High Performance ARIA Block Cipher Implementation on Graphics Processing Unit

1. Abstract

It has been shown by many researches that the running time of encryption, decryption algorithms on GPU are significantly faster compared to their CPU implementation given a large data. As the data size to be encrypted or decrypted gets larger, GPU becomes a more suitable environment to process it in parallel. It is possible to find many GPU implementations of popular block cipher algorithms such as DES, AES which are idealized to run efficiently on GPU.

This paper proposes different GPU implementations of a less popular block cipher ARIA. ARIA is a block cipher designed in 2003 by a group of South Korean researchers. In 2004, it is selected as a standard cryptographic technique. The encryption algorithm is implemented in Electronic Codebook (ECB) mode using two methods and various memory types in GPU. The decryption algorithm is provided both in ECB and CBC mode.

2. Motivation & Significance

It is important to encrypt and decrypt data in network end systems in an efficient manner where data to be sent is large and must be processed fast enough to minimize the delay. To achieve the minimum delay and maximum efficiency in such cases, the computation is done on GPU. Many studies have been done on implementing block ciphers such as AES, DES, Blowfish on GPU. Studies have shown significant speed-up compared to their CPU implementation. However there is less research about the parallel implementation of ARIA. This paper aims to provide an efficient implementation of the ARIA Block Cipher on GPU, making use of NVIDIA GPU's parallel computing engine CUDA.

3. Prior Work & Limitations

Only the research in [1] discusses the design of the ARIA encryption algorithm using CUDA. The paper discusses several methods and for each method uses several memory allocation schemes. However only ECB mode encryption designs are discussed and compared using one key-size. In this paper ECB mode decryption and CBC mode decryption implementations and comparisons are provided. Also different key size performance comparisons are shown.

4. Algorithm Description

The algorithm uses a substitution-permutation network which a series of linked mathematical operations used in block cipher algorithms. A substitution-network takes a block of plaintext as input and applies a number of alternating rounds. The rounds are chained and each round's output is the input of the consecutive round. Each round has three steps:

1. XOR with the round key

This step simply takes the input to the round and performs a bitwise *XOR* with the round key. The output of this step is the input to the Substitution Layer. Both round key and the text block are 16 bytes.

2. Substitution Layer

Each byte of the text is used as indices to one of the look-up tables. The value of the table at the corresponding index is the output value of the byte. There exists two look-up tables used for encryption named *SB1* and *SB2*. Their inverses are *SB3* and *SB4* are used in decryption process. The substitution layer functions are denoted *SL1* and *SL2* for odd and even rounds respectively.

SL1 computes the following for each consecutive 4 bytes (*x0,x1,x2,x3*) of the 16 byte text block:

y0 = SB1(x0), y1 = SB2(x1), y2 = SB3(x2), y3 = SB4(x3)

SL2 computes the following for each consecutive 4 bytes (*x0,x1,x2,x3*) of the 16 byte text block:

y0 = SB3(x0), y1 = SB4(x1), y2 = SB1(x2), y3 = SB2(x3)

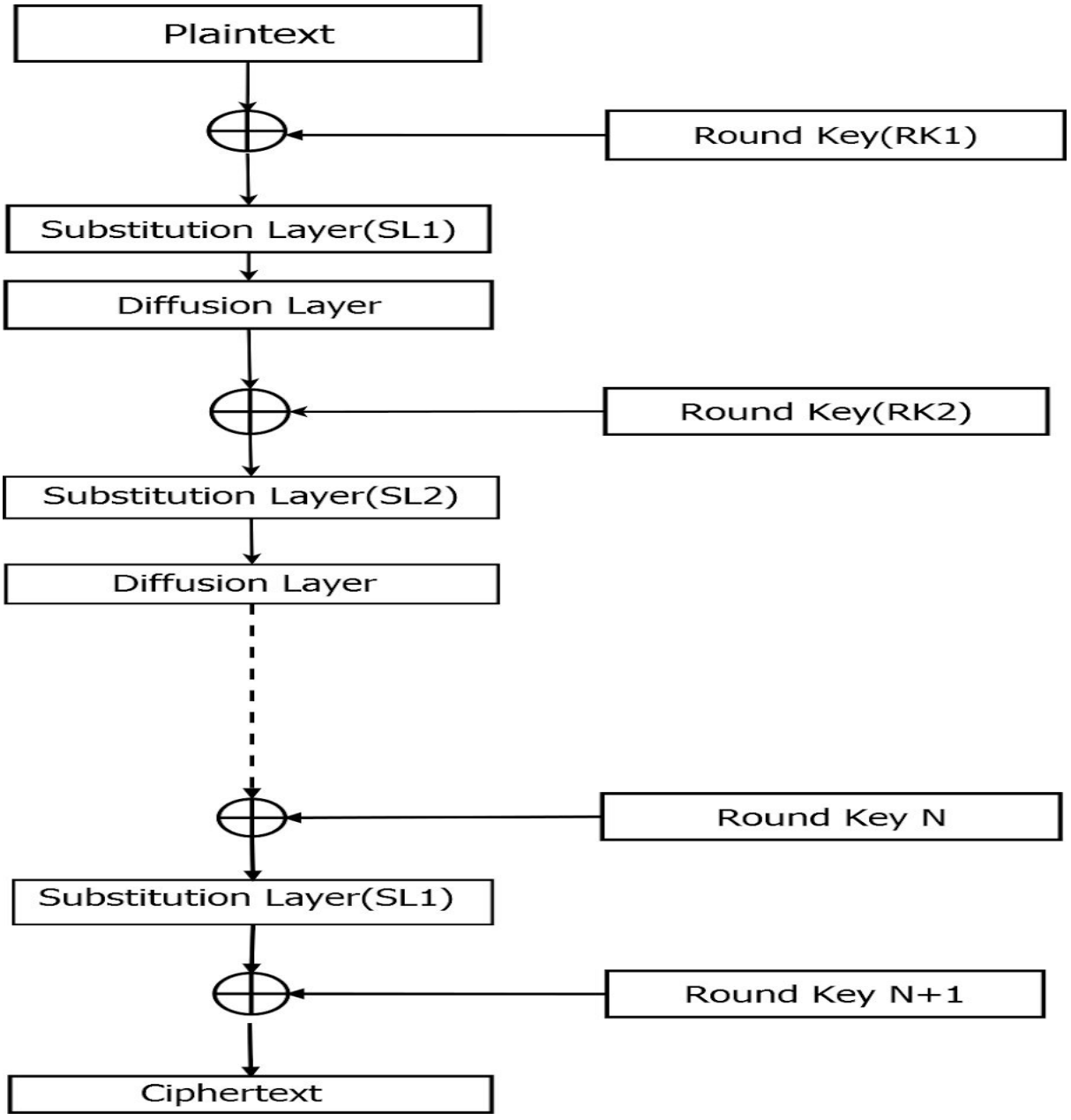
3. Diffusion Layer

For each output byte a determined seven bytes of the input is XORed with each other. The diffusion layer function is denoted as A. The function is involution, that is *A(A(x)) = x*. This property allows the ciphertext to be deciphered using the same function. The output of this step is input to the next round.

To sum up these 3 steps in one round function following formulas are used:

FO(D,RK) = A(SL1(D ^ RK))
FE(D,RK) = A(SL2(D ^ RK))

where *FO* is called the odd round function and *FE* is called the even round function. For the details of substitution layer and diffusion layer functions please refer to [2].



The number of rounds is dependent on the key size. For 128-bit key there are 12 rounds, 192-bit keys require 14 rounds and 256-bit keys require 16 rounds. Due to an extra key xor after the last round algorithms require 13,15 and 17 round keys respectively. Same round keys are used for encryption/decryption of all the 16 byte text blocks. Round key generation is called key scheduling and is explained below.

Key Scheduling

An 32 byte space is reserved for key. If 128-bit key is used the rightmost 16 bytes are padded with 0's. If 192-bit keys are used the rightmost 8 bytes are padded with 0's. Then we define four 128-bit values that will be used in the generation of the round keys.

Let KL and KR denote the leftmost 16 bytes and rightmost 16 bytes of the key. Then the key generators are:

W0 = KL
W1 = FO(W0, CK1) ^ KR
W2 = FE(W1, CK2) ^ W0
W3 = FO(W2, CK3) ^ W1

where FO and FE are round-functions which are also used in the substitution layers. CK1, CK2 and CK3 are determined constants by the ARIA algorithm. The key generators W0, W1, W32, W3 generate the round keys ek1,...,ek17 as follows:

ek1 = W0 ^ (W1 >> 19)
ek2 = W1 ^ (W2 >> 19)
ek3 = W2 ^ (W3 >> 19)
ek4 = (W0 >> 19) ^ W3
ek5 = W0 ^ (W1 >> 31)
ek6 = W1 ^ (W2 >> 31)
ek7 = W2 ^ (W3 >> 31)
ek8 = (W0 >> 31) ^ W3
ek9 = W0 ^ (W1 << 61)
ek10 = W1 ^ (W2 << 61)
ek11 = W2 ^ (W3 << 61)
ek12 = (W0 << 61) ^ W3
ek13 = W0 ^ (W1 << 31)
ek14 = W1 ^ (W2 << 31)
ek15 = W2 ^ (W3 << 31)
ek16 = (W0 << 31) ^ W3
ek17 = W0 ^ (W1 << 19)

where >> and << are right-circular-shift and left-circular-shift operations respectively. The round keys are then used for encryption of all the 16-byte text blocks. For 128-bit keys 13 of them are used, for 192-bit keys 15 and for 256-bit keys 17. Decryption round keys are derived from the encryption round keys:

dk1 = ek{n+1}, dk2 = A(ek{n}) .... dk{n+1}= ek1

5. Design And Implementation

This section discusses design and implementation of ARIA block cipher using CUDA. There are two different methods implemented for the ECB mode encryption, decryption. The first method assigns one thread per one text block(16 bytes). The second method assigns four method per one text block. For both of these methods different memory schemes and block sizes are benchmarked and compared.

The key scheduling process is carried out in the CPU, the round keys are generated on the CPU. Because each text block uses the same round keys and due to the sequential nature of the key scheduling this part is carried out in the CPU. The generated round keys are then moved to the GPU's global memory then to the shared memory of the blocks by the threads. Keeping them in the registers will limit the occupancy due to high register usage per thread. (At least 221 bytes must be stored in the registers which requires additional 56 registers.) Both methods keep the input text(plain or cipher) in their registers to achieve peak performance.

The first method assigns one thread to process one text block(16 bytes). The main advantage of this method is, there is no need for thread synchronization since each block's encryption process is independent. Each thread loads 16 bytes of text from the global memory to its registers and encrypts it. Each byte of the text is kept on a different register due to the operations on byte level such as substitution table look-ups and diffusion layer byte addition. Additional 16 registers are used to keep the intermediate results during diffusion layer which sums up to 32 registers per thread.

The second method assigns four threads per one text block to process collaboratively. The advantage of this method is to assign more threads in total to process the text. However, the experimental results shows significant amount of performance degrade compared to method 1. One of the main reasons behind this performance degrade is the frequent load and write to the shared memory and the need for thread synchronization after each shared memory write.

For the two methods mentioned above the two different memory schemes are tested. The first scheme stores the substitution layer tables denoted *SB1, SB2, SB3, SB4* in the constant memory. The second scheme stores the substitution layer tables in the shared memory. Based on the experiments shared memory usage is significantly faster for the storage of substitution layer tables. Both methods store the round keys in the shared memory and the input text in the registers. Another memory layout that may be tried would be to keep the input text (plain or cipher) in the shared memory instead of the registers. That can increase the theoretical occupancy which is limited by the registers, however shared memory latency and bank conflicts are introduced.

6. Experiments

In this section experiments are presented with varying key sizes and text sizes for each method with different memory layouts. The specification of the experimental system is as follows:

- CPU: Intel Core i7 4710MQ 2.5GHz / 3.5GHz
- Memory: 8 GB
- GPU: NVIDIA Geforce GTX 860M
- OS: Ubuntu 16.04
- CUDA Driver Version: 10.0

While measuring the total runtime on the GPU, kernel runtime, memory copies from host to device and device to host are included. Key scheduling is carried out on the CPU and the timing cost is not included for both GPU and CPU results.

Figure 1a shows the runtime of ARIA encryption algorithm on GPU with 128-bit key size and plaintext size varying from 32MB to 512MB for the first method.

Figure 1b shows the runtime of ARIA encryption algorithm on GPU with 128-bit key size and plaintext size varying from 32MB to 512MB for the second method.

Figure 1c compares the two methods for 128-bit key size. The best runtimes are selected from the Figure 1 and 2 for this comparison.

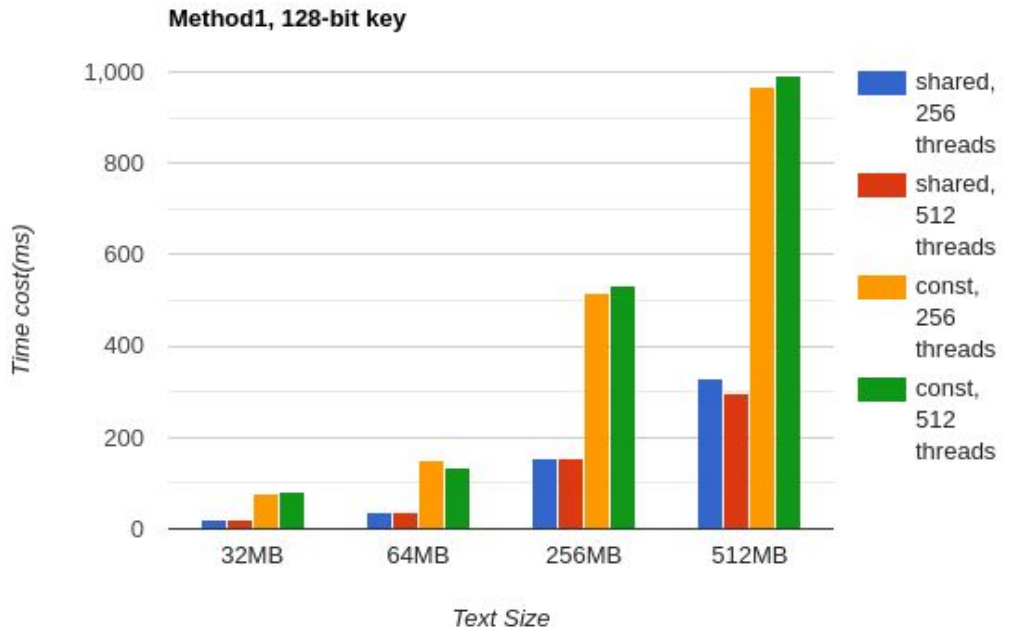


Figure 1a

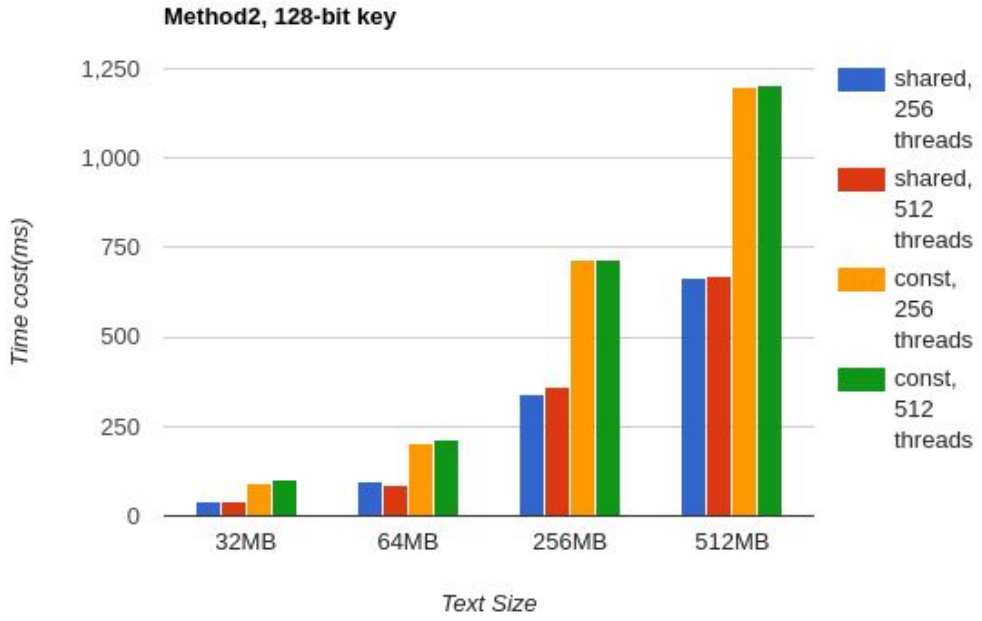


Figure 1b

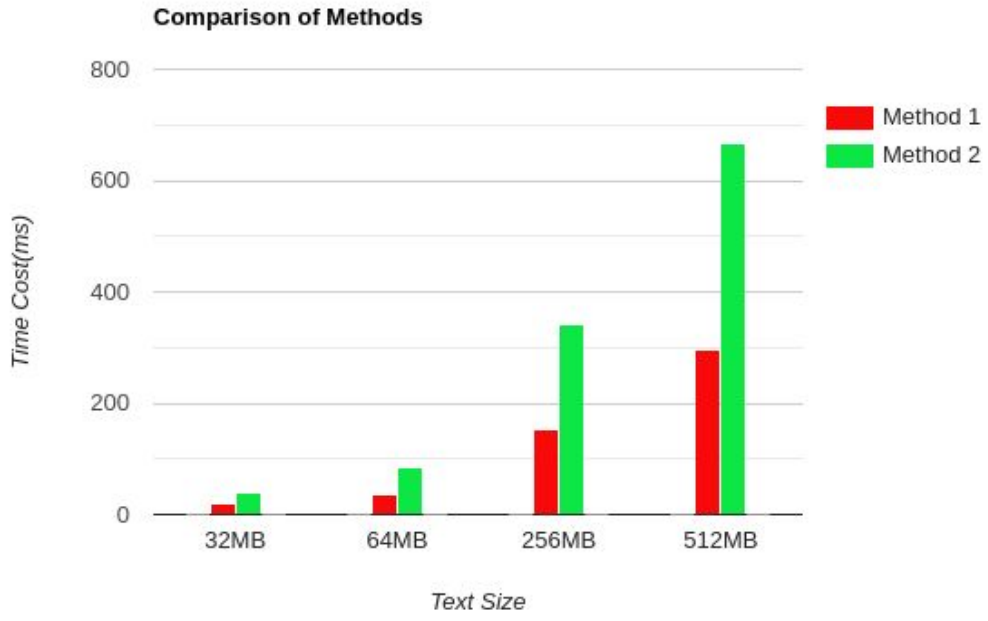


Figure 1c

Figure 2a shows the runtime of ARIA encryption algorithm on GPU with 256-bit key size and plaintext size varying from 32MB to 512MB for the first method.

Figure 2b shows the runtime of ARIA encryption algorithm on GPU with 256-bit key size and plaintext size varying from 32MB to 512MB for the second method.

Figure 2c compares the two methods for 256-bit key size.

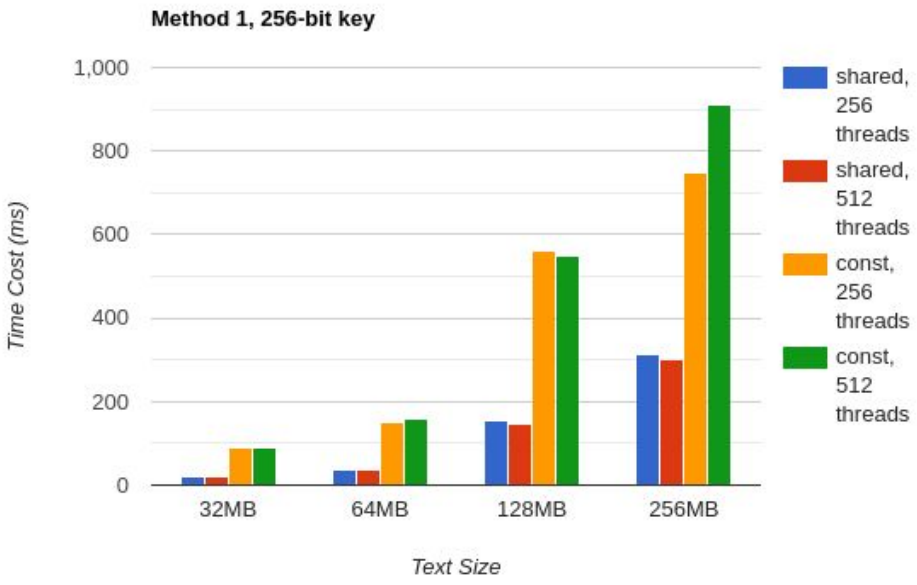


Figure 2a

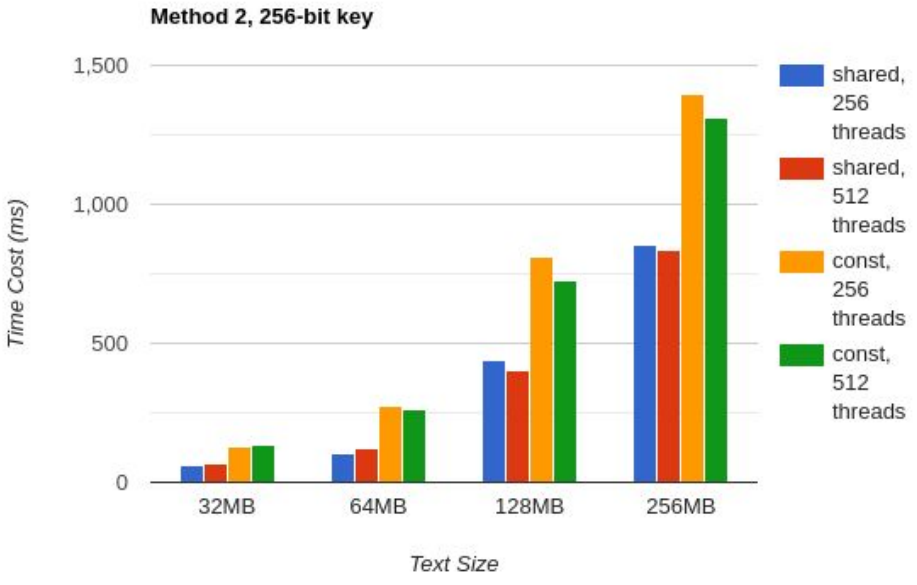


Figure 2b

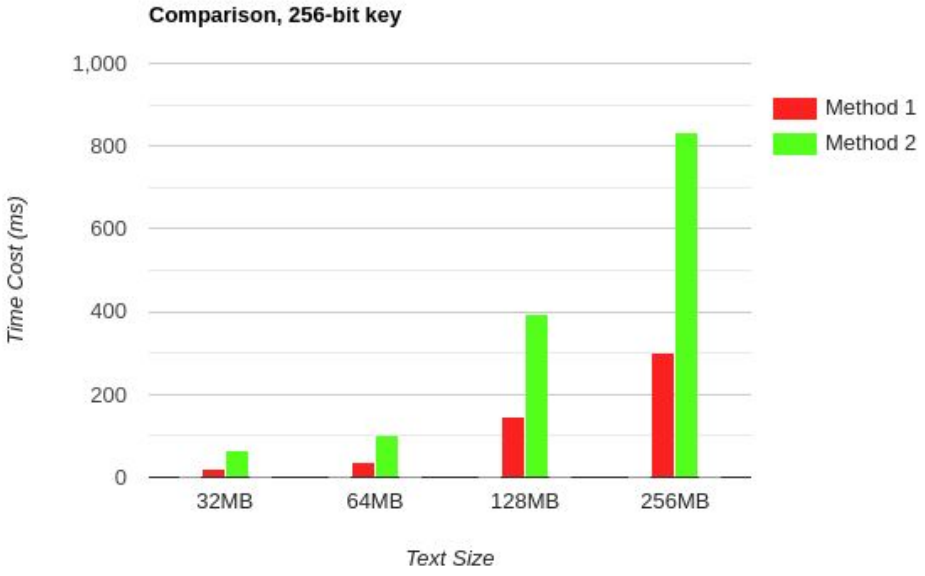
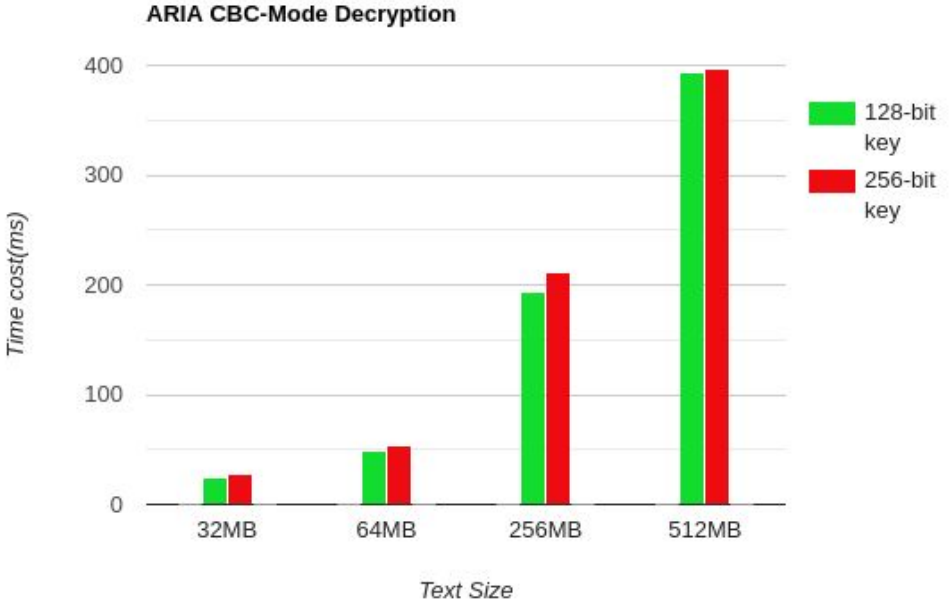


Figure 2c

CBC-Mode Decryption

Figure 3 shows the results for running time of ARIA block cipher in CBC-mode decryption with 128 and 256 bit key sizes. The running time is slightly higher than method 1 due to increased register usage thus limiting occupancy.



Comparison with Prior Work

The paper in [1] does not specify the key-size used. The performance increase for the first method is expected considering the GPU model used in the experiments in [1] is Nvidia GTX 285, an older model. However, the paper accomplishes better results for method 2. Method 2 implementation for this paper is therefore inadequate and must be improved. Another unexpected result is the running times with substitution layer tables stored in the constant memory. The shared memory option is also faster for the experiments conducted in [1] however the results are not as distinguishable.

Figure 4 makes performance comparison between CPU and GPU implementations of ARIA block cipher with 256-bit key size. For both CPU and GPU implementations key generation step are not included. For GPU implementation memory copies are included. This comparison shows a great performance gain in GPU implementation compared to CPU implementation of the ARIA encryption, decryption algorithm.

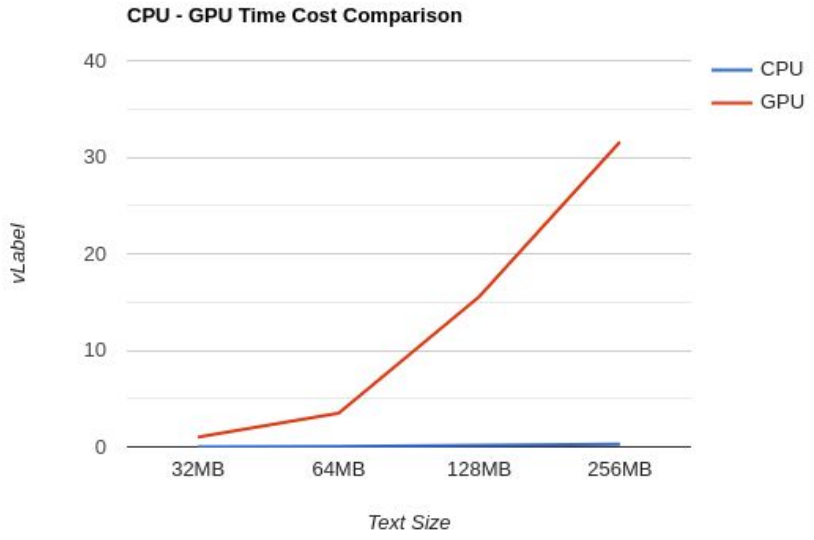


Figure 3

7. Conclusion

This paper presented a high performance implementation of ARIA block cipher. Experiments include different key-sizes with different methods and memory designs. Different block modes of operation implementations are implemented which are not present in previous works. However some methods were not as efficient as expected when compared to prior work due to implementation errors which will be improved. More block cipher modes can be written for ARIA block cipher. It has been shown that there is a great performance increase compared to the sequential implementation of ARIA on CPU.

8. References

[1] Xiao, Limin, et al. "High performance implementation of aria encryption algorithm on graphics processing units." High Performance Computing and Communications & 2013 IEEE International Conference on Embedded and Ubiquitous Computing (HPCC_EUC), 2013 IEEE 10th International Conference on. IEEE, 2013.

[2] Kwon, D., Kim, J., Lee, J., Lee, J., Kim, C.: A Description of the ARIA Encryption Algorithm. RFC 5794, March 2010. https://tools.ietf.org/html/rfc5794