Image Encryption With AES Algorithm
ECE1782 Project Report

April 13, 2016

Tong Zou 1001913815

Jing Wang 1002093183

1. Introduction

Image processing plays an important role in our daily life. There are a number of techniques can be implemented by CUDA, including image compression, blurring, detection, embossing, encryption and so on. During image transmission, there might be a problem that the information contained in the images can be leaked or hacked by attackers. Therefore, image encryption can be a useful way to prevent information leakage to some extent.

Image data, however, are generally large to compute and manipulate. Meanwhile, a good encryption method is also required to ensure that the encryption will not be hacked or modified. These require the computation and processing of the image pixel data. The encrypted image should be random enough so that the attacker cannot tell the detail of the pictures. As image data are processed faster in GPU than CPU, and different encryption methods yield various operating speeds. Therefore, CUDA can be used in this scenario to boost the calculating speed of image encryption.

In this project, an encryption method based on AES (advanced encryption standard) encryption algorithm is used to secure the image data and this kind of algorithm is implemented both in CPU and GPU to make comparisons of their execution speeds. The speed up results are the most important references for the performance of the project programs.

2. Expanded Motivation

As the methods and codes of AES algorithm are designed to run with CUDA, the algorithm can not only be used in image encryption but also data and plain text encryption, such as transaction information, software protection and email security.

Changing the pre-set substitution encoding table and round key values, the encryption method can be re-implemented in the same way. The input value lengths are not strictly limited, therefore the method can be adaptive to a number of encryption demands. The method can be used and referred in future AES algorithm implementations with GPU. More optimizations can be done to enhance the execution efficiency.

3. Related Work

AES is widely used as a standard encryption algorithm, which has been used in a number of data encryptions applications.

Deguang introduced an implementation of a parallel AES algorithm for fast data encryption on GPU [1]. The developed program gave a speedup of more than 7 times compared to CPU execution, with simple implementation of 10 rounds of AES algorithm. Stratulat used CUDA to implement AES on small and large data volume encryption [2], which can offer speedups of almost 40 times in comparison to CPU and up to 130 times in S-box step, given that the access time for the large data volume is included in the encryption time. In 2009, Kipper at University of Toronto used AES and GPU to encrypt keys with various lengths [3], the performance can be more than 14.5 times over a similar encryption on CPU. In terms of memory distribution and performance, according to Mei's study, a 10 rounds AES algorithm is introduced and the relationships between the performance and memory allocation is observed [4] (several sets of number of blocks and threads were tried to find the

suitable memory allocation for their cases). There are a number of other researches on AES algorithm with GPU, however, few of which is used to encrypt images.

4. Algorithm and Implementation

AES is a kind of symmetric block cipher which has longer key length compared with DES, making attackers more difficult to perform brute-force attack. Furthermore, it can be easily implemented in hardware and software.

4.1 Key Length and Block Size

It supports variable key and block lengths:

- Can use 128-, 192- or 256-bit keys
- Can operate on 128-, 192- or 256-bit blocks
- Any combination of key and block length is possible
- Extensions exist to allow it to take block and key lengths of 160 and 224 bits

Each input block is used to form a matrix of bytes called state. The state always has 4 rows, and a variable number of columns, depending on block size. In this project, the state is a matirx with 4 rows and 4 columns which is easy to feed into the threads and to compute.

4.2 AES Computation

AES is based on rounds and the number of rounds is based on the key length and block size (varies from 10-14). Each round consists of four stages:

- Byte Substitution transformation
- Shift Rows transformation
- Mix Columns transformation
- Addition of Round Keys

4.2.1 Byte Substitution

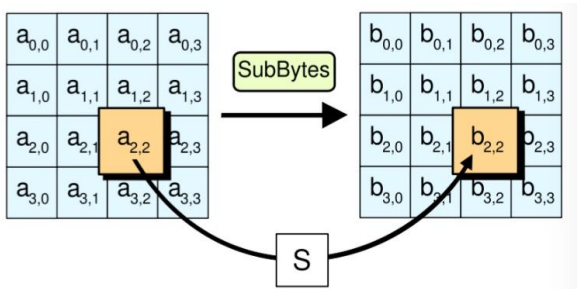


Fig 4.2.1 Byte Substitution

The formed matrix is input into the algorithm. Substitution is performed using a 256-entry lookup table called an S-box (shown in the appendix) which maps input characters to pre-set random characters.

4.2.2 Shift Rows

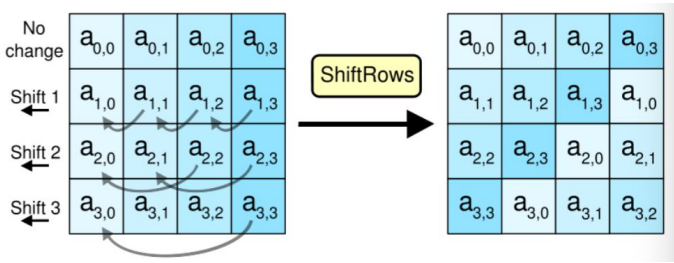


Fig 4.2.2 Shift Rows

This round operates on the rows of the state and iteratively shifts the bytes in each row by an offset. Blocks with different size have slightly different offsets.

4.2.3 Mix Columns

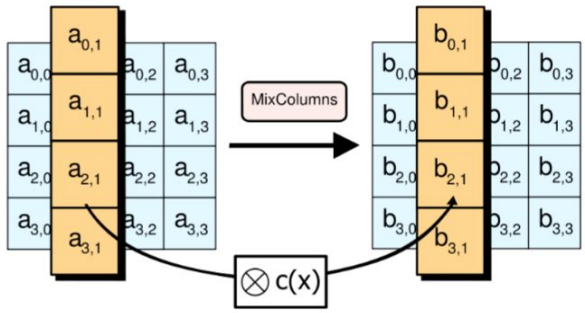


Fig 4.2.3 Mix Columns

After row shifting, a matrix c(x) is applied to multiply with each column of the state in order to mix the elements of each column.

4.2.4 Round Keys

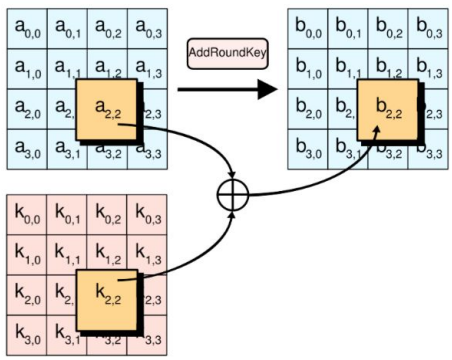


Fig 4.2.4 Round Keys

Round keys are generated from the original key for each round via a key schedule algorithm and then the round key is XORed with the state to produce the input to the next round.

4.3 AES Implementation On CPU

There are four main encryption functions along with their inverse functions for decryption in CPU - substitution, shift row, mix column and round key xor. They are shown in the table below:

Table 1: Encryption and decryption functions of the AES algorithm in CPU

Encryption Functions	Decryption Functions

`//byte substitution`
`for(i=0;i < size;i++)`
`{`
`  bitmapImage[i] = s_box[bitmapImage[i]];`
`}`
	`//byte substitution`
	`for(i=0;i < size;i++)`
	`{`
	`  bitmapImage[i] = inv_s_box[bitmapImage[i]];`
	`}`

`//shift rows`
`void shift_rows(unsigned char *state) {`
`  unsigned char i, k, s, tmp;`
`  for (i = 1; i < 4; i++) {`
`  s = 0;`
`  while (s < i) {`
`  tmp = state[Nb*i];`
`  for (k = 1; k < Nb; k++) {`
`  state[Nb*i+k-1] = state[Nb*i+k];`
`  }`
`  state[Nb*i+Nb-1] = tmp;`
`  s++;`
`  }`
`  }`
`}`
	`//inverse shift rows`
	`void inv_shift_rows(unsigned char *state) {`
	`  unsigned char i, k, s, tmp;`
	`  for (i = 1; i < 4; i++) {`
	`  s = 0;`
	`  while (s < i) {`
	`  tmp = state[Nb*i+Nb-1];`
	`  for (k = Nb-1; k > 0; k--) {`
	`  state[Nb*i+k] = state[Nb*i+k-1];`
	`  }`
	`  state[Nb*i] = tmp;`
	`  s++;`
	`  }`
	`  }`
	`}`

`//mix columns`
`void mix_columns(unsigned char *state) {`
`  unsigned char a[] = {0x02, 0x01, 0x01, 0x03};`
`  unsigned char i, j, col[4], res[4];`
`  for (j = 0; j < Nb; j++) {`
`  for (i = 0; i < 4; i++) {`
`  col[i] = state[Nb*i+j];`
`  }`
`  coef_mult(a, col, res);`
`  for (i = 0; i < 4; i++) {`
`  state[Nb*i+j] = res[i];`
`  }`
`  }`
`}`
	`//inverse mix columns`
	`void inv_mix_columns(unsigned char *state) {`
	`  unsigned char a[] = {0x0e, 0x09, 0x0d, 0x0b};`
	`  unsigned char i, j, col[4], res[4];`
	`  for (j = 0; j < Nb; j++) {`
	`  for (i = 0; i < 4; i++) {`
	`  col[i] = state[Nb*i+j];`
	`  }`
	`  coef_mult(a, col, res);`
	`  for (i = 0; i < 4; i++) {`
	`  state[Nb*i+j] = res[i];`
	`  }`
	`  }`
	`}`

```
void key_xor(unsigned char *state){
       for(int i=0;i < 16;i++)
       {
          state[i] = state[i]^key[i];
       }
}
```

These four kinds of functions above in the table are based on the basic AES algorithm described in the previous section. The input is firstly put into a substitution function and all the characters are substituted with pre-set random values in the s_box (the same during the decryption substitution stage). The s_box table is shown in the appendix section. While after this stage, the image is vague but the shape of the objects can still be noticed after the substitution. Therefore, the image information is not secured after this manipulation.

Then the processed image data are fed into the shift rows function. The preset row shifting parameter is 4. The shifted data are then multiply with a matrix a[] in a per-column manner. Finally the image is XORed with a key matrix which is generated from the user key of the AES algorithm. The decryption is basically run inverse functions in an inverse order as shown in the table above.

4.4 AES Implementation On GPU

The impelemtation of AES algorithm on CUDA is similar to the impelementation on CPU. The basic functions remain the same while they are set as device functions so that they can be called in the kernel. 16 pixels are loaded into one thread and the total number of blocks equals to the total pixel numbers divided by 512 x 16. This means each thread will process 16 pixels and there are 8192 pixels processed in each block. The memory allocation can be seen in the codes below:

```
__global__ void encrypt(unsigned char *bitmapImage, int size, int threadN)
{
       int threadId = threadIdx.x + blockIdx.x*blockDim.x;
       int i;
       unsigned char *p = bitmapImage;
       //substitution
       for(i = threadId * 16; i < (threadId+1) * 16; i++){
                   if(i < size)
                            bitmapImage[i] = s_box[bitmapImage[i]];
       }
       __syncthreads();
```

For loop is used to process 16 pixels per thread and syncthreads is to wait for all the threads finishing their work and then goes on to the next step.

4.5 Optimization With Shared Memory

Though the GPU performance of AES turned out to be fairly good, there are still several methods to optimize the kernel code. In this project, shared memory is used to boost the speed of execution and the data is fed into the shared memory in a per-block manner. The functions running in the shared memory are similar to the functions in the previous kernel codes. The final optimized version of the kernel codes with shared memory is shown below:

```
__global__ void encrypt(unsigned char *bitmapImage, int size, int threadN)
{
    int threadId = threadIdx.x + blockIdx.x*blockDim.x;
    __shared__ unsigned char sdata[512*16];
    int i;
    unsigned int tid = threadIdx.x;

    for(int k = tid * 16; k < (tid + 1) * 16; k++){
       int gid = k + blockIdx.x * 512 * 16;
        if(gid < size)
          sdata[k] = bitmapImage[gid];
    }
    __syncthreads();
     //substitution
    for(i = tid * 16; i < (tid+1) * 16; i++){
       sdata[i] = s_box[sdata[i]];
    }
    __syncthreads();
    //shift rows
    shift_rows(&sdata[tid * 16]);
    __syncthreads();
    //mix columns
    mix_columns(&sdata[tid * 16]);
    __syncthreads();
    //key_xor
    key_xor(&sdata[tid * 16]);
    __syncthreads();
    for(int k = tid * 16; k < (tid + 1) * 16; k++){
       int gid = k + blockIdx.x * 512 * 16;
        if(gid < size)
          bitmapImage[gid] = sdata[k];
    }
    __syncthreads();

}
```

5. Methodology

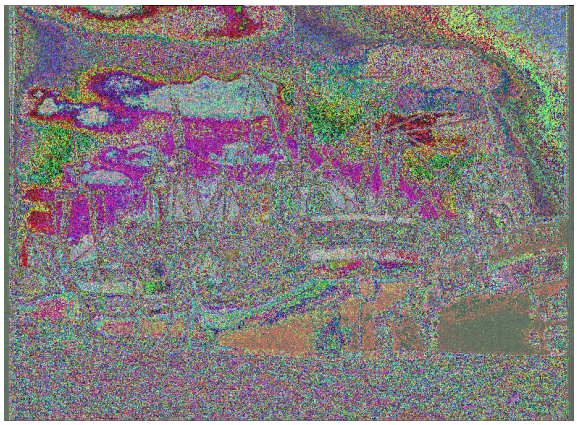
A BMP image (size: 1.4M) was used to test three different implementations of AES algorithm. The program was run 10 times for each version and then we calculated the average running time including encryption time and decryption time.

Four indicators are used to compare the performance.

- **Average Encryption Time (ms)** is the average time used to encrypt the image. The less time spent on encryption, the better performance the implementation has.
- **Average Decryption Time (ms)** is the average time used to decrypt the image. The less time spent on decryption, the better performance the implementation has.
- **Encryption Time Speedup (Compared with CPU) =** Average Encryption Time (CPU) / Average Encryption Time (GPU). The more speedup we get, the better performance the implementation has.
- **Decryption Time Speedup (Compared with CPU): =** Average Decryption Time (CPU) / Average Decryption Time (GPU). The more speedup we get, the better performance the implementation has.

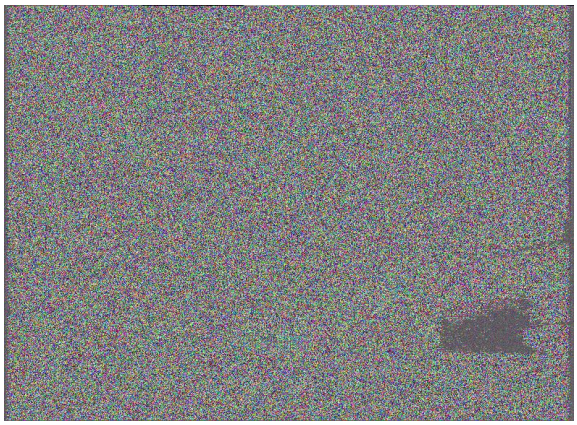
6. Evaluation

Before the time consumptions of the CPU and GPU codes are analyzed, the encryption performance can be seen directly from the encrypted images. The encrypted images of different stages are shown below:

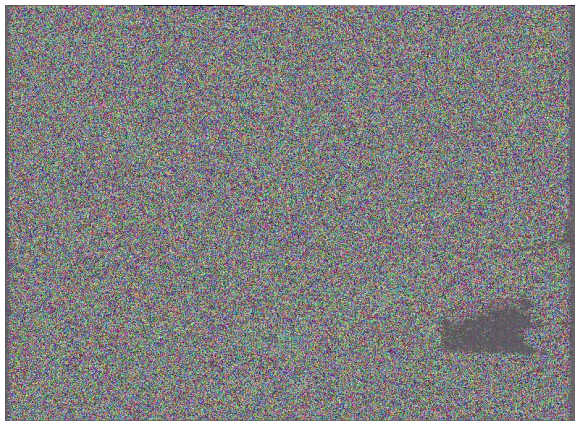


i. Substitution

ii. Shift Rows



iii. Mix Columns



iv. Key XOR

Fig 6. Original Picture

From the pictures above, it is obvious the first two stages of AES algorithm are not secure enough as the shapes of ships in the figure i and ii can still be distinguished. While after four stages of encryption, the image is fully encrypted and hard to recover by the attacker. There sometimes exists

another problem that when some parts of the image are totally black, the encryption method will hardly make that part completely random. This problem, however, will not hurt much thanks to most of images will not be completely black or white, and this can also be overcome by generating pseudo random values based on their locations if these pixels are nearly black or white.

The table below compares the performance of three different implementations of AES algorithm in terms of encryption/decryption time and corresponding speedup. In terms of encryption/decryption time, the speedup gained by using basic GPU implementation and GPU with shared memory is significant, around 350 and 600 respectively. Meanwhile, the encryption time and decryption time are roughly the same for three kinds of versions since AES algorithm is a kind of symmetric encryption algorithm.

Table 2: Performances in CPU and GPU

	CPU	GPU	GPU(shared memory)
Encryption Time (ms) (Average 10 time)	169.720096	0.474438	0.276883
Decryption Time (ms) (Average 10 time)	171.934595	0.500678	0.291840
Encryption Time Speedup (Compared with CPU)	1	357.728715	612.966834
Decryption Time Speedup (Compared with CPU)	1	343.403535	589.139922

7. Conclusion

The AES encryption algorithm for images has been implemented using CUDA, with a large speed gain. For a naive transfer from serial to parallel, this yields an improvement of just above 300 times in running time. Optimizing the code using shared memory to run more efficiently on the GPU pays off, with the final result running at more than a 600 times faster than the serial implementation. Futhermore, the image fully encrypted with four stages of AES algorithm has a good confidentiality property that makes attacker hard to recover.

However, AES algorithm is a type of iterated block cipher based on several rounds of encryption. For convenience, the authors only implemented four basic stages of one round of AES algorithm so the future improvement could be application of multiple rounds encryption based on the key length and the block size. Meanwhile, the strategy of parallelization can be further implemented for other famous algorithms such as RSA algorihtm and DES algorithm and the memory space could be better allocated to gain more speedup.

8. Reference

[1] D. Le, J. Chang, X. Gou, A. Zhang, and C. Lu, "Parallel AES algorithm for fast Data Encryption on GPU," *2010 2nd International Conference on Computer Engineering and Technology*, 2010

[2] Daniel, Tomoiagă Radu, and Stratulat Mircea. "AES Algorithm Adapted on GPU Using CUDA for Small Data and Large Data Volume Encryption." *International Journal of Applied Mathematics and Informatics* 5.2, 2011: 71-81.

[3] Kipper, Michael, Joshua Slavkin, and Dmitry Denisenko. "Implementing AES on GPU Final Report." *University of Toronto, Toronto,* 2009.

[4] C. Mei, H. Jiang, and J. Jenness, "CUDA-based AES parallelization with fine_tuned GPU memory utilization," *2010 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW)*, 2010

Appendices

The S_box of the AES algorithm:

```
__device__ static unsigned char s_box[256] = {
// 0     1     2     3     4     5     6     7     8     9     a     b     c     d     e     f
0x63, 0x7c, 0x77, 0x7b, 0xf2, 0x6b, 0x6f, 0xc5, 0x30, 0x01, 0x67, 0x2b, 0xfe, 0xd7, 0xab, 0x76, // 0
0xca, 0x82, 0xc9, 0x7d, 0xfa, 0x59, 0x47, 0xf0, 0xad, 0xd4, 0xa2, 0xaf, 0x9c, 0xa4, 0x72, 0xc0, // 1
0xb7, 0xfd, 0x93, 0x26, 0x36, 0x3f, 0xf7, 0xcc, 0x34, 0xa5, 0xe5, 0xf1, 0x71, 0xd8, 0x31, 0x15, // 2
0x04, 0xc7, 0x23, 0xc3, 0x18, 0x96, 0x05, 0x9a, 0x07, 0x12, 0x80, 0xe2, 0xeb, 0x27, 0xb2, 0x75, // 3
0x09, 0x83, 0x2c, 0x1a, 0x1b, 0x6e, 0x5a, 0xa0, 0x52, 0x3b, 0xd6, 0xb3, 0x29, 0xe3, 0x2f, 0x84, // 4
0x53, 0xd1, 0x00, 0xed, 0x20, 0xfc, 0xb1, 0x5b, 0x6a, 0xcb, 0xbe, 0x39, 0x4a, 0x4c, 0x58, 0xcf, // 5
0xd0, 0xef, 0xaa, 0xfb, 0x43, 0x4d, 0x33, 0x85, 0x45, 0xf9, 0x02, 0x7f, 0x50, 0x3c, 0x9f, 0xa8, // 6
0x51, 0xa3, 0x40, 0x8f, 0x92, 0x9d, 0x38, 0xf5, 0xbc, 0xb6, 0xda, 0x21, 0x10, 0xff, 0xf3, 0xd2, // 7
0xcd, 0x0c, 0x13, 0xec, 0x5f, 0x97, 0x44, 0x17, 0xc4, 0xa7, 0x7e, 0x3d, 0x64, 0x5d, 0x19, 0x73, // 8
0x60, 0x81, 0x4f, 0xdc, 0x22, 0x2a, 0x90, 0x88, 0x46, 0xee, 0xb8, 0x14, 0xde, 0x5e, 0x0b, 0xdb, // 9
0xe0, 0x32, 0x3a, 0x0a, 0x49, 0x06, 0x24, 0x5c, 0xc2, 0xd3, 0xac, 0x62, 0x91, 0x95, 0xe4, 0x79, // a
0xe7, 0xc8, 0x37, 0x6d, 0x8d, 0xd5, 0x4e, 0xa9, 0x6c, 0x56, 0xf4, 0xea, 0x65, 0x7a, 0xae, 0x08, // b
0xba, 0x78, 0x25, 0x2e, 0x1c, 0xa6, 0xb4, 0xc6, 0xe8, 0xdd, 0x74, 0x1f, 0x4b, 0xbd, 0x8b, 0x8a, // c
0x70, 0x3e, 0xb5, 0x66, 0x48, 0x03, 0xf6, 0x0e, 0x61, 0x35, 0x57, 0xb9, 0x86, 0xc1, 0x1d, 0x9e, // d
0xe1, 0xf8, 0x98, 0x11, 0x69, 0xd9, 0x8e, 0x94, 0x9b, 0x1e, 0x87, 0xe9, 0xce, 0x55, 0x28, 0xdf, // e
0x8c, 0xa1, 0x89, 0x0d, 0xbf, 0xe6, 0x42, 0x68, 0x41, 0x99, 0x2d, 0x0f, 0xb0, 0x54, 0xbb, 0x16};// f
```

The round key of the AES algorithm:

```
__device__ unsigned char key[16] = {
        0x00, 0x01, 0x02, 0x03,
        0x04, 0x05, 0x06, 0x07,
        0x08, 0x09, 0x0a, 0x0b,
        0x0c, 0x0d, 0x0e, 0x0f};
```