made them an easy choice to utilize them in general purpose computations. Traditionally, any type of computing except for the graphics processing has been done on CPUs. However, due to the hardware limitations that the CPU makers has encountered such as size, heat, and number of other issues forced them to look for alternative methods to do general purpose computation. GPGPUs have been one of the leading alternative methods that is being studied and taken further to replace current methods and architectures being use in the computing world. One of the biggest differences between GPU and CPU is the amounts of cores that they contain. The GPU contains very large number of smaller and less powerful

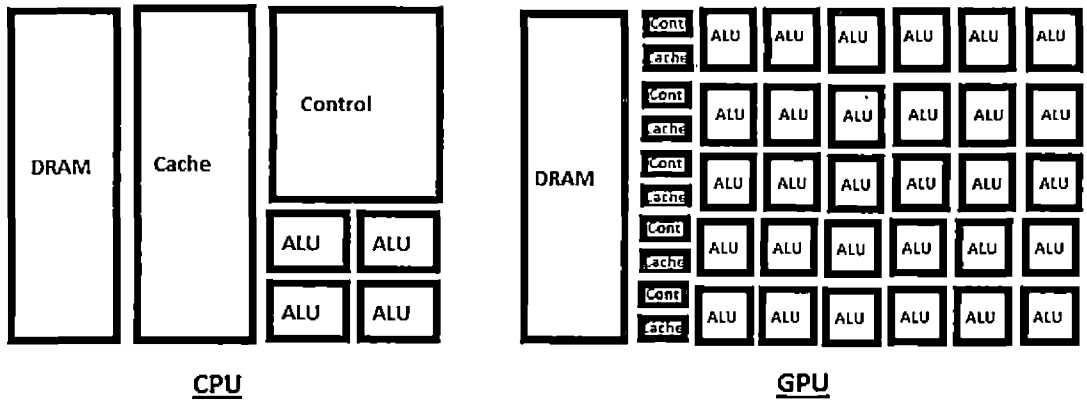


*Fig. 1.1:* Comparison Between Basic CPU and GPU Architecture

cores. Whereas, CPU contains fewer cores that are bigger and more powerful. This clearly separates the type of tasks that each should handle in an optimal way. CPU can handle small amounts of tasks that require very intense computational power. On the other hand, GPU can handle large amount of small tasks that is not as intense and therefore does not require as much computational power for each small task. In short, CPU can handle few computationally demanding tasks and GPU can handle many less demanding task in an optimal way. There has been numerous studies done in the GPGPU field and it is currently being used in real life production environments in number of different sub fields of computer science. However, as it was mentioned before, encryption in the field of GPGPU is fairly new and there is only hand full of published works on this particular area of research in GPGPU field. NVIDIA's CUDA architecture allows the utilization of massive parallel computation through a great number of blocks which contains a large number of threads that are exacted in parallel as it can be seen in the Figure 1.1 above which illustrates the CUDA architecture. With the utilization of CUDA capable NVIDIA graphical processing

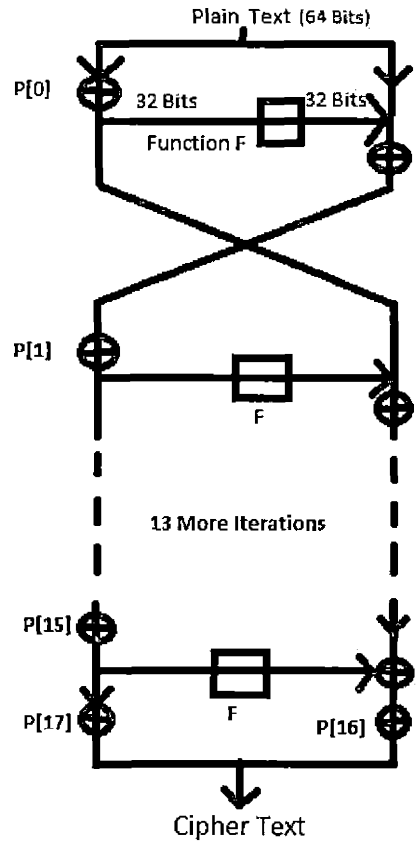


*Fig. 1.2:* Illustration of Feistel Network Used in Blowfish

unit, a block cipher encryption algorithm can be re-designed and coded in with the use of CUDA C/C++ to tap in to the massive parallel computation power of GPU. Due to the nature of block cipher algorithms, they can naturally be adapted in to CUDA architecture since use of blocks and threads within blocks can be easily applied to the algorithms like the Blowfish block cipher. Diagram below demonstrates how the Blowfish block cipher algorithm works.

There are numerous block cipher algorithms that are equally suitable for the task of demonstrating and comparing the processing speed and power of GPGPUs. However,

for this thesis, Blowfish algorithm will be the choice due to availability of resources. As Figure 1.2 demonstrates, handling the computation of each set of 64 bits of plain text with assigning them to a thread within a block would ideally produce a great deal of speed-up given the fact memory management is done properly to allow optimal results. Figure 1.2 illustrates the cycles of computation that each 64 bits of plain text goes through to produce the Cipher text. Since each iterative cycle produces a cipher text, and all iterations with in the computation of the Cipher text must be completed in a sequential order to get the correct results. It is only logical that each cycle that produce a cipher text from 64 bits of plain text should be handled by a thread with in a block. Each of these Feistel Network iterative cycles will be divided in to threads that are part of a blocks that will be computed in parallel. In other words, they will be smallest component that will be computed independently from the other threads in parallel without the need of synchronization while computing the results of each block. Furthermore, how the blocks and number of threads assigned to each block also can alter the results. To gain the optimal results, maximum possible amount of threads should be assigned to each block.

## *1.2 Compute Unified Device Architecture (CUDA)*

CUDA or Compute Unified Device Architecture refers to, as the name suggest, the architecture of the GPU device as well as the programming language and the interface. As it is discussed in the paper by S. Che et al. [1] CUDA is an extension to C language. It contains number of additional commands that is used with the C syntax to utilize the large number of threads run by many core processors built in to the GPU. Above

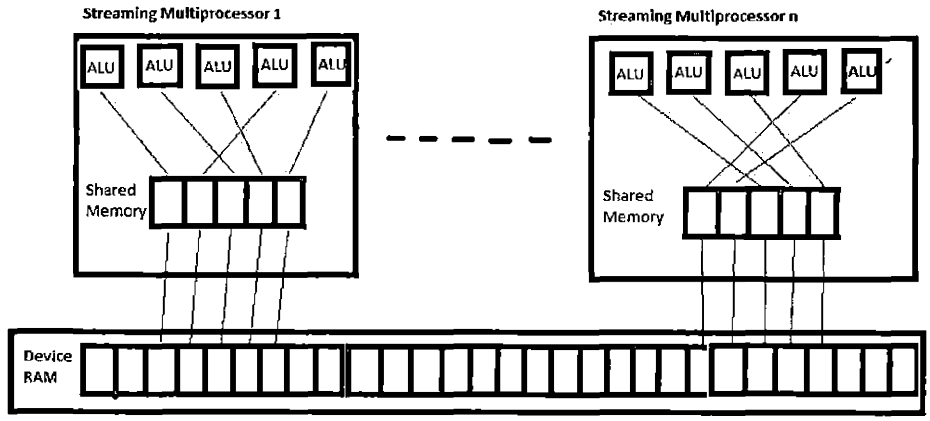


*Fig. 1.3:* Basic GPU Architecture

figure illustrates the basic layout of the architecture.

Memory management for the GPU is an important drawback that always needs the attention of the programmer when trying to implement an algorithm such as Blowfish encryption. The GPU has its own dedicated global memory and every launched application or kernel must have memory allocated and data has to be transferred from the host or the CPU for the computation, and then the answer must be transferred back. This incurs a significant latency penalty, and is a drawback of a GPU when compared to a CPU.

## *1.3 Related Work*

There are number of similar work in the area of using GPU to obtain speed up in encryption algorithms. Further more, there are number of studies published that utilizes the CUDA architecture and programming language to implement Blowfish or similar blocking cipher that utilizes Fiestel networks. For instance the work of J.A.

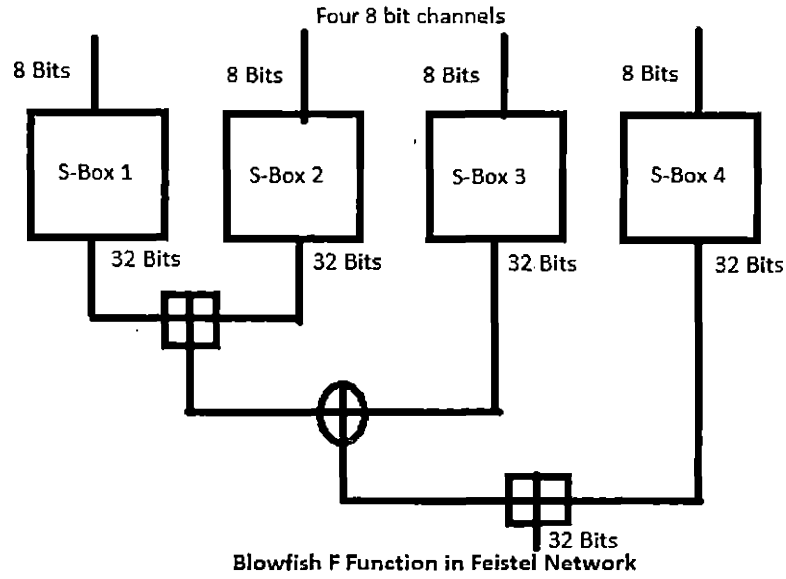


*Fig. 1.4:* F Feistel Function

Feist [2] specifically deals with implementing the Blowfish algorithm with the use of CUDA.

Work of J.A. Feist [2] differs from this thesis in number of ways. First, in J.A. Feist's work, the values used in S-Boxes and P-Arrays are randomly generated rather than being implemented in to the code. In this work, these values are previously calculated hence they will not need to be computed during the execution of the algorithm. There is a trade of between increased security and speed due to the use of previously calculated values. Second difference is the handling of plain text data. J.A. Feist [2] uses preprocessed 32 bit plain text arrays where in this work, as it is described in the original algorithm by Bruce Schneier [6], 64 bits of plain text data is split in to two 32 bit haves and processed according to the original specifications of the algorithm. Last but maybe the most important difference in J.A. Feist [2] work

is that only the Encryption process has been implemented in his work. Decryption process in the blowfish algorithm has not been implemented in both CPU or GPU implementations.

The work done by Z. Wang et al. [7]discuses the advantages of utilizing CUDA on MD5 and Blowfish algorithms and proposes to combine them to get better security and performance over both algorithms. It is more of a comparative work on MD5, Blowfish, and proposed MD5-Blowfish combined algorithm. Whereas this thesis specifically goes in to the benefits of using CUDA on GPU versus C on CPU and explores the possibilities of gaining performance between GPU versus CPU implementation with the use of CUDA. Certainly there are similarities with the approach to increase performance with the utilization of CUDA on GPU. However, this thesis goes in to detail and explores and expends the idea of massive performance increase possibilities with the utilization of CUDA on GPU for the blowfish algorithm alone. Further more, this work does not propose a new algorithm. Rather, it proposes new approach to an existing algorithm to gain performance. In other words, purpose of this work is not to introduce a new combined algorithm, but to re implement the existing blowfish algorithm with CUDA and explore in detail the possibilities of performance gain that can be accomplished by this approach.

There are vast number of works published in the area of encryption using GPU. One work that deals with specifically AES encryption algorithm and use of CUDA architecture is published by H. Nguyen [4] in GPU Gems 3. AES is in the same family of block ciphers as Blowfish. Even though AES and blowfish ciphers are both block ciphers, they differ in the fact Blowfish utilizes Feistel network, whereas AES is

based on Rijindael cipher. Some of the important qualities that sets Rijndael cipher apart from Blowfish and numerous other ciphers are; use of symmetric key, fixed 128 bit block size, and fixed 128, 192, and 256 bits key lengths as it is stated in the paper by H. Nguyen [4] where similar to this thesis implementation of AES encryption algorithm is implemented with the use of CUDA. Furthermore, H. Nguyen [4] goes in to explaining how GPU processes the data with the CUDA implemented AES encryption algorithm and illustrates in great detail how a plain text goes through all of the cycles and iterations to get to the state of cipher text. When comparing H. Nguyen's work to this thesis, one can see clear parallelism in the way of approaching the subject and how both work explores and experiments with a particular type of encryption algorithm and how we can gain information on inter works of utilizing GPU and performance increase.

### *1.4 Significance*

Currently, utilization of encryption for everyday use is not a common practice. One of the big reasons for this tendency is the fact that encrypting personal or business related data can be very time consuming process for individuals when they would like to have this process done in a fast and efficient way that would not take up a lot of time out of their daily life. Encryption with the use of GPU can address this issue. Use of GPU hardware along with implementing an encryption algorithm such as Blowfish for this purpose would solve this issue with the expected speed up it is going to provide. Once the issue of having to wait for a considerable amounts of time to have the encryption process done is resolved through use of GPGPU and CUDA, it

will be much easier for individuals to take advantage of the security that encryption can provide for them on protecting their data.

## *1.5 Purpose*

Perhaps the most important purpose of seeking to gain a speed up for the Blowfish algorithm is to make it fast and in turn making it user friendly for everyday use. Encryption of a data especially if the amount of data is large can be a very challenging task to complete in a fast manner. This alone can be a cause of unpopularity of encryption of data for average everyday use. What this thesis aims to accomplish is to make the process of encrypting data, especially large amounts of data much faster process, and in turn make it less time consuming for the users to encrypt their data. Taking the burden of waiting for encryption process for a long time out of the equation can instill the habit of utilizing this method to secure their data.

# 2. BLOWFISH ALGORITHM

Blowfish algorithm has been designed and introduced by Bruce Schneier in 1993. Blowfish is an open source algorithm that Bruce Schneier placed the C code of this algorithm in a public domain for everyone to utilize it. Since its creation, blowfish has been very widely used by many industry professionals and companies due to the fact it has never been broken still to this day. Currently many companies such as CrashPlan for securely backing up their client data, OpenSSH and PuTTY for establishing secure SSH communication, OpenBSD and many more Linux distributions for protecting the used data utilize it for this purpose. However, the fact of performance issues still remains due to the fact that as the key length grows time and computational power cost of the algorithm also grows. This presents a challenge of making the algorithm function faster. Parallelizing this algorithm with the use of CUDA is the answer to this problem at hand. Since the massive parallelization capabilities of CUDA is available to us, and since the block ciphers are relatively easier to parallelize due to their nature.

## *2.1 Algorithm*

Blowfish utilizes a Fiestel Network much like many other block ciphers that it has competed against over time. Fiestel networks simply iterate a function n times to

produce the guaranteed reversible result. Blowfish utilizes this method and takes the plain text that needs to be encrypted and runs it through this simple function 16 times to produce the cipher text. Consider the following algorithm that describes how the Blowfish cipher is implemented:

As described by Bruce Schneier [6], the Blowfish block cipher algorithm is a variable-length key, 64-bit block cipher. The Blowfish algorithm has two parts: First part is the key generation part and second part is the data encryption part. Key generation part converts a key of at most 448 bits into number of sub key arrays totaling 4168 bytes. Data encryption part takes place through a 16-round Feistel network. Each round has a key-dependent permutation, and a key- and data-dependent substitution. All of the operations in the data encryption part are XORs and additions on 32-bit words. Blowfish algorithm makes use of a large number of sub keys. It is required that the all of the keys be precomputed before any data encryption or decryption can be done.

**1.** Following is the steps to compute the keys:

**1.1.** The P-array contains 18 32-bit sub keys:

- P[1], P[2],P[3],..., P[18].

**1.2.** There are four 32 bit S boxes and each contain 256 entries:

- S1[0], S1[1],S1[2],..., S1[255];
- S2[0], S2[1],S2[2],..,, S2[255];
- S3[0], S3[1],S3[3],..., S3[255];
- S4[0], S4[1],S4[3],..,, S4[255].

2. Following is the steps to do the Encryption process:

Blowfish is a Feistel network consisting of 16 rounds (see Figure 1.2).

- The input is a 64-bit plain text data, x.
- Divide plain text data x into two 32-bit halves:
- Left half xLeft and right half xRight
- For i = 1 to 16:
- xLeft = xLeft XOR Pi
- xRight = F(xLeft) XOR xRight
- Swap xLeft and xRight
- Next i
- Swap left half xLeft and right half xRight (Undo the last swap.)
- xRight = xRight XOR P17
- xLeft = xLeft XOR P18
- Recombine left half xLeft and right half xRight
- Function F:
- Divide left half xLeft into four eight-bit quarters: a, b, c, and d
- F(xLeft) = ((S1[a] + S2[b] mod 232) XOR S3[c]) + S4[d] mod 232

Decryption is exactly the same as encryption, except that P[1], P[2],P[3],..., P[18] are used in the reverse order.

With the Blowfish block cypher algorithm [6] next step is to get the algorithm transferred in to CUDA architecture. First thing that needs to be done in transforming the algorithm to CUDA is the generation of the round keys or the P array as it was described in Mukherjee et all [3]. Keys are generated in the CPU as it is a sequential task and there is no need to utilize GPU for the key generation. Once the keys have been generated the entire data set (P array, S-boxes, and the plain text) has to be transferred to the GPU for the next step. When the data is being transferred to GPU, memory allocation of the data being transferred is one of the important concerns. Ideally cache would be the best option for all of the data, however due to the fact there is only a small amount of cache memory availability in the GPU, only the P array should be stored in the constant cache and S-boxes can be handled in the shared memory in GPU. As for the plain text and cipher text can be handled by the global memory. Since memory management for the blowfish cipher is completed on the GPU, getting the algorithm in a parallelized form is the remaining task to complete the entire process. Each CUDA thread can encrypt and write back one 64 bits block of plain text. Each CUDA block can have 512 threads per block. This way GPU has enough work in the pipeline to process. Maximizing the number of threads running concurrently in each block maximizes the speed up that can be achieved from this process. As it can be seen in figure 1.1 CUDA architecture, threads are contained in blocks and blocks are contained in grids. CUDA architecture utilizes the SIMD (Single Instruction Multiple Data) approach where multiple processors performing the same operation on multiple data in parallel. Thus, as the amount of data and number of processing elements (threads) increases, this approach

should produce better results, in other words it should provide higher speed-up. As it has been demonstrated above with the algorithm for Blowfish cipher, CUDA architecture should utilize the parallelizing nature of this algorithm and produce high rate of speed-up compared to CPU utilized process.

In this thesis, NVIDIA GeForce GT 230M [5] GPU hardware is being used along with NVIDIA CUDA GPU Computing SDK 4.0 for all of the procedures including coding with the use of CUDA C/C++ and testing phases. NVIDIA GeForce GT 230M graphics card contains 48 CUDA cores each operating at 1100 MHz, and it has 1 GB of dedicated memory with a memory bandwidth of 16 GB per second.

### *2.2 CPU Implementation*

Appendix 4.2 contains the C code that is written for this thesis. It is based on the Blowfish encryption algorithm code that is published and placed in public domain by Bruce Schneier in 1993. However, it has been rewritten to be used for this thesis. Original work of Bruce Schneier has been used as basis for this work. Entire code including all of the functions and the hexadecimal values have been re-written to serve for the purpose of this work.

The code implemented for this thesis uses precomputed hexadecimal digits that were randomly generated for security purposes for generating the S-Box and P-Box values outlined in the Blowfish algorithm. Bruce Schneier uses the precomputed hexadecimal digits of Pi for his implementation. Since these values have been used numerous time by many others, for this work, all of the hexadecimal values have been regenerated for security and originality purposes along with recreation of all of the

functions used in the original implementation. It takes 64 bits of data and splits it in to two 32 bits parts (Left and right as stated in the algorithm) and follows the rest of the algorithm in a sequential manner to encrypt the plain text. Implementation of the algorithm for this work follows the original ideas introduced by Bruce Schneier and uses the same ideas to come up with the procedures to implement the algorithm in a way that will serve the purpose of this thesis.

## *2.3 GPU Implementation*

After the analysis of the Blowfish algorithm to seek parallelism, it can be stated that there is data-level parallelism. Parallelizing the 16 rounds that are part of the algorithm, due to the nature of the Feistel network it is utilizing, is not an option since each round has to be in a sequential order and has to be completed in the specific order to obtain the required result. Thus, each block of data that has to go through this sequential cycle can not be implemented in parallel. However, since the sequential order only matters per each block of data, parallelizing each blocks of data would be the logical method to convert the CPU intended implementation of the Blowfish algorithm to GPU intended CUDA implementation that will bring speed up. With the analysis and understanding of the algorithm and further more, with figuring out how to implement and convert the sequential algorithm in to parallel, coding both the CPU and GPU version of the algorithm is the next step in the process of completing this work. Both CPU and GPU versions of the Blowfish algorithm implementation can be found in the Source Code section of the Appendix.

CUDA contains all of the necessary elements to allow a programmer to fully par-

allelize a algorithm. Understanding the Blowfish algorithm and being able to figure out the best way to implemented in a parallel with the use of CUDA is the main goal of the code in Appendix 4.2. A given algorithm may or may not be optimal for parallelization and for a programmer to identify the nature of the algorithm can only be done through analyzing and being able to see certain characteristics in it. There may be instances where only the certain parts of a given algorithm can be implemented in a parallel way. Thus, identifying if the Blowfish algorithm can be fully or partially parallelized is the first step of converting the sequential CPU code above to a efficiently parallelized GPU (CUDA) code. When looking in to the Blowfish algorithm, it is clear that parts of it cannot be implemented in parallel, For instance, required iterative function in the encryption and decryption process must run16 rounds in a specific sequential manner. In other words, when trying to have the each round completed by a thread, 16 threads will be one single block. Thus, this makes it not possible for implementing it in a parallel manner. However, when examining the algorithm further, it becomes obvious that since each block must be processed in a specific sequential manner, each block of data can be processed by one thread. Thus, best methods to create parallelism in the Blowfish algorithm is to taking an entire block of data and allow a thread to handle it completely. With this method, data-level parallelism can be utilized. Since each thread will be processing a block, using the maximum allowed, 512, thread limit for the GPU will be the logical choice to maximize the speed up.

GPU can handle massive numbers of threads that runs in parallel as mentioned previously. One of the challenge of implementing a algorithm to utilize the 512 thread

limit can be found due to the data size of any given set of instances. This becomes an issue if data set is small and can not be distributed in to all of the threads that are available to use with in the GPUs limitations. This creates a reduction in the speedup gained by the GPU over CPU. Another reason for the lack of optimized performance and scalability is due to the implementation of the algorithm. When implementation is not able to distribute the load over to all of the available threads, results will stray away from the optimum performance and scalability. This issue can be further studied and improvement on reduction of scalability and performance can be optimized.

Memory transfer, especially if it is with the host memory, is probably the single costliest item in the GPU processing. This issue is one of the areas that would require further study and research to find out if there are ways to eliminate or minimize this cost. It would be especially beneficial if the use of host memory can be eliminated due to its very high cost in time. In this thesis there were not any specific efforts made to mange the Memory transfer, however, it is clear that it is one of the most important aspects of improving the speed up that is gained by the use of GPU.

# 3. RESULTS

With both CPU and GPU implementations at hand, next step is to run some test and figure out if there are any speed up and if there are how do they correlate to the data size, number of blocks, and number of threads.

## *3.1 Methodology*

Testing methodology for this work is based on the theory that CPU implementation of Blowfish algorithm can be sped up by implementing it on GPU due to its massive parallelization potential. Tough, speed up is expected with the use of CUDA implementation of the Blowfish algorithm to be computed on GPU, it is also imperative that looking in to the correlation between data size and amount of speed up that is gained are parallel. In other words, as the data size increases, the amount of speed up gained increases as well. CPU version of the Blowfish implementation seems to be no competition even with small data sizes. Though, speed up gained with GPU implementation is much less at the lover end of data size.

The methodology used for testing is simply using different sets of data and data sizes to test the possible speed up gained by different numbers of blocks ranging from 1 to 32000 blocks per test run. Reason for testing number of different block sizes is to simple find out about the relationship between data size and amount of speed up

gained by the use of GPU versus CPU. As it was predicted, the data size plays a big role in the amount of speed up gained by the use of GPU compared to CPU. In each test run there were a different number of blocks tested against CPU and GPU to obtain the run time of both GPU and CPU to ultimately find out the speedup gained by GPU compared to CPU. The system used for this test is a HP Pavilion dv6t-2000 model with a Intel(R) Core(TM) i7 CPU with a 4 GB of RAM and a NVIDIA(R) GeForce(TM) GT230M GPU unit with a 1 GB of dedicated graphics memory. Microsoft Visual Studio 2008 with CUDA 4.0 has been utilized to run the test.

### *3.2 Analysis*

Table 3.1 contains the findings of these tests. In seven different set of tests illustrates the difference between the use of small data size and small number of blocks versus larger data size and number of blocks. Each test run is used to obtain GPU runtime, CPU runtime, and the speedup gained by the use of GPU in small to large data sets and block sizes.

Table 3.1 contains sets of data for Number of Block, GPU runtime in ms, CPU runtime in ms, and Speed up. For each test run varying item in the Table 3.1 is the Number of Blocks. There are seven different number of blocks for each test ranging from one to 32000. As the number of blocks change, the GPU and CPU runtime in ms changed for each test run. Speed up is obtained by comparing the GPU runtime versus CPU runtime to find out how much of a speed up is gained by GPU in each test instance depending on number of blocks. Ultimate goal of the tests and the data

*Tab. 3.1:* CPU vs GPU Comparison for Different Sizes.

| Number of Blocks | GPU runtime in ms | CPU Time in ms | Speed up |
|---|---|---|---|
| 1 | 1.7248 | 0.5132 | 0.2975 |
| 100 | 2.4201 | 34.6753 | 14.3280 |
| 1000 | 8.4821 | 306.5874 | 36.1452 |
| 4000 | 22.3891 | 1260.5367 | 56.3013 |
| 8000 | 42.5267 | 2536.8456 | 59.6530 |
| 16000 | 53.5622 | 5169.2549 | 96.5093 |
| 32000 | 105.5681 | 11442.4586 | 108.3893 |

presented in the Table 3.1 is to find out about the relationship between the number of blocks and runtime for both GPU and CPU, than compare the runtime of GPU and CPU to figure out the Speed up gained by GPU versus CPU.

Table 3.2 contains sets of data for Number of test runs, Number of Blocks, Data size in KB, and Speed up gained by the use of GPU. Table 3.2 simply help analyzing the relationship between amount of speed up gained, data size and number of blocks. As it can be seen from the Table 3.2, there is a direct relationship between data size, number of blocks, and speed up gained by use of GPU. It can be clearly seen that as the data and number of blocks increase, speedup gained by the use of GPU also increases.

The comparison between GPU and CPU with different block numbers and data sizes compared to time it took for it to run shows the Speedup gained by the use

*Tab. 3.2:* Data Size, Number of Blocks, and Speed Up Relationship.

| Test Run | Number of Blocks | Data Size in KB | Speed up |
|---|---|---|---|
| 1 | 1 | 4 | 0.2975 |
| 2 | 100 | 400 | 14.3280 |
| 3 | 1000 | 4000 | 36.1452 |
| 4 | 4000 | 16000 | 56.3013 |
| 5 | 8000 | 32000 | 59.6530 |
| 6 | 16000 | 64000 | 96.5093 |
| 7 | 32000 | 128000 | 108.3893 |

of GPU versus CPU. Ultimate goal of this experiment is to find out if the expected result of gaining speed up with the use of GPU versus CPU can be clearly seen in the results. Figures below will further demonstrate the speed up and relationship between percentage of speed up and number of blocks in each test. Figure 3.1 illustrates the speed up and block size relation by illustrating the comparison of specific numbers of block numbers used in the test runs and comparing those values to the speed up gained by GPU where GPU and CPU runtime in ms results help us find the speed up for each test run.

Figure 3.2 illustrates the CPU and GPU run times against number of blocks that had been processed in each run by simply comparing the number of blocks used in each test run to the results of both GPU and CPU runtime in ms for all of the test runs.

Figure 3.3 shows the relationship between number of blocks used for each test run

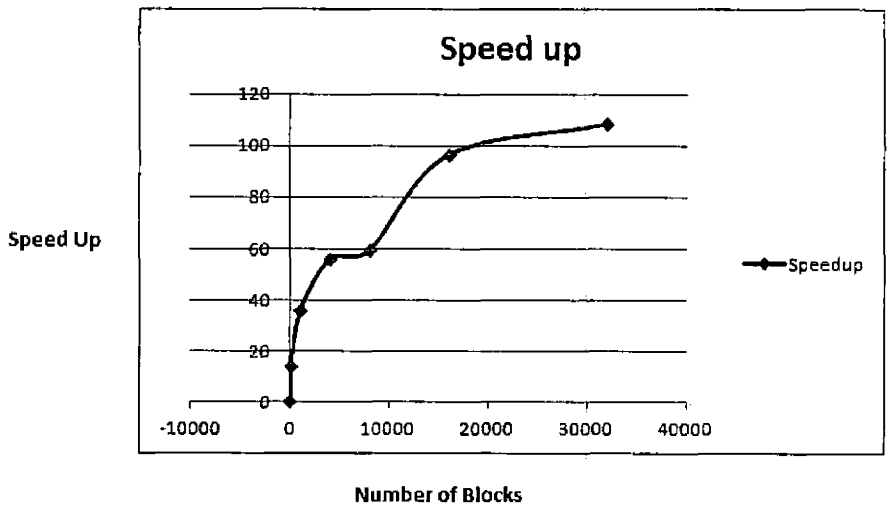


*Fig. 3.1:* Speed-Up Gained by GPU Processing

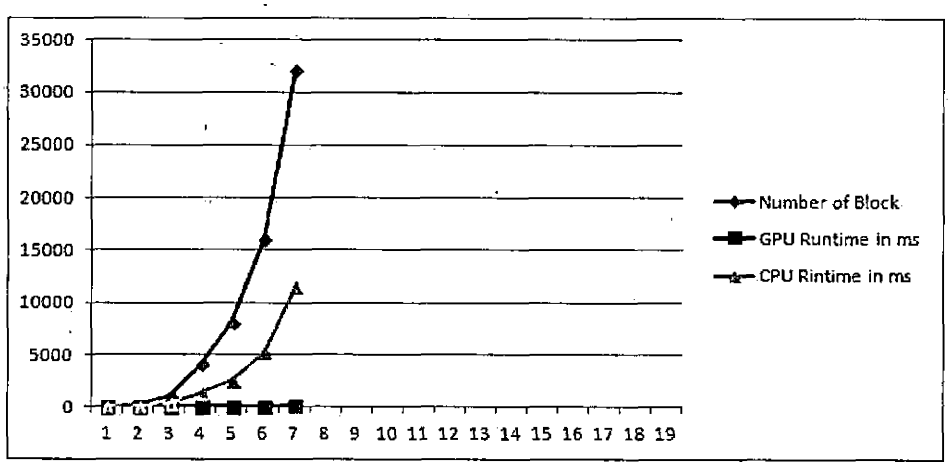


*Fig. 3.2:* GPU and CPU Runtime Compared to Amount of Blocks

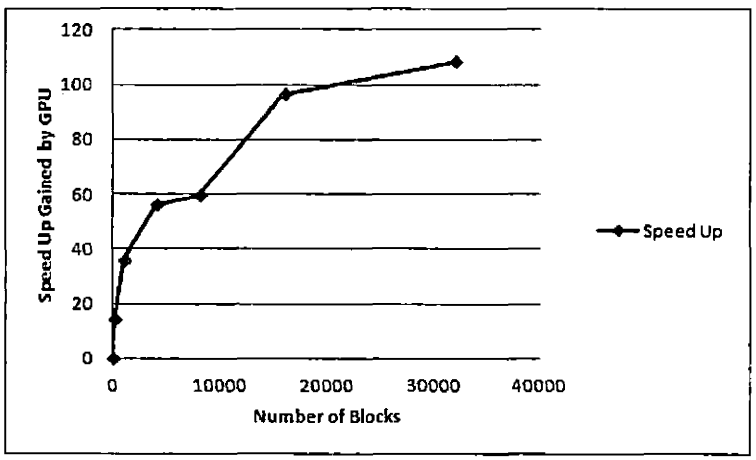


*Fig. 3.3:* Number of Blocks Used for Each Test Run Compared to Speed Up Gained by the Use of GPU

and speed up gained by the use of GPU for that particular test run. It helps us understand the correlation between the number of blocks used for an test instance and effect it has on the speed up gained by the use of GPU compared to CPU.

Figure 3.4 shows the relationship between data size for each test run and speed up gained by the use of GPU for that particular test run. It helps us understand the correlation between the data size and effect it has on the speed up gained by the use of GPU compared to CPU.

As it is clearly demonstrated above, there is a tremendous speed up gained by utilizing the power of GPU. CPU and GPU implementations of blowfish algorithm very clear demonstrate that as the data size and number of blocks increase in a given set of data, amount of speed up that can be gained increases along with it. It is clear that ability to breakdown blocks and run them in parallel threads do give results that favor the use of GPU versus CPU especially data size increases.

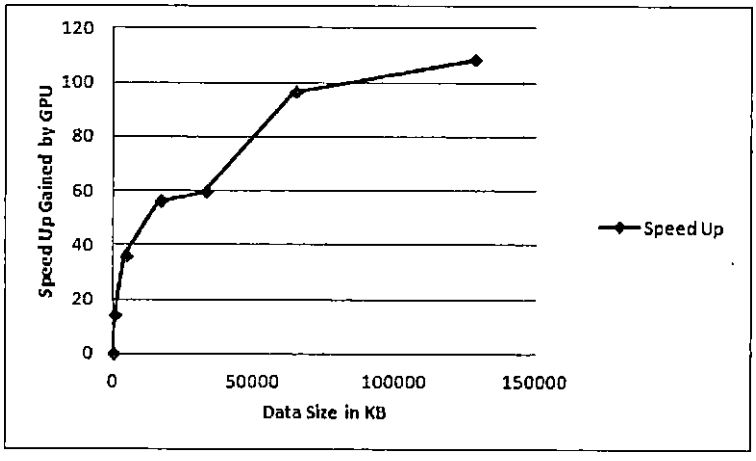


*Fig. 3.4:* Data Size of Each Test Run Compared to Speed Up Gained by the Use of GPU

# 4. CONCLUSION

## *4.1 Accomplishments*

This thesis has explored the possibility of gaining performance and speed up with the utilization of GPU for blowfish cipher. Results clearly indicate that tremendous amount of speed up can be gained by implementing the blowfish algorithm in CUDA to work with GPU rather than using traditional CPU implementation. This work and all of the results obtained form the experiments show that there are massive amounts of speed up that can be gained by implementing the Blowfish encryption algorithm to utilize GPU rather than CPU to gain great deal of speedup. Findings of this thesis can be used to understand the potential power of using GPU for encryption and possibly other types of algorithms. It is clear that there is a great deal of speed up and performance that can be gained by using GPU with the help of CUDA programming language and architecture.

## *4.2 Future Directions*

There are number of different studies published and currently being done in this area of research. Though it is a new and expanding field in the computing, GPGPU and specifically encryption with the use of GPU can be further studied. This thesis only explored the use of GPGPU for the Blowfish encryption algorithm because the fact

that it is a block cipher that uses Feistel network. There are a large number of other block cipher algorithms that use the Feistel network and because of the similarity of these algorithms, each one of these algorithms should separately be studied and compared against each other to have better understanding of the relationship between them for ultimate goal of gaining better performance and further speed up. Further more, different types of algorithms, other than encryption, should be experimented and studied to find out if there are ways to gain similar or even better performance and speedup with the use of GPU and CUDA.

# APPENDIX A

# CPU CODE

```
#ifdef little_endian    /* Eg: Intel */
   #include <dos.h>
   #include <graphics.h>
   #include <io.h>
#endif

#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#ifdef little_endian    /* Eg: Intel */
   #include <alloc.h>
#endif

#include <ctype.h>

#ifdef little_endian    /* Eg: Intel */
   #include <dir.h>
   #include <bios.h>
#endif

#ifdef big_endian
   #include <Types.h>
#endif

typedef struct {
  unsigned long P[18];
  unsigned long S[4][256];
} BFish_CText;

#define MAXKEYBYTES 56          /* 448 bits */
#define big_endian 1            /* Eg: Motorola */
void BFish_Init(BFish_CText *CText, unsigned char *key, int keyLen);
```

```
void BFish_Encrypt(BFish_CText *CText, unsigned long *xLt, unsigned long *xRt);
void BFish_Decrypt(BFish_CText *CText, unsigned long *xLt, unsigned long *xRt);

#define N               16
#define noErr            0
#define DatERROR        -1
#define KEYBYTES         8

FILE*          SubkeyFile;

unsigned long F(BFish_CText *CText, unsigned long x) {

   unsigned short a, b, c, d;
   unsigned long  y;

   d = x & 0x00FF;
   x >>= 8;
   c = x & 0x00FF;
   x >>= 8;
   b = x & 0x00FF;
   x >>= 8;
   a = x & 0x00FF;
   y = CText->S[0][a] + CText->S[1][b];
   y = y ^ CText->S[2][c];
   y = y + CText->S[3][d];

   return y;

}

void BFish_Encrypt(BFish_CText *CText, unsigned long *xLt, unsigned long
*xRt) {

  unsigned long  xLt;
```

```
  unsigned long  xRt;
  unsigned long  temp;
  short          i;

  xLt = *xLt;
  xRt = *xRt;

  for (i = 0; i < N; ++i) {
    xLt = xLt ^ CText->P[i];
    xRt = F(CText, xLt) ^ xRt;

    temp = xLt;
    xLt = xRt;
    xRt = temp;

  }

  temp = xLt;
  xLt = xRt;
  xRt = temp;

  xRt = xRt ^ CText->P[N];
  xLt = xLt ^ CText->P[N + 1];

  *xLt = xLt;
  *xRt = xRt;

}

void BFish_Decrypt(BFish_CText *CText, unsigned long *xLt, unsigned long
*xRt) {

  unsigned long  xLt;
  unsigned long  xRt;
```

```
  unsigned long  temp;
  short          i;

  xLt = *xLt;
  xRt = *xRt;

  for (i = N + 1; i > 1; --i) {
    xLt = xLt ^ CText->P[i];
    xRt = F(CText, xLt) ^ xRt;

    /* Exchange xLt and xRt */

    temp = xLt;
    xLt = xRt;
    xRt = temp;

  }

  /* Exchange xLt and xRt */

  temp = xLt;
  xLt = xRt;
  xRt = temp;
  xRt = xRt ^ CText->P[1];
  xLt = xLt ^ CText->P[0];

  *xLt = xLt;
  *xRt = xRt;
}

short InitializeBFish(BFish_CText *CText, char key[], short keybytes)
{
   short          i, j, k, error, numread;
   unsigned long  Dat, Datl, Datr;
```

```
unsigned long sbox0[] = {
0xd2120ba5L, 0x87d9b4acL, 0x299d62dbL, 0xd02ad9b6L, 0xb7f2a9fdL, 0x5a256f85L,
0xba6c8034L, 0x922c6988L, 0x23a28836L, 0xb1825c96L, 0x070292f2L, 0x747f9c25L,
0x515820d7L, 0x62463f58L, 0xa3479fa1L, 0x93811d6fL, 0x0d846379L, 0x627fb547L,
0x627bcd47L, 0x72243affL, 0x6b43a32dL, 0xc24a48b4L, 0x8c10d418L, 0x2a925021L,
0xc4d2b021L, 0x27507490L, 0xca326827L, 0xb7db17f9L, 0x7f68dcb0L, 0x501a270fL,
0x5c8f0f7bL, 0xb02f7a1fL, 0xd62466c2L, 0xbd123b26L, 0x67a929daL, 0x44504c50L,
0xf5442491L, 0xaa44ab83L, 0x46378752L, 0x51f72330L, 0x44ca185aL, 0x2aab20b5L,
0xb3cc4c13L, 0x2232f7cfL, 0xa24375a9L, 0x6c62f881L, 0xb1ff2322L, 0x5159bc2aL,
0x2ba8c44dL, 0x63271295L, 0xcf4c1f25L, 0x8b76812fL, 0xa9d5ba11L, 0x5c23c94cL,
0x6a124172L, 0x27847566L, 0x1b793787L, 0x5b3bb8a9L, 0xc3b9f72bL, 0x55272281L,
0x52d708ccL, 0x9b22a882L, 0x376cac50L, 0x4dfc7012L, 0xf9734d4dL, 0xf87464b2L,
0xdc252102L, 0xfb542b77L, 0x21781f72L, 0xd185acc4L, 0x095d5991L, 0x71933218L,
0x2f0b3372L, 0xa3732003L, 0x58c7903aL, 0x8f298b4fL, 0x22c55732L, 0x95f85c8aL,
0x560c8c52L, 0xabd17790L, 0x5a42a0d2L, 0xd7432957L, 0x8509a627L, 0xab4211a1L,
0x5ff90b5cL, 0x216a1bf3L, 0xba1b9040L, 0x6f9b2a87L, 0xa292542dL, 0x18a90265L,
0x55ca481fL, 0x72310f77L, 0x7cff7528L, 0x345989b3L, 0x6d73a4c1L, 0x1b7b4fbfL,
0xf05964d7L, 0x74c22061L, 0x302a3389L, 0x45c25aa5L, 0x3fd1aa52L, 0x15196605L,
0x2b9fd962L, 0x328b021dL, 0x16d0d623L, 0xd00a2237L, 0xdb09fad1L, 0x3892c08bL,
0x064162c8L, 0x70882b6bL, 0x24d368d7L, 0x95f7df96L, 0xf19f402aL, 0xb5683c1bL,
0x865cf0bdL, 0x03c005baL, 0xc2a839b5L, 0x308950c3L, 0x4f4c8fc2L, 0x285a2351L,
0x579b59a9L, 0x1f5c41b4L, 0x2118b2fbL, 0x1b42fc59L, 0x5d9c4229L, 0x8b10842cL,
0xcc723433L, 0xa94fbd08L, 0xbff1d003L, 0xdf113a9dL, 0x55092706L, 0x282f3bb1L,
0xc0cba746L, 0x34c76309L, 0xd20b4918L, 0xb8d19bdbL, 0x4468c0bdL, 0x2a50120aL,
0xd5a200c5L, 0x302c6268L, 0x5689249fL, 0x9b29a1ccL, 0x7fa4f897L, 0xdb122297L,
0x1c6425d9L, 0x9d525b24L, 0x29402fc7L, 0xad0442abL, 0x121db49aL, 0x9d217650L,
0x41126b37L, 0x1f00d972L, 0x8f4c46bbL, 0xca597ca0L, 0x2a76452fL, 0xd92658dbL,
0xd432a795L, 0x276f99c1L, 0xac5612c5L, 0x7c394461L, 0x584b26b0L, 0xbbca47c7L,
0xf299a14dL, 0xb79022a0L, 0x209a1d87L, 0x9d2271b7L, 0x3a9cb45cL, 0x2dd2d14bL,
0x8a41f368L, 0xb5973454L, 0xd27f38bcL, 0x3b9b8680L, 0xf2dd92daL, 0xa3cb6f11L,
0x529b2132L, 0xcff3c5f7L, 0xf920cadaL, 0x15663c02L, 0xd06f8f9fL, 0x2b9229b3L,
0x84dbda3dL, 0xaf808287L, 0xfaad7f62L, 0x5b81d4a0L, 0xd07fd2d0L, 0xa9c624f0L,
```

```
0x7f1c4b29L, 0x7f6483b6L, 0x7995f29bL, 0x92222b53L, 0x7777b722L, 0x800d902cL,
0x39ad4fa0L, 0x5779c12cL, 0xd2c99282L, 0xb1a7c2adL, 0x29292227L, 0xbf0f2666L,
0xfa642d9fL, 0x7b0229a2L, 0xf4a0cc09L, 0xb45963f7L, 0x27ac91d5L, 0xcf78f288L,
0xb3a739f0L, 0x9d21f0b6L, 0x6cc31b72L, 0xd2ada7d8L, 0x2549a255L, 0x70846604L,
0x81cc6123L, 0x222a2366L, 0xf5ad2054L, 0x66b49a75L, 0xc6433294L, 0x9b8d14c9L,
0xfbcda90cL, 0x6b1f78a0L, 0xd5322bd1L, 0xaf2f6f38L, 0x00240f2dL, 0x2062b14fL,
0x225700bbL, 0x46b7f0a9L, 0x2353158bL, 0x9008b82fL, 0x4451822dL, 0x48d9a5aaL,
0x67c23178L, 0xd84a4169L, 0x206d4ba2L, 0x02f4b8c4L, 0x71250165L, 0x5284c9a8L,
0x22c72857L, 0x3f613a32L, 0xb1362dcaL, 0x6b23a83aL, 0x2b420042L, 0x8a412824L,
0xd5094619L, 0xbc8bc5f3L, 0x2b50a365L, 0x72f56300L, 0x07ba59b4L, 0x462bf829L,
0x9285fc5bL, 0x2a0dd824L, 0xb5515422L, 0xf6b898b5L, 0x9913042fL, 0xc4744553L,
0x41b02d4dL, 0xa88979a2L, 0x07ba3688L, 0x5f74065aL};

unsigned long sbox1[] = {
0x3b6a60f8L, 0xb4b12833L, 0xdb64082fL, 0xc3282521L, 0xad5fa5b0L, 0x38a6d96dL,
0x8cff50b7L, 0x79fdb255L, 0xfcaa7c62L, 0x588a2699L, 0x4553425cL, 0xc2b28ff2L,
0x281502a4L, 0x64083c28L, 0xa0482130L, 0xf3271a1fL, 0x1943878aL, 0x4b328d54L,
0x5b79f3d5L, 0x889619d5L, 0xa2d28c06L, 0xf9f71094L, 0x3d2d17f5L, 0x90244dc2L,
0x3cdd2075L, 0x7360fb25L, 0x5172f8c5L, 0x022fcc4fL, 0x08575b19L, 0x1fbaf9c8L,
0x1c862723L, 0x5b5a60a2L, 0x57691473L, 0x42a0f275L, 0xb68c4104L, 0xaa400616L,
0x1f06732cL, 0x69dfaf4cL, 0x7f6d33fcL, 0x462592b7L, 0xb01ada16L, 0x90400c0dL,
0x902c2903L, 0x0200b199L, 0xaf0c942aL, 0x1cb463b2L, 0x24716a47L, 0xdc0822bdL,
0xd2822198L, 0x6ca82995L, 0x83123661L, 0x22943602L, 0x1af4f472L, 0x16c2dadcL,
0xc7b46513L, 0x8a91dda6L, 0xa8335235L, 0x09d0010fL, 0xfcc7c61fL, 0xa3642f32L,
0xf217cd88L, 0x1bfa0f29L, 0x1270bba2L, 0x271fb112L, 0x3f437b17L, 0x395db807L,
0x59320d01L, 0x950a03b9L, 0x2cb72280L, 0x23866c68L, 0x4568b062L, 0xbca978a9L,
0xdf8a6629L, 0xd8810720L, 0xb17baf22L, 0xdcc9192fL, 0x44226229L, 0x2f5b6223L,
0x402addf5L, 0x8973cd76L, 0x6a473627L, 0x6307da26L, 0xbc898abcL, 0xf83b6d7cL,
0xfc6afc1aL, 0xdb742d9aL, 0x51083155L, 0xc353c1d2L, 0xf92c2736L, 0x1224d807L,
0xdd311b16L, 0x23c2ba25L, 0x22a23d31L, 0x2a54c342L, 0x40830002L, 0x211af3ddL,
0x62d9978fL, 0x20123f44L, 0x72ac66d5L, 0x4922288bL, 0x03144592L, 0xd6a1c65bL,
0x1c22271bL, 0x4823a408L, 0x9279f5fdL, 0x86929b9aL, 0x8fbab92cL, 0x2f241c5fL,
0x75f13460L, 0xfaf859b2L, 0x750f4f0aL, 0x4a1f2ab1L, 0x6629f62cL, 0x3f1d059aL,
0x2854dcb8L, 0x88f62d09L, 0x701f78d5L, 0x4255c724L, 0x2f3cc867L, 0x8c20b15aL,
```

```
0xc5240fbaL, 0x83f2fa67L, 0xa49c1c41L, 0x2f0a2d93L, 0x92963fa6L, 0x152d2b1dL,
0x28182509L, 0x28c26850L, 0x4221a607L, 0x962122b5L, 0xfbad9f5fL, 0xfac12955L,
0xf1bc3484L, 0xa56bc771L, 0xb26916d2L, 0x027c9927L, 0xc112ddf9L, 0xbf5c4aa4L,
0x54472274L, 0x57ab8702L, 0xffcfa409L, 0xdb29841bL, 0x2af96dadL, 0x4b5f2973L,
0x2422b527L, 0x28065260L, 0xfcdd3664L, 0x52892420L, 0x21cca710L, 0xfb52bd85L,
0x01139f2fL, 0xaa0151c9L, 0xb4614c80L, 0x3c60a218L, 0xd48f8f0bL, 0xcbaadf23L,
0xffcc75bcL, 0x50522ca6L, 0x8cab4cabL, 0xb291735fL, 0x537b2fa9L, 0x28bd90caL,
0xa02158b8L, 0x544abb40L, 0x30574a12L, 0x1c2ab3b1L, 0x128ff8d4L, 0xc022b796L,
0x8b430b28L, 0x7649a088L, 0x8496886fL, 0x521d6da7L, 0x9716778aL, 0x86f12d66L,
0x22fd8149L, 0x25572272L, 0x0f147728L, 0xc6f529d5L, 0x85dfd9a2L, 0x6747ba88L,
0x469473a4L, 0x2b226251L, 0x8b71c199L, 0x2ac23585L, 0xcdb10afbL, 0x412f1043L,
0x79d837f3L, 0x5dbc1227L, 0x47fb92f9L, 0x13c599faL, 0x9f27fd52L, 0xff6c1c61L,
0x4d3a23d8L, 0xf753b6f1L, 0x32204d23L, 0x201f21f0L, 0x34fff2b5L, 0xa1aaabfaL,
0xdb5c3924L, 0x9acb39d0L, 0xc6329332L, 0xf95abbb4L, 0x54391b2dL, 0x32cd2204L,
0xd72f688fL, 0x75743dc6L, 0xf33b365aL, 0x1d725240L, 0xc952a292L, 0x4b7d2535L,
0x9c7771a0L, 0xc2c6b5a1L, 0x692423c1L, 0x58cb6382L, 0x36737a0bL, 0x4582b274L,
0x084bb900L, 0xad28378dL, 0x2352b263L, 0x21720f00L, 0x47327d2aL, 0x0c4494faL,
0x2dad931fL, 0x21196052L, 0x11629082L, 0x7d816f32L, 0xd549fc92L, 0x5c221bdbL,
0x6cdf1648L, 0xcbff6350L, 0x307492a6L, 0xcf66125fL, 0xa5067073L, 0x2897408fL,
0xf7f9d744L, 0x52d88614L, 0xa858a6aaL, 0xc40c05c2L, 0x4a03ab9cL, 0x700bcadcL,
0x8f336a2fL, 0xc1341373L, 0x9dd45604L, 0x0f2f8fc8L, 0xdb61dbd1L, 0x204477cdL,
0x5649da68L, 0xf1563130L, 0xc4c31354L, 0x621f17d7L, 0x1d27978fL, 0x925d9920L,
0x241f22f6L, 0x79b01d3aL, 0xf5f1892bL, 0xdb71ad96L};

unsigned long sbox2[] = {
0xf81d4a57L, 0x83723096L, 0x953c252cL, 0x83582813L, 0x32242096L, 0x6502d396L,
0xbc935b2fL, 0xd3a20057L, 0xd3072362L, 0x1120935aL, 0x31b6d3b6L, 0x400052a9L,
0x2f18952fL, 0x86233435L, 0x23223963L, 0xb97b7730L, 0x3d849c2dL, 0x85b482a9L,
0x6093ddd1L, 0x55a02934L, 0xb9bc08fcL, 0x01bd8674L, 0x69ac5dd0L, 0x12cb7403L,
0x85fb26b1L, 0x449d1832L, 0xda2436f5L, 0xabca0a8aL, 0x27406724L, 0x41032893L,
0x0a2c75daL, 0xf8b55d9bL, 0x57dc2352L, 0xd6375800L, 0x570fc0a3L, 0x26a27dffL,
0x39199fa2L, 0xf776ad7cL, 0xb47cf005L, 0x6a93d5b5L, 0xaacf2f6cL, 0xd11649fcL,
0xcf67a188L, 0x305b2a32L, 0x209f8f14L, 0xd89174b8L, 0xff18d6abL, 0x1b223f7bL,
0x2dc89a96L, 0x3b5d2745L, 0x25a15512L, 0xfaf186b2L, 0x1a5f9a63L, 0xdd4b3112L,
```

0x5732f696L, 0xca67209bL, 0x9b0a943fL, 0xd79fb186L, 0x343045acL, 0xba378426L,
0x44411a1aL, 0x20717d76L, 0x9f5ba8b6L, 0xd085843bL, 0x44a756bcL, 0xa2248a47L,
0xcca82851L, 0x88f2db11L, 0xa52a3a45L, 0x19122498L, 0x4f936f2cL, 0x8028126cL,
0x9d97f702L, 0x03262960L, 0x70bb244cL, 0x04272cf1L, 0x84c22437L, 0xf3c55d22L,

0x37c22119L, 0xc60975dcL, 0x0698c8ffL, 0x32032909L, 0x303668a3L, 0x4d775f26L,
0x124942fbL, 0xd48bc0d2L, 0x92bcc279L, 0x32221453L, 0x246b6713L, 0x502a8c50L,
0xd997f7a1L, 0x29515c2bL, 0x0f22b3c2L, 0x02f2128fL, 0xa95539d2L, 0xcad27224L,
0x5b2184f0L, 0x111f82f2L, 0x1b230b52L, 0xffbfb822L, 0x74b2a20fL, 0xf5ba0d88L,
0xdf620c7cL, 0x2da29627L, 0xd0226734L, 0x84b6839dL, 0x536d0752L, 0xf6cc9490L,
0x4338a159L, 0x766d379aL, 0xc18d9d26L, 0x911f7d2fL, 0x0a365132L, 0x882f9963L,
0x1a595fabL, 0x93979d16L, 0xa722dc50L, 0xa2fbdd97L, 0x882bf23cL, 0xdb5f5b0dL,
0xc56b4420L, 0x5d562c16L, 0x2654d31bL, 0xdcd0f703L, 0x92280dc6L, 0xcc0099a1L,
0xb4180982L, 0x5809fd0bL, 0x556b899bL, 0xcfdb6d8cL, 0xa082c90bL, 0xd8244fa1L,
0xbb212977L, 0x424bad23L, 0x6b8368b9L, 0x651bd5fbL, 0x16182fb1L, 0xcc224868L,
0x7025f286L, 0x932f122dL, 0x5732ada6L, 0xc55a2b1bL, 0x22643cccL, 0x672f922cL,
0x5a223216L, 0xb68242f6L, 0x05a2bbf5L, 0x3b9b5140L, 0x2a5b2027L, 0x22cafd9aL,
0x1d24bdd7L, 0xf2f2c1c8L, 0x33322548L, 0x0a222175L, 0xd80cfc5fL, 0xd4abfa2aL,
0x53a9563fL, 0xda75a749L, 0xbfb9f877L, 0x53f3c19fL, 0x8dbc7046L, 0x9096c075L,
0x50676b97L, 0x5001503dL, 0xd29d7135L, 0x951729b0L, 0x6634af03L, 0xd6159cccL,
0x71325b11L, 0x902fab62L, 0xb0703276L, 0x1c004f49L, 0x66a046bfL, 0xbdf7af23L,
0x44353288L, 0xb9472f52L, 0x3f479379L, 0x92dd9da2L, 0x9363f917L, 0x7678bdc2L,
0x415598c1L, 0xc7b17f63L, 0xb3649244L, 0x359cd8b8L, 0x6afb2552L, 0x7b2dd973L,
0x735a0f68L, 0x824984f2L, 0x355f487fL, 0x20b34660L, 0x7cd44482L, 0xc802df3cL,
0xb80bacf2L, 0xbb7204d0L, 0x22a75237L, 0x6463a88fL, 0xb66928b5L, 0xf0a8dc08L,
0x552d08a2L, 0xc3123511L, 0xf74a2902L, 0x0890bf7cL, 0x3a88a024L, 0x2d5f9f20L,
0x2ab81d2dL, 0x0ba4a3d9L, 0xa2759209L, 0x27579258L, 0xdcb6da71L, 0x4618059fL,
0xa2f2cf8bL, 0x39cd6942L, 0x40224f02L, 0xa605719aL, 0xa002b4c3L, 0x0df5d026L,
0x8a977c26L, 0x66197532L, 0xc1503c05L, 0x52a705b4L, 0x90266a27L, 0xc09475f0L,
0x005047aaL, 0x10dc6d52L, 0x22f58fd6L, 0x2117fa51L, 0x41c2dd83L, 0xc2c22513L,
0xbbcbff45L, 0x80bcb5dfL, 0xfb9c6da2L, 0xcf482d65L, 0x5904f308L, 0x3b6c0277L,
0x18620a1dL, 0x6c826c23L, 0x75f16249L, 0x623d8db8L, 0x2ac24bb3L, 0xd18fb79cL,
0xfd434467L, 0x079ca4b4L, 0xd71d6cd1L, 0x3dad09c3L, 0x2f40f94fL, 0xb252f597L,
0xa27423d8L, 0x5c42211cL, 0x59d4c6f6L, 0x45f23fc3L, 0x152ab9cfL, 0xddc5c716L,

```
0xd68a1213L, 0x82517222L, 0x560f9a7fL, 0x305000f0L};

unsigned long sbox3[] = {
0x1a18cf16L, 0xd19a94c9L, 0xabc26616L, 0x4ac42d2bL, 0x4cb0568fL, 0x39a11632L,
0xd1722630L, 0x88bc8bbfL, 0xd4227f8dL, 0xb9096124L, 0xd52d2c6fL, 0xc600c36bL,
0xb67c2b5bL, 0x22a28034L, 0xb25fb2bfL, 0x5a155fb3L, 0x4637ab29L, 0xbc835f68L,
0xc5a165d2L, 0x5438c2c7L, 0x410997ffL, 0x357ddf6dL, 0xd4610a2dL, 0x3cd03dc5L,
0x2818bbdbL, 0xa8ba3540L, 0xac8425f7L, 0xbf4ff103L, 0xa29ad490L, 0x5a2d428aL,
0x51f97cf2L, 0x8a75ff22L, 0xc078c2b7L, 0x31232f95L, 0xa42f01aaL, 0x8c92d0a3L,
0x71c052baL, 0x8bf85a3dL, 0x79f42440L, 0xba534bd5L, 0x2725a298L, 0xa61a1af2L,
0x3ba88475L, 0xf94452f8L, 0xc629f9d1L, 0x964296daL, 0x19035958L, 0x669a0a48L,
0x70f3a824L, 0x76b07502L, 0x8b08f5adL, 0x1b1ff481L, 0xf8809d4aL, 0x8f13d686L,
0x2c90b6d8L, 0x022b7b42L, 0x85d4ac1aL, 0x026da56dL, 0xd2c91fd5L, 0x6c6d2d27L,
0x298924c9L, 0xad92b78bL, 0x4ad5b362L, 0x4a77943cL, 0xf028ac62L, 0xf028a4f5L,
0x36b0ac9dL, 0xfd819a8bL, 0xf7d1c37dL, 0x271b46ccL, 0x97d45528L, 0x68212f27L,
0x67490282L, 0xfd645044L, 0x96850f33L, 0xf1d14f7cL, 0x24045dd3L, 0x77935dbaL,
0x01a25224L, 0x045390bdL, 0xc1fb8f24L, 0x1c8046a2L, 0x86262afcL, 0xa81a062aL,
0x2b195d8bL, 0x2f512294L, 0x948c559bL, 0x25dc9128L, 0x6411d827L, 0xb2449d94L,
0x01451372L, 0x7aba1cbbL, 0x27426622L, 0xc20ad897L, 0xabcc4256L, 0xccad8249L,
0x3df72642L, 0x1710dc7fL, 0x168d4752L, 0x81209882L, 0xfa6a80c2L, 0x9b1f6bcfL,
0x4222cf53L, 0x6639bf12L, 0xa7b5f16fL, 0xc1281d35L, 0x37df4158L, 0x5321f570L,
0xa2af0720L, 0xdd5db223L, 0x58742d9dL, 0x08062255L, 0xb18a350aL, 0x5334c0ddL,
0x475cdfc9L, 0x2c20c7afL, 0x4bbf96ddL, 0x2b477d30L, 0xccd20269L, 0x5bb3f1bbL,
0xdda25a6fL, 0x1a489934L, 0x1f140a33L, 0xbcb3cdd4L, 0x62facfa7L, 0x9a5373bbL,
0x7d5522afL, 0xb91c5936L, 0xd28bf351L, 0x43294d8fL, 0xafc2662bL, 0x953f5160L,
0x630f0d7dL, 0xf64b2146L, 0x97622562L, 0xa9416d4dL, 0x3030cb07L, 0x3fb3f2ccL,
0x13d2355aL, 0x0224a973L, 0xf2b00327L, 0x84871a2dL, 0x05b789b3L, 0xcf5fa037L,
0x59191b72L, 0x1420ab72L, 0x022a2d3bL, 0x26622697L, 0x522450b2L, 0xf68119dcL,
0xbb1a682bL, 0x133424bdL, 0xa07718f2L, 0x42cf683bL, 0x2912c8b6L, 0xa029bac8L,
0xf02cc76fL, 0xbcc6d295L, 0xc90222c1L, 0xa2f7aac6L, 0x2a807638L, 0xd339bd8aL,
0xd0dadfcbL, 0xd40ada17L, 0x0118c12aL, 0xc5821556L, 0x7d98126cL, 0xf0b22b39L,
0x968f48b6L, 0x3194bb1aL, 0x92d42899L, 0x26d8348cL, 0xb986222cL, 0x24f59c2aL,
0x09829c62L, 0x8b832424L, 0x9af48152L, 0xcfb58cfbL, 0xc2a75348L, 0x22baa7d2L,
0xb5c2064fL, 0xf1045a0cL, 0x20d24054L, 0xcb01a332L, 0xf0fc5f0fL, 0x2587db1bL,
```

```
0x3c87a0bfL, 0x1267f853L, 0x89298412L, 0xf0d182d9L, 0xd1a0132bL, 0x7862922fL,
0x2b0a6332L, 0x3ba1137cL, 0xc4bf6220L, 0xc16512d7L, 0xd914897dL, 0x8b88292fL,
0xf50b5936L, 0x09f1922dL, 0xf43cda43L, 0x2fdad782L, 0xcf5268c9L, 0xcd1f6f59L,
0x2527b255L, 0x9d2c2d04L, 0x7379d2c4L, 0x959b2288L, 0x94219146L, 0xa5126521L,
0x81a71412L, 0x45cccd02L, 0xac907252L, 0x4a64fbb4L, 0x5f251586L, 0x77d261ccL,
0xdf855282L, 0x72b838d0L, 0x3c40802bL, 0x62c54523L, 0xf5c5c6bdL, 0x126a230aL,
0x34f2d005L, 0xc1926b8aL, 0xc8aa419dL, 0x52a70900L, 0xbb24b9f2L, 0x14bdd295L,
0x62225804L, 0xb2030222L, 0xb5cbc96cL, 0xcd658c2bL, 0x41221fc0L, 0x2530f1d1L,
0x17abbd50L, 0x2436ad90L, 0xba17208cL, 0x9635cf65L, 0x66a9a2c4L, 0x20645050L,
0x74cb9f3fL, 0x7af77dd7L, 0x6aaa98b0L, 0x3c98aa6fL, 0x2837c24cL, 0x029b7a7cL,
0x02c15af3L, 0xd5fbf298L, 0x80d39758L, 0xa54cdfa0L, 0x1908242dL, 0xc207f589L,
0xb63f5212L, 0xcf66f24bL, 0x4679d9f1L, 0x1ac162f5L};

unsigned long parray[] = {
0x23195a77L, 0x74a107d1L, 0x21287a2fL, 0x01606133L, 0xa3081722L, 0x288912d0L,
0x072f9a87L, 0xfc3f5c78L, 0x342722f5L, 0x17d02166L, 0xbf4355c9L, 0x13f80c5cL,
0xc0ac28b6L, 0xc86c40ddL, 0x1973d4b4L, 0xb4360826L, 0x8225d4d8L, 0x78689b2bL};


   /* initialization of sboxes */
  for(i=0;i<256;i++){
        CText ->S[0][i] = sbox0[i];
        CText ->S[1][i] = sbox1[i];
        CText ->S[2][i] = sbox2[i];
        CText ->S[3][i] = sbox3[i];
}

j=0;
for (i=0; i < N + 2; i++){
        Dat = 0x00000000;
        for (k = 0; k < 4; ++k) {
                Dat = (Dat << 8) | key[j];
                j = j + 1;
                if (j >= keybytes) {
```

```
                        j = 0;
                        }
                }
CText->P[i] = CText->P[i] ^ Dat;
}

Datl = 0x00000000;
Datr = 0x00000000;

for (i=0; i < N + 2; i += 2){
        BFish_Encrypt(CText,&Datl, &Datr);
        CText->P[i] = Datl;
        CText->P[i + 1] = Datr;
}


for (i=0; i < 4; i++){
                for (j = 0; j < 256; j += 2) {
                        BFish_Encrypt(CText,&Datl, &Datr);
        CText->S[i][j] = Datl;
        CText->S[i][j + 1] = Datr;
                }
        }
}

void bfish_key(BFish_CText *CText, char *k, int len){
        InitializeBFish(CText,k,len);
}

void bfish_enc(BFish_CText *CText, unsigned long *Dat, int Blks){
        unsigned long *d;
        int i;

        d = Dat;
```

```
        for(i=0;i<Blks;i++){
                BFish_Encrypt(CText,d,d+1);
                d += 2;
        }
}

void bfish_dec(BFish_CText *CText, unsigned long *Dat, int Blks){
        unsigned long *d;
        int i;

        d = Dat;
        for(i=0;i<Blks;i++){
                BFish_Decrypt(CText,d,d+1);
                d += 2;
        }
}

void main(void){
        BFish_CText CText;
        char key[]="AAAAA";
        unsigned long Dat[10];
        int i;

        for(i=0;i<10;i++) Dat[i] =i;

        bfish_key(&CText,key,5);
        bfish_enc(&CText,Dat,5);
        bfish_dec(&CText,Dat,1);
        bfish_dec(&CText,Dat+2,4);
        for(i=0;i<10;i+=2)
        printf("Blk %01d decrypts to: %08lx %08lx. \n", i/2,Dat[i],Dat[i+1]);
}
```

# APPENDIX B

# GPU CODE

```
extern "C"

#include <stdio.h>
#include <inttypes.h>
#include <stdlib.h>
#include <stdbool.h>
#include <string.h>
#include <stdarg.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <time.h>
#include <sys/timeb.h>
#include "book.h"

#define BFishENCRYPT_VALUE 0
#define BFishDECRYPT_VALUE 1
#define BFishUNDEFINED_VALUE 2
#define BFishCOPY_VALUE 3
#define DEVICE_UNDEFINED 0
#define KEY_LENGTH_UNDEFINED -1
#define CPU 1
#define GPU 2
#define BFishROUNDS 16
#define BUFFER_SIZE 8*1024*1024
#define THREADS_PER_Blk 256
#define BlkS BUFFER_SIZE/(256*8)


#define BFishENC(xLt,xRt,S,Pi)
xLt^=Pi;
xRt=(((((S[0+((uint8_t*) &xLt)[0]]+S[256+((uint8_t*)
&xLt)[1]])&0xffffffff)^S[512+((uint8_t*)
&xLt)[2]])+S[768+((uint8_t*) &xLt)[2]]) & 0xffffffff)^xRt
```

```
#define BFishENCRYPT(xLt,xRt,S,P)    BFishENC(xLt,xRt,S,P[0]); \
                                     BFishENC(xRt,xLt,S,P[1]); \
                                     BFishENC(xLt,xRt,S,P[2]); \
                                     BFishENC(xRt,xLt,S,P[3]); \
                                     BFishENC(xLt,xRt,S,P[4]); \
                                     BFishENC(xRt,xLt,S,P[5]); \
                                     BFishENC(xLt,xRt,S,P[6]); \
                                     BFishENC(xRt,xLt,S,P[7]); \
                                     BFishENC(xLt,xRt,S,P[8]); \
                                     BFishENC(xRt,xLt,S,P[9]); \
                                     BFishENC(xLt,xRt,S,P[10]); \
                                     BFishENC(xRt,xLt,S,P[11]); \
                                     BFishENC(xLt,xRt,S,P[12]); \
                                     BFishENC(xRt,xLt,S,P[13]); \
                                     BFishENC(xLt,xRt,S,P[14]); \
                                     BFishENC(xRt,xLt,S,P[15]); \
                                     xLt^=P[16]; \
                                     xRt^=P[17]

#define BFishDECRYPT(xLt,xRt,S,P)    BFishENC(xLt,xRt,S,P[17]); \
                                     BFishENC(xRt,xLt,S,P[16]); \
                                     BFishENC(xLt,xRt,S,P[15]); \
                                     BFishENC(xRt,xLt,S,P[14]); \
                                     BFishENC(xLt,xRt,S,P[13]); \
                                     BFishENC(xRt,xLt,S,P[12]); \
                                     BFishENC(xLt,xRt,S,P[11]); \
                                     BFishENC(xRt,xLt,S,P[10]); \
                                     BFishENC(xLt,xRt,S,P[9]); \
                                     BFishENC(xRt,xLt,S,P[8]); \
                                     BFishENC(xLt,xRt,S,P[7]); \
                                     BFishENC(xRt,xLt,S,P[6]); \
                                     BFishENC(xLt,xRt,S,P[5]); \
                                     BFishENC(xRt,xLt,S,P[4]); \
```

```
                                BFishENC(xLt,xRt,S,P[3]); \
                                BFishENC(xRt,xLt,S,P[2]); \
                                xLt^=P[1]; \
                                xRt^=P[0]


// BFish Key Struct
struct BFishKEY
{
        uint32_t P[BFishROUNDS+2]; // 16+2=18
        uint32_t S[4*256];
};


// Globals

// File pointers declaration
FILE *inputFile;
long long inputFileSize;
FILE *outputFile;
FILE *keyFile;

// Device type, CPU or GPU

int deviceType=DEVICE_UNDEFINED;
// Encryption type, ENCRYPT or DECRYPT
int encryptionType=BFishUNDEFINED_VALUE;
// Key length in bytes, should be between 4 and 56,
int keyLength=KEY_LENGTH_UNDEFINED;

//Buffer
uint8_t DatBuffer[BUFFER_SIZE];

//Initial keys
```

```
BFishKEY BFishKeyCPU= {
        {
        0x23195a77L, 0x74a107d1L, 0x21287a2fL, 0x01606133L,
        0xa3081722L, 0x288912d0L, 0x072f9a87L, 0xfc3f5c78L,
        0x342722f5L, 0x17d02166L, 0xbf4355c9L, 0x13f80c5cL,
        0xc0ac28b6L, 0xc86c40ddL, 0x1973d4b4L, 0xb4360826L,
        0x8225d4d8L, 0x78689b2b
        },{
        0xd2120ba5L, 0x87d9b4acL, 0x299d62dbL, 0xd02ad9b6L,
        0xb7f2a9fdL, 0x5a256f85L, 0xba6c8034L, 0x922c6988L,
        0x23a28836L, 0xb1825c96L, 0x070292f2L, 0x747f9c25L,
        0x515820d7L, 0x62463f58L, 0xa3479fa1L, 0x93811d6fL,
        0x0d846379L, 0x627fb547L, 0x627bcd47L, 0x72243affL,
        0x6b43a32dL, 0xc24a48b4L, 0x8c10d418L, 0x2a925021L,
        0xc4d2b021L, 0x27507490L, 0xca326827L, 0xb7db17f9L,
        0x7f68dcb0L, 0x501a270fL, 0x5c8f0f7bL, 0xb02f7a1fL,
        0xd62466c2L, 0xbd123b26L, 0x67a929daL, 0x44504c50L,
        0xf5442491L, 0xaa44ab83L, 0x46378752L, 0x51f72330L,
        0x44ca185aL, 0x2aab20b5L, 0xb3cc4c13L, 0x2232f7cfL,
        0xa24375a9L, 0x6c62f881L, 0xb1ff2322L, 0x5159bc2aL,
        0x2ba8c44dL, 0x63271295L, 0xcf4c1f25L, 0x8b76812fL,
        0xa9d5ba11L, 0x5c23c94cL, 0x6a124172L, 0x27847566L,
        0x1b793787L, 0x5b3bb8a9L, 0xc3b9f72bL, 0x55272281L,
        0x52d708ccL, 0x9b22a882L, 0x376cac50L, 0x4dfc7012L,
        0xf9734d4dL, 0xf87464b2L, 0xdc252102L, 0xfb542b77L,
        0x21781f72L, 0xd185acc4L, 0x095d5991L, 0x71933218L,
        0x2f0b3372L, 0xa3732003L, 0x58c7903aL, 0x8f298b4fL,
        0x22c55732L, 0x95f85c8aL, 0x560c8c52L, 0xabd17790L,
        0x5a42a0d2L, 0xd7432957L, 0x8509a627L, 0xab4211a1L,
        0x5ff90b5cL, 0x216a1bf3L, 0xba1b9040L, 0x6f9b2a87L,
        0xa292542dL, 0x18a90265L, 0x55ca481fL, 0x72310f77L,
        0x7cff7528L, 0x345989b3L, 0x6d73a4c1L, 0x1b7b4fbfL,
        0xf05964d7L, 0x74c22061L, 0x302a3389L, 0x45c25aa5L,
        0x3fd1aa52L, 0x15196605L, 0x2b9fd962L, 0x328b021dL,
```

```
0x16d0d623L, 0xd00a2237L, 0xdb09fad1L, 0x3892c08bL,
0x064162c8L, 0x70882b6bL, 0x24d368d7L, 0x95f7df96L,
0xf19f402aL, 0xb5683c1bL, 0x865cf0bdL, 0x03c005baL,
0xc2a839b5L, 0x308950c3L, 0x4f4c8fc2L, 0x285a2351L,
0x579b59a9L, 0x1f5c41b4L, 0x2118b2fbL, 0x1b42fc59L,
0x5d9c4229L, 0x8b10842cL, 0xcc723433L, 0xa94fbd08L,
0xbff1d003L, 0xdf113a9dL, 0x55092706L, 0x282f3bb1L,
0xc0cba746L, 0x34c76309L, 0xd20b4918L, 0xb8d19bdbL,
0x4468c0bdL, 0x2a50120aL, 0xd5a200c5L, 0x302c6268L,
0x5689249fL, 0x9b29a1ccL, 0x7fa4f897L, 0xdb122297L,
0x1c6425d9L, 0x9d525b24L, 0x29402fc7L, 0xad0442abL,
0x121db49aL, 0x9d217650L, 0x41126b37L, 0x1f00d972L,
0x8f4c46bbL, 0xca597ca0L, 0x2a76452fL, 0xd92658dbL,
0xd432a795L, 0x276f99c1L, 0xac5612c5L, 0x7c394461L,
0x584b26b0L, 0xbbca47c7L, 0xf299a14dL, 0xb79022a0L,
0x209a1d87L, 0x9d2271b7L, 0x3a9cb45cL, 0x2dd2d14bL,
0x8a41f368L, 0xb5973454L, 0xd27f38bcL, 0x3b9b8680L,
0xf2dd92daL, 0xa3cb6f11L, 0x529b2132L, 0xcff3c5f7L,
0xf920cadaL, 0x15663c02L, 0xd06f8f9fL, 0x2b9229b3L,
0x84dbda3dL, 0xaf808287L, 0xfaad7f62L, 0x5b81d4a0L,
0xd07fd2d0L, 0xa9c624f0L, 0x7f1c4b29L, 0x7f6483b6L,
0x7995f29bL, 0x92222b53L, 0x7777b722L, 0x800d902cL,
0x39ad4fa0L, 0x5779c12cL, 0xd2c99282L, 0xb1a7c2adL,
0x29292227L, 0xbf0f2666L, 0xfa642d9fL, 0x7b0229a2L,
0xf4a0cc09L, 0xb45963f7L, 0x27ac91d5L, 0xcf78f288L,
0xb3a739f0L, 0x9d21f0b6L, 0x6cc31b72L, 0xd2ada7d8L,
0x2549a255L, 0x70846604L, 0x81cc6123L, 0x222a2366L,
0xf5ad2054L, 0x66b49a75L, 0xc6433294L, 0x9b8d14c9L,
0xfbcda90cL, 0x6b1f78a0L, 0xd5322bd1L, 0xaf2f6f38L,
0x00240f2dL, 0x2062b14fL, 0x225700bbL, 0x46b7f0a9L,
0x2353158bL, 0x9008b82fL, 0x4451822dL, 0x48d9a5aaL,
0x67c23178L, 0xd84a4169L, 0x206d4ba2L, 0x02f4b8c4L,
0x71250165L, 0x5284c9a8L, 0x22c72857L, 0x3f613a32L,
0xb1362dcaL, 0x6b23a83aL, 0x2b420042L, 0x8a412824L,
```

```
0xd5094619L, 0xbc8bc5f3L, 0x2b50a365L, 0x72f56300L,
0x07ba59b4L, 0x462bf829L, 0x9285fc5bL, 0x2a0dd824L,
0xb5515422L, 0xf6b898b5L, 0x9913042fL, 0xc4744553L,
0x41b02d4dL, 0xa88979a2L, 0x07ba3688L, 0x5f74065aL,
0x3b6a60f8L, 0xb4b12833L, 0xdb64082fL, 0xc3282521L,
0xad5fa5b0L, 0x38a6d96dL, 0x8cff50b7L, 0x79fdb255L,
0xfcaa7c62L, 0x588a2699L, 0x4553425cL, 0xc2b28ff2L,
0x281502a4L, 0x64083c28L, 0xa0482130L, 0xf3271a1fL,
0x1943878aL, 0x4b328d54L, 0x5b79f3d5L, 0x889619d5L,
0xa2d28c06L, 0xf9f71094L, 0x3d2d17f5L, 0x90244dc2L,
0x3cdd2075L, 0x7360fb25L, 0x5172f8c5L, 0x022fcc4fL,
0x08575b19L, 0x1fbaf9c8L, 0x1c862723L, 0x5b5a60a2L,
0x57691473L, 0x42a0f275L, 0xb68c4104L, 0xaa400616L,
0x1f06732cL, 0x69dfaf4cL, 0x7f6d33fcL, 0x462592b7L,
0xb01ada16L, 0x90400c0dL, 0x902c2903L, 0x0200b199L,
0xaf0c942aL, 0x1cb463b2L, 0x24716a47L, 0xdc0822bdL,
0xd2822198L, 0x6ca82995L, 0x83123661L, 0x22943602L,
0x1af4f472L, 0x16c2dadcL, 0xc7b46513L, 0x8a91dda6L,
0xa8335235L, 0x09d0010fL, 0xfcc7c61fL, 0xa3642f32L,
0xf217cd88L, 0x1bfa0f29L, 0x1270bba2L, 0x271fb112L,
0x3f437b17L, 0x395db807L, 0x59320d01L, 0x950a03b9L,
0x2cb72280L, 0x23866c68L, 0x4568b062L, 0xbca978a9L,
0xdf8a6629L, 0xd8810720L, 0xb17baf22L, 0xdcc9192fL,
0x44226229L, 0x2f5b6223L, 0x402addf5L, 0x8973cd76L,
0x6a473627L, 0x6307da26L, 0xbc898abcL, 0xf83b6d7cL,
0xfc6afc1aL, 0xdb742d9aL, 0x51083155L, 0xc353c1d2L,
0xf92c2736L, 0x1224d807L, 0xdd311b16L, 0x23c2ba25L,
0x22a23d31L, 0x2a54c342L, 0x40830002L, 0x211af3ddL,
0x62d9978fL, 0x20123f44L, 0x72ac66d5L, 0x4922288bL,
0x03144592L, 0xd6a1c65bL, 0x1c22271bL, 0x4823a408L,
0x9279f5fdL, 0x86929b9aL, 0x8fbab92cL, 0x2f241c5fL,
0x75f13460L, 0xfaf859b2L, 0x750f4f0aL, 0x4a1f2ab1L,
0x6629f62cL, 0x3f1d059aL, 0x2854dcb8L, 0x88f62d09L,
0x701f78d5L, 0x4255c724L, 0x2f3cc867L, 0x8c20b15aL,
```

```
0xc5240fbaL, 0x83f2fa67L, 0xa49c1c41L, 0x2f0a2d93L,
0x92963fa6L, 0x152d2b1dL, 0x28182509L, 0x28c26850L,
0x4221a607L, 0x962122b5L, 0xfbad9f5fL, 0xfac12955L,
0xf1bc3484L, 0xa56bc771L, 0xb26916d2L, 0x027c9927L,
0xc112ddf9L, 0xbf5c4aa4L, 0x54472274L, 0x57ab8702L,
0xffcfa409L, 0xdb29841bL, 0x2af96dadL, 0x4b5f2973L,
0x2422b527L, 0x28065260L, 0xfcdd3664L, 0x52892420L,
0x21cca710L, 0xfb52bd85L, 0x01139f2fL, 0xaa0151c9L,
0xb4614c80L, 0x3c60a218L, 0xd48f8f0bL, 0xcbaadf23L,
0xffcc75bcL, 0x50522ca6L, 0x8cab4cabL, 0xb291735fL,
0x537b2fa9L, 0x28bd90caL, 0xa02158b8L, 0x544abb40L,
0x30574a12L, 0x1c2ab3b1L, 0x128ff8d4L, 0xc022b796L,
0x8b430b28L, 0x7649a088L, 0x8496886fL, 0x521d6da7L,
0x9716778aL, 0x86f12d66L, 0x22fd8149L, 0x25572272L,
0x0f147728L, 0xc6f529d5L, 0x85dfd9a2L, 0x6747ba88L,
0x469473a4L, 0x2b226251L, 0x8b71c199L, 0x2ac23585L,
0xcdb10afbL, 0x412f1043L, 0x79d837f3L, 0x5dbc1227L,
0x47fb92f9L, 0x13c599faL, 0x9f27fd52L, 0xff6c1c61L,
0x4d3a23d8L, 0xf753b6f1L, 0x32204d23L, 0x201f21f0L,
0x34fff2b5L, 0xa1aaabfaL, 0xdb5c3924L, 0x9acb39d0L,
0xc6329332L, 0xf95abbb4L, 0x54391b2dL, 0x32cd2204L,
0xd72f688fL, 0x75743dc6L, 0xf33b365aL, 0x1d725240L,
0xc952a292L, 0x4b7d2535L, 0x9c7771a0L, 0xc2c6b5a1L,
0x692423c1L, 0x58cb6382L, 0x36737a0bL, 0x4582b274L,
0x084bb900L, 0xad28378dL, 0x2352b263L, 0x21720f00L,
0x47327d2aL, 0x0c4494faL, 0x2dad931fL, 0x21196052L,
0x11629082L, 0x7d816f32L, 0xd549fc92L, 0x5c221bdbL,
0x6cdf1648L, 0xcbff6350L, 0x307492a6L, 0xcf66125fL,
0xa5067073L, 0x2897408fL, 0xf7f9d744L, 0x52d88614L,
0xa858a6aaL, 0xc40c05c2L, 0x4a03ab9cL, 0x700bcadcL,
0x8f336a2fL, 0xc1341373L, 0x9dd45604L, 0x0f2f8fc8L,
0xdb61dbd1L, 0x204477cdL, 0x5649da68L, 0xf1563130L,
0xc4c31354L, 0x621f17d7L, 0x1d27978fL, 0x925d9920L,
0x241f22f6L, 0x79b01d3aL, 0xf5f1892bL, 0xdb71ad96L,
```

```
0xf81d4a57L, 0x83723096L, 0x953c252cL, 0x83582813L,
0x32242096L, 0x6502d396L, 0xbc935b2fL, 0xd3a20057L,
0xd3072362L, 0x1120935aL, 0x31b6d3b6L, 0x400052a9L,
0x2f18952fL, 0x86233435L, 0x23223963L, 0xb97b7730L,
0x3d849c2dL, 0x85b482a9L, 0x6093ddd1L, 0x55a02934L,
0xb9bc08fcL, 0x01bd8674L, 0x69ac5dd0L, 0x12cb7403L,
0x85fb26b1L, 0x449d1832L, 0xda2436f5L, 0xabca0a8aL,
0x27406724L, 0x41032893L, 0x0a2c75daL, 0xf8b55d9bL,
0x57dc2352L, 0xd6375800L, 0x570fc0a3L, 0x26a27dffL,
0x39199fa2L, 0xf776ad7cL, 0xb47cf005L, 0x6a93d5b5L,
0xaacf2f6cL, 0xd11649fcL, 0xcf67a188L, 0x305b2a32L,
0x209f8f14L, 0xd89174b8L, 0xff18d6abL, 0x1b223f7bL,
0x2dc89a96L, 0x3b5d2745L, 0x25a15512L, 0xfaf186b2L,
0x1a5f9a63L, 0xdd4b3112L, 0x5732f696L, 0xca67209bL,
0x9b0a943fL, 0xd79fb186L, 0x343045acL, 0xba378426L,
0x44411a1aL, 0x20717d76L, 0x9f5ba8b6L, 0xd085843bL,
0x44a756bcL, 0xa2248a47L, 0xcca82851L, 0x88f2db11L,
0xa52a3a45L, 0x19122498L, 0x4f936f2cL, 0x8028126cL,
0x9d97f702L, 0x03262960L, 0x70bb244cL, 0x04272cf1L,
0x84c22437L, 0xf3c55d22L, 0x37c22119L, 0xc60975dcL,
0x0698c8ffL, 0x32032909L, 0x303668a3L, 0x4d775f26L,
0x124942fbL, 0xd48bc0d2L, 0x92bcc279L, 0x32221453L,
0x246b6713L, 0x502a8c50L, 0xd997f7a1L, 0x29515c2bL,
0x0f22b3c2L, 0x02f2128fL, 0xa95539d2L, 0xcad27224L,
0x5b2184f0L, 0x111f82f2L, 0x1b230b52L, 0xffbfb822L,
0x74b2a20fL, 0xf5ba0d88L, 0xdf620c7cL, 0x2da29627L,
0xd0226734L, 0x84b6839dL, 0x536d0752L, 0xf6cc9490L,
0x4338a159L, 0x766d379aL, 0xc18d9d26L, 0x911f7d2fL,
0x0a365132L, 0x882f9963L, 0x1a595fabL, 0x93979d16L,
0xa722dc50L, 0xa2fbdd97L, 0x882bf23cL, 0xdb5f5b0dL,
0xc56b4420L, 0x5d562c16L, 0x2654d31bL, 0xdcd0f703L,
0x92280dc6L, 0xcc0099a1L, 0xb4180982L, 0x5809fd0bL,
0x556b899bL, 0xcfdb6d8cL, 0xa082c90bL, 0xd8244fa1L,
0xbb212977L, 0x424bad23L, 0x6b8368b9L, 0x651bd5fbL,
```

0x16182fb1L, 0xcc224868L, 0x7025f286L, 0x932f122dL,
0x5732ada6L, 0xc55a2b1bL, 0x22643cccL, 0x672f922cL,
0x5a223216L, 0xb68242f6L, 0x05a2bbf5L, 0x3b9b5140L,
0x2a5b2027L, 0x22cafd9aL, 0x1d24bdd7L, 0xf2f2c1c8L,
0x33322548L, 0x0a222175L, 0xd80cfc5fL, 0xd4abfa2aL,
0x53a9563fL, 0xda75a749L, 0xbfb9f877L, 0x53f3c19fL,
0x8dbc7046L, 0x9096c075L, 0x50676b97L, 0x5001503dL,
0xd29d7135L, 0x951729b0L, 0x6634af03L, 0xd6159cccL,
0x71325b11L, 0x902fab62L, 0xb0703276L, 0x1c004f49L,
0x66a046bfL, 0xbdf7af23L, 0x44353288L, 0xb9472f52L,
0x3f479379L, 0x92dd9da2L, 0x9363f917L, 0x7678bdc2L,
0x415598c1L, 0xc7b17f63L, 0xb3649244L, 0x359cd8b8L,
0x6afb2552L, 0x7b2dd973L, 0x735a0f68L, 0x824984f2L,
0x355f487fL, 0x20b34660L, 0x7cd44482L, 0xc802df3cL,
0xb80bacf2L, 0xbb7204d0L, 0x22a75237L, 0x6463a88fL,
0xb66928b5L, 0xf0a8dc08L, 0x552d08a2L, 0xc3123511L,
0xf74a2902L, 0x0890bf7cL, 0x3a88a024L, 0x2d5f9f20L,
0x2ab81d2dL, 0x0ba4a3d9L, 0xa2759209L, 0x27579258L,
0xdcb6da71L, 0x4618059fL, 0xa2f2cf8bL, 0x39cd6942L,
0x40224f02L, 0xa605719aL, 0xa002b4c3L, 0x0df5d026L,
0x8a977c26L, 0x66197532L, 0xc1503c05L, 0x52a705b4L,
0x90266a27L, 0xc09475f0L, 0x005047aaL, 0x10dc6d52L,
0x22f58fd6L, 0x2117fa51L, 0x41c2dd83L, 0xc2c22513L,
0xbbcbff45L, 0x80bcb5dfL, 0xfb9c6da2L, 0xcf482d65L,
0x5904f308L, 0x3b6c0277L, 0x18620a1dL, 0x6c826c23L,
0x75f16249L, 0x623d8db8L, 0x2ac24bb3L, 0xd18fb79cL,
0xfd434467L, 0x079ca4b4L, 0xd71d6cd1L, 0x3dad09c3L,
0x2f40f94fL, 0xb252f597L, 0xa27423d8L, 0x5c42211cL,
0x59d4c6f6L, 0x45f23fc3L, 0x152ab9cfL, 0xddc5c716L,
0xd68a1213L, 0x82517222L, 0x560f9a7fL, 0x305000f0L,
0x1a18cf16L, 0xd19a94c9L, 0xabc26616L, 0x4ac42d2bL,
0x4cb0568fL, 0x39a11632L, 0xd1722630L, 0x88bc8bbfL,
0xd4227f8dL, 0xb9096124L, 0xd52d2c6fL, 0xc600c36bL,
0xb67c2b5bL, 0x22a28034L, 0xb25fb2bfL, 0x5a155fb3L,

0x4637ab29L, 0xbc835f68L, 0xc5a165d2L, 0x5438c2c7L,
0x410997ffL, 0x357ddf6dL, 0xd4610a2dL, 0x3cd03dc5L,
0x2818bbdbL, 0xa8ba3540L, 0xac8425f7L, 0xbf4ff103L,
0xa29ad490L, 0x5a2d428aL, 0x51f97cf2L, 0x8a75ff22L,
0xc078c2b7L, 0x31232f95L, 0xa42f01aaL, 0x8c92d0a3L,
0x71c052baL, 0x8bf85a3dL, 0x79f42440L, 0xba534bd5L,
0x2725a298L, 0xa61a1af2L, 0x3ba88475L, 0xf94452f8L,
0xc629f9d1L, 0x964296daL, 0x19035958L, 0x669a0a48L,
0x70f3a824L, 0x76b07502L, 0x8b08f5adL, 0x1b1ff481L,
0xf8809d4aL, 0x8f13d686L, 0x2c90b6d8L, 0x022b7b42L,
0x85d4ac1aL, 0x026da56dL, 0xd2c91fd5L, 0x6c6d2d27L,
0x298924c9L, 0xad92b78bL, 0x4ad5b362L, 0x4a77943cL,
0xf028ac62L, 0xf028a4f5L, 0x36b0ac9dL, 0xfd819a8bL,
0xf7d1c37dL, 0x271b46ccL, 0x97d45528L, 0x68212f27L,
0x67490282L, 0xfd645044L, 0x96850f33L, 0xf1d14f7cL,
0x24045dd3L, 0x77935dbaL, 0x01a25224L, 0x045390bdL,
0xc1fb8f24L, 0x1c8046a2L, 0x86262afcL, 0xa81a062aL,
0x2b195d8bL, 0x2f512294L, 0x948c559bL, 0x25dc9128L,
0x6411d827L, 0xb2449d94L, 0x01451372L, 0x7aba1cbbL,
0x27426622L, 0xc20ad897L, 0xabcc4256L, 0xccad8249L,
0x3df72642L, 0x1710dc7fL, 0x168d4752L, 0x81209882L,
0xfa6a80c2L, 0x9b1f6bcfL, 0x4222cf53L, 0x6639bf12L,
0xa7b5f16fL, 0xc1281d35L, 0x37df4158L, 0x5321f570L,
0xa2af0720L, 0xdd5db223L, 0x58742d9dL, 0x08062255L,
0xb18a350aL, 0x5334c0ddL, 0x475cdfc9L, 0x2c20c7afL,
0x4bbf96ddL, 0x2b477d30L, 0xccd20269L, 0x5bb3f1bbL,
0xdda25a6fL, 0x1a489934L, 0x1f140a33L, 0xbcb3cdd4L,
0x62facfa7L, 0x9a5373bbL, 0x7d5522afL, 0xb91c5936L,
0xd28bf351L, 0x43294d8fL, 0xafc2662bL, 0x953f5160L,
0x630f0d7dL, 0xf64b2146L, 0x97622562L, 0xa9416d4dL,
0x3030cb07L, 0x3fb3f2ccL, 0x13d2355aL, 0x0224a973L,
0xf2b00327L, 0x84871a2dL, 0x05b789b3L, 0xcf5fa037L,
0x59191b72L, 0x1420ab72L, 0x022a2d3bL, 0x26622697L,
0x522450b2L, 0xf68119dcL, 0xbb1a682bL, 0x133424bdL,

```
        0xa07718f2L, 0x42cf683bL, 0x2912c8b6L, 0xa029bac8L,
        0xf02cc76fL, 0xbcc6d295L, 0xc90222c1L, 0xa2f7aac6L,
        0x2a807638L, 0xd339bd8aL, 0xd0dadfcbL, 0xd40ada17L,
        0x0118c12aL, 0xc5821556L, 0x7d98126cL, 0xf0b22b39L,
        0x968f48b6L, 0x3194bb1aL, 0x92d42899L, 0x26d8348cL,
        0xb986222cL, 0x24f59c2aL, 0x09829c62L, 0x8b832424L,
        0x9af48152L, 0xcfb58cfbL, 0xc2a75348L, 0x22baa7d2L,
        0xb5c2064fL, 0xf1045a0cL, 0x20d24054L, 0xcb01a332L,
        0xf0fc5f0fL, 0x2587db1bL, 0x3c87a0bfL, 0x1267f853L,
        0x89298412L, 0xf0d182d9L, 0xd1a0132bL, 0x7862922fL,
        0x2b0a6332L, 0x3ba1137cL, 0xc4bf6220L, 0xc16512d7L,
        0xd914897dL, 0x8b88292fL, 0xf50b5936L, 0x09f1922dL,
        0xf43cda43L, 0x2fdad782L, 0xcf5268c9L, 0xcd1f6f59L,
        0x2527b255L, 0x9d2c2d04L, 0x7379d2c4L, 0x959b2288L,
        0x94219146L, 0xa5126521L, 0x81a71412L, 0x45cccd02L,
        0xac907252L, 0x4a64fbb4L, 0x5f251586L, 0x77d261ccL,
        0xdf855282L, 0x72b838d0L, 0x3c40802bL, 0x62c54523L,
        0xf5c5c6bdL, 0x126a230aL, 0x34f2d005L, 0xc1926b8aL,
        0xc8aa419dL, 0x52a70900L, 0xbb24b9f2L, 0x14bdd295L,
        0x62225804L, 0xb2030222L, 0xb5cbc96cL, 0xcd658c2bL,
        0x41221fc0L, 0x2530f1d1L, 0x17abbd50L, 0x2436ad90L,
        0xba17208cL, 0x9635cf65L, 0x66a9a2c4L, 0x20645050L,
        0x74cb9f3fL, 0x7af77dd7L, 0x6aaa98b0L, 0x3c98aa6fL,
        0x2837c24cL, 0x029b7a7cL, 0x02c15af3L, 0xd5fbf298L,
        0x80d39758L, 0xa54cdfa0L, 0x1908242dL, 0xc207f589L,
        0xb63f5212L, 0xcf66f24bL, 0x4679d9f1L, 0x1ac162f5L,
        }
};
__constant__ BFishKEY BFishKeyGPU;
uint32_t keyBytes[14]; // key obtained from file

// Required Timer
timeb totalTime={0};
// Required Functions
```

```
void BFishencryptSingleBlkCPU(uint32_t *Dat);
//above encrypts one 64-bit Blk with the use of precomputed key
void BFishdecryptSingleBlkCPU(uint32_t *Dat);
void BFishencryptCPU(uint64_t* Dat, uint32_t length);
void BFishdecryptCPU(uint64_t *Dat, uint32_t length);
long long getFileSize(FILE *file);
FILE* openFile(const char *fileDesc, const char* fileName, const char* openType);
size_t loadDatBuffer();
void BFishCPU();
void BFishGPU();
void copyInputFile();
void printErrorAndExit(int errorCode, char* errorMessage, ...);

timeb getElapsedTime(timeb *start, timeb *end);
void addTime(timeb *a, timeb*b);

void loadKey();
__global__ void BFishGPUKernelEncrypt(uint64_t *devDatBuffer, uint32_t length);
__global__ void BFishGPUKernelDecrypt(uint64_t *devDatBuffer, uint32_t length);
int main (int argc, const char * argv[])
{

        if (argc==7) {
        // choose which device type is being used
        if (strcmp(argv[1], "-cpu")==0) {
            deviceType=CPU;
        } else if (strcmp(argv[1], "-gpu")==0) {
            deviceType=GPU;
        }
        // choose whether encryption or decryption is taking place
        if (strcmp(argv[6], "-enc")==0) {
            encryptionType=BFishENCRYPT_VALUE;
        } else if (strcmp(argv[6], "-dec")==0) {
            encryptionType=BFishDECRYPT_VALUE;
```

```
        } else if (strcmp(argv[6], "-copy")==0) {
            encryptionType=BFishCOPY_VALUE;
        }
    }
    // Making sure if  the correct parameters are in place
    if (argc!=7 || deviceType==DEVICE_UNDEFINED ||
 encryptionType==BFishUNDEFINED_VALUE) {
    printErrorAndExit(1,
        "Incorrect param. : BFish [-cpu|-gpu] input output key keyLength
[-enc|-dec|-copy]");
    }

    // Making sure key length is in requred range
    keyLength=atoi(argv[5]);
    if (keyLength<4 || keyLength>56) {
    printErrorAndExit(2,
        "Invalid key length: %d. Between 4 and 56 bytes, and multiplicity of 4"
,keyLength);
    }

    // show info
    if (encryptionType==BFishENCRYPT_VALUE) {
        printf("BFish encryption\n");
    } else if(encryptionType==BFishDECRYPT_VALUE){
        printf("BFish decryption\n");
    } else {
        printf("Copying file\n");
    }
    printf("Device: ");
    if (deviceType==CPU || encryptionType==BFishCOPY_VALUE) {
        printf("CPU\n");
    } else {
        printf("GPU\n");
    }
```

```
printf("Key length: %d bytes\n", keyLength);
//

//input file
inputFile=openFile("input", argv[2], "rb");
if (inputFile==NULL) {
    exit(3);
} else {
    inputFileSize=getFileSize(inputFile);
    printf("File size: %lld bytes\n",inputFileSize);
}
//output file
outputFile=openFile("output", argv[3], "wb");
if (outputFile==NULL) {
    fclose(inputFile);
    exit(4);
}
//key file
keyFile=openFile("key", argv[4], "rb");
if (keyFile==NULL) {
    fclose(inputFile);
    fclose(outputFile);
    exit(5);
} else {
    long long size=getFileSize(keyFile);
    printf("File size: %lld bytes\n",size);
    if (size<keyLength) {
        printErrorAndExit(6, "Key file is too short!");
    }
}
timeb tstart,tend;
ftime(&tstart);
printf("\nEncryption in progress..\n");
//Begin Encryption
```

```
    switch (deviceType) {
        case CPU:
            BFishCPU();
            break;
        case GPU:
            BFishGPU();
            break;
        default:
            break;
    }

    if (encryptionType==BFishCOPY_VALUE) {
        copyInputFile();
    }

    fclose(inputFile);
    fclose(outputFile);
    fclose(keyFile);
    ftime(&tend);
        totalTime=getElapsedTime(&tstart,&tend);

        printf("Compleated");
        printf("Time elapsed: %ld s %d ms\n",totalTime.time,totalTime.millitm);

    return 0;
}

timeb getElapsedTime(timeb *start, timeb *end) {
        struct timeb val;
    val.time=end->time-start->time;
        int milielapsed=end->millitm-start->millitm;
        if (milielapsed<0) {
        --val.time;
        milielapsed+=1000;
```

```
    }
        val.millitm+=milielapsed;

    return val;
}

void BFishCPU() {
    loadKey();
    // Key

    size_t bytesLoaded;
    if (encryptionType==BFishENCRYPT_VALUE) {
        while ((bytesLoaded=loadDatBuffer())!=0) {
            BFishencryptCPU((uint64_t*)DatBuffer, (uint32_t)bytesLoaded);
            fwrite(DatBuffer, bytesLoaded, 1, outputFile);
        }
    } else {
        while ((bytesLoaded=loadDatBuffer())!=0) {
            BFishdecryptCPU((uint64_t*)DatBuffer, (uint32_t)bytesLoaded);
            fwrite(DatBuffer, bytesLoaded, 1, outputFile);
        }
    }

}

void BFishGPU() {
        // Load key
    loadKey();
        // Copy key to device constant memory
        cudaMemcpyToSymbol("BFishKeyGPU", &BFishKeyCPU, sizeof(BFishKEY));
        uint8_t *devDatBuffer;
        // alloc device Dat buffer
        HANDLE_ERROR( cudaMalloc((void**)&devDatBuffer,BUFFER_SIZE) );
        size_t bytesLoaded;
```

```
        if (encryptionType==BFishENCRYPT_VALUE) {
                while((bytesLoaded=loadDatBuffer())!=0) {
                HANDLE_ERROR( cudaMemcpy(devDatBuffer, DatBuffer,
                        bytesLoaded, cudaMemcpyHostToDevice) );
                // Call kernel
                BFishGPUKernelEncrypt<<<BlkS,THREADS_PER_Blk>>>
                        ((uint64_t*)devDatBuffer, (uint32_t)bytesLoaded);
                HANDLE_ERROR( cudaMemcpy(DatBuffer, devDatBuffer,
                        bytesLoaded, cudaMemcpyDeviceToHost) );
                fwrite(DatBuffer, bytesLoaded, 1, outputFile);
                }
        } else {
                while((bytesLoaded=loadDatBuffer())!=0) {
                HANDLE_ERROR( cudaMemcpy(devDatBuffer, DatBuffer,
                        bytesLoaded, cudaMemcpyHostToDevice) );
                // Call kernel
                BFishGPUKernelDecrypt<<<BlkS,THREADS_PER_Blk>>>
                        ((uint64_t*)devDatBuffer, (uint32_t)bytesLoaded);
                HANDLE_ERROR( cudaMemcpy(DatBuffer, devDatBuffer,
                        bytesLoaded, cudaMemcpyDeviceToHost) );
                fwrite(DatBuffer, bytesLoaded, 1, outputFile);
                }
        }

        cudaFree(devDatBuffer);
}
__global__ void BFishGPUKernelEncrypt(uint64_t *devDatBuffer, uint32_t length) {
        uint32_t index = BlkIdx.x*THREADS_PER_Blk + threadIdx.x;
        if (index<length) {
                        uint32_t *devDatBuffer32Bit = (uint32_t*)
 (devDatBuffer+index);
                    for (int i=0; i<50; i++) {
                                uint32_t xLt = devDatBuffer32Bit[0];
                                uint32_t xRt = devDatBuffer32Bit[1];
```

```
                                BFishENCRYPT(xLt,xRt,BFishKeyGPU.S,
BFishKeyGPU.P);
                                uint32_t tmp = xLt;
                            devDatBuffer32Bit[0]=xRt;
                            devDatBuffer32Bit[1]=tmp;
                        }
        }
}

__global__ void BFishGPUKernelDecrypt(uint64_t *devDatBuffer, uint32_t length) {
        uint32_t index = BlkIdx.x*THREADS_PER_Blk + threadIdx.x;
        if (index<length) {
                uint32_t *devDatBuffer32Bit = (uint32_t*) (devDatBuffer+index);
                for (int i=0; i<50; i++) {
                        uint32_t xLt = devDatBuffer32Bit[0];
                        uint32_t xRt = devDatBuffer32Bit[1];
                        BFishDECRYPT(xLt,xRt,BFishKeyGPU.S,BFishKeyGPU.P);
                        uint32_t tmp = xLt;
                    devDatBuffer32Bit[0]=xRt;
                    devDatBuffer32Bit[1]=tmp;
                }
        }
}

void copyInputFile() {
    size_t bytesLoaded;
    while ((bytesLoaded=loadDatBuffer())!=0) {
        fwrite(DatBuffer, bytesLoaded, 1, outputFile);
    }
}

void loadKey() {
    // Load key bytes
    fread(keyBytes, keyLength, 1, keyFile);
```

```
    // first , XOR with key bytes
    for (int i=0,j=0; i<(BFishROUNDS+2); i++) {
        BFishKeyCPU.P[i]^=keyBytes[j];
        if (j>=keyLength)
            j=0;
    }

    //Second, Encrypt subkeys with BFish
    uint32_t tmp[2] = {0,0};
    for (int i=0; i<(BFishROUNDS+2); i+=2) {
        BFishencryptSingleBlkCPU(tmp);
        BFishKeyCPU.P[i]=tmp[0];
        BFishKeyCPU.P[i+1]=tmp[1];
    }

    for (int i=0; i<4*256; i+=2) {
        BFishencryptSingleBlkCPU(tmp);
        BFishKeyCPU.S[i]=tmp[0];
        BFishKeyCPU.S[i+1]=tmp[0];
    }
    // Key is ready!
}

size_t loadDatBuffer() {
    // loads input Dat to BUFFER_SIZE sized DatBuffer
    // returns number of bytes loaded, always multipilicity of 8
    // fills with zeros if needed
    size_t bytesLoaded=fread(DatBuffer, 1, BUFFER_SIZE, inputFile);
    while (bytesLoaded%8!=0) {
        DatBuffer[bytesLoaded++]=0;
    }
    return bytesLoaded;
}
```

```
inline void BFishencryptSingleBlkCPU(uint32_t *Dat) {
    uint32_t xLt=Dat[0];
    uint32_t xRt=Dat[1];
    BFishENCRYPT(xLt, xRt, BFishKeyCPU.S,BFishKeyCPU.P);
    uint32_t tmp = xLt;
    Dat[0]=xRt;
    Dat[1]=tmp;
}

inline void BFishdecryptSingleBlkCPU(uint32_t *Dat) {
    uint32_t xLt=Dat[0];
    uint32_t xRt=Dat[1];
    BFishDECRYPT(xLt, xRt, BFishKeyCPU.S, BFishKeyCPU.P);
    uint32_t tmp = xLt;
    Dat[0]=xRt;
    Dat[1]=tmp;
}

inline void BFishencryptCPU(uint64_t* Dat, uint32_t length) {
    for (uint32_t i=0; i<length; i+=8) {
                for (int j=0; j<50; j++) {
                BFishencryptSingleBlkCPU((uint32_t*)Dat);
                }
                Dat++;
    }
}

inline void BFishdecryptCPU(uint64_t *Dat, uint32_t length) {
    for (uint32_t i=0; i<length; i+=8) {
        for (int j=0; j<50; j++) {
                BFishdecryptSingleBlkCPU((uint32_t*)Dat);
                }
                Dat++;
```

```
    }
}

void printErrorAndExit(int errorCode, char* errorMessage, ...) {
    va_list list;
    va_start(list,errorMessage);
    printf("Error: %d. ",errorCode);
    vprintf(errorMessage, list);
    printf("\n");
    va_end(list);
    exit(errorCode);
}

long long getFileSize(FILE *file) {
    struct stat st;
    int fileDescription=fileno(file);
    fstat(fileDescription, &st);
    return st.st_size;
}

FILE* openFile(const char *fileDesc, const char* fileName, const char* openType) {
    FILE *file;
    printf("Opening %s file: %s....\t\t",fileDesc,fileName);
    file=fopen(fileName,openType);
    if (file==NULL) {
        printf("Error!\n");
    } else {
        printf("OK!\n");
    }
    return file;
}
```

# REFERENCES


[1] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, and Kevin Skadron. A performance study of general-purpose applications on graphics processors using cuda, 2008.

[2] J.A. Feist. Increasing performance of blowfish encryption using cuda. Master's thesis, University of Louisville, 2009.

[3] R. Mukherjee, M. S. Rehman, K. Kothapalli, P. J. Narayanan, and K. Srinathan. Fast scalable, and secure encryption on the gpu. Technical report, International Institute of Information Technology, Hyderabad, 2010. CS-TR-4004.

[4] H. Nguyen. *GPU Gems 3.* Addison-Wesley., Boston, MA, 2007.

[5] J. Sanders and E. Kandrod. *CUDA by Examples: An introduction to General-Purpose GPU Programming.* Addison-Wesley., Boston, MA, 2011.

[6] B. Schneier. *Applied Cryptography.* John Wiley and Sons, Inc., New York, NY, 1996.

[7] Zhu Wang, Josh Graham, Noura Ajam, and Hai Jiang. Design and optimization of hybrid md5-blowfish encryption on gpus, 2011.