IMAGE ENCRYPTION

Report by:

1. Ankit Lakra 201301137
2. Aditya P.S.V.S 201301427

Introduction

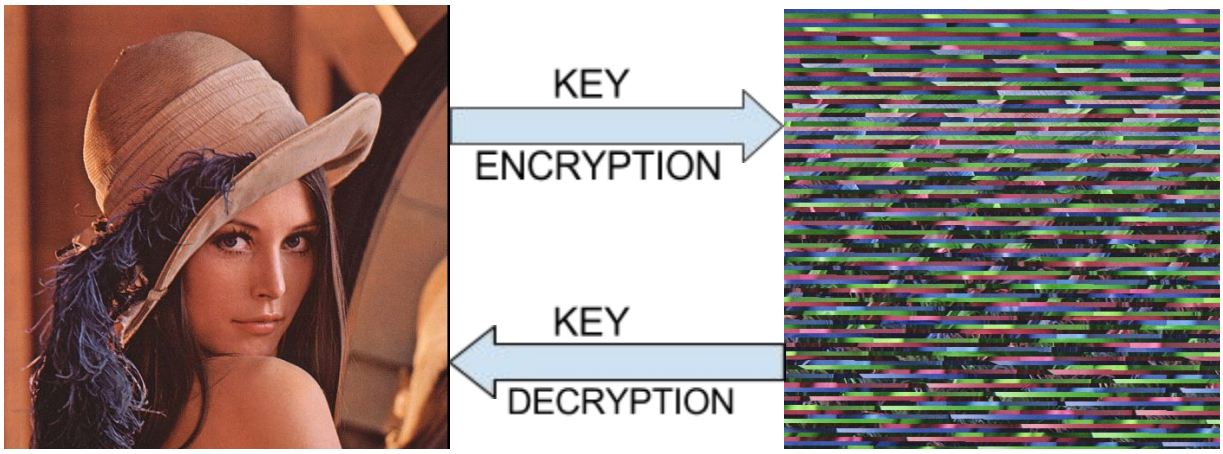
This project report gives an illustration of image encryption, i.e., hiding of image data from unauthorized access. Image encryption tries to convert an original image to another image that is hard to understand, and keep the image confidentiality between users. In order to fulfill, many encryption methods have been proposed.

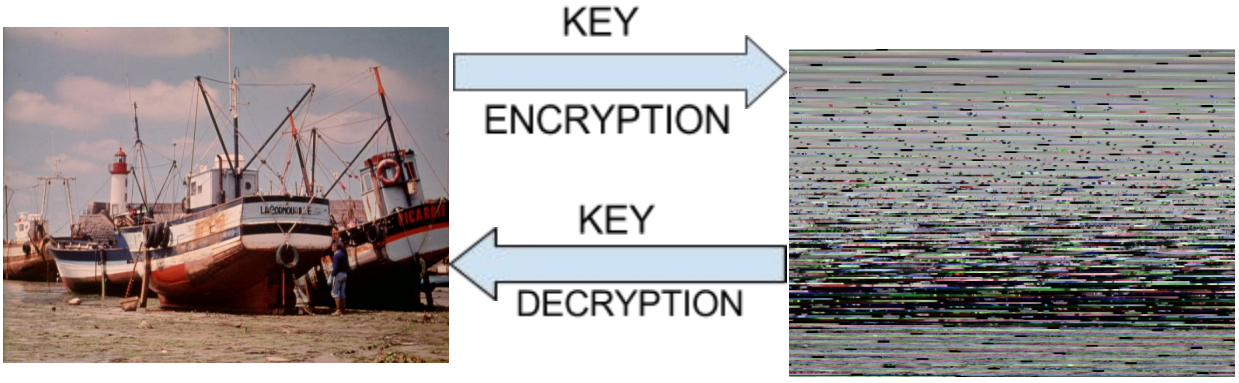
In this project report, a custom made encryption method is implemented, that uses the principle of position permutation based algorithm. Though this may not be efficient, the goal is solely to demonstrate the procedure of encryption in serial and then in parallel. This has been demonstrated in the following way:

- Serial code: C language
- Parallel code: CUDA C language

Also, only images with ".bmp" format have been used.

The user is asked to provide an image and a key to encrypt it. The same key is to be used for decryption and recovery of the image. The following flow diagram demonstrates this process.


KEY
ENCRYPTION
KEY
DECRYPTION


KEY
ENCRYPTION
KEY
DECRYPTION

Serial Algorithm

Consider an image “lenna.bmp” and a key “8000” provided by the user. Let the image be of 512x512 pixels. Each pixel consists of 3 values that describes the intensity of three colours red, green and blue (RGB). Thus, the array in which the image data is stored is of size 512*512*3 = 786432 bytes. Note that a BMP file consists of a header, which contains all the information about the file, and data. We only require the data to be stored in our array.

In a BMP file, the data is stored in BGR format. This is required to be converted to the standard RGB format, i.e., swapping of G and R. This is done by the following segment of code.

```
for (imageIdx = 0; imageIdx < bitmapInfoHeader->biSizeImage;imageIdx+=3)
{
    tempRGB = bitmapImage[imageIdx];
    bitmapImage[imageIdx] = bitmapImage[imageIdx + 2];
    bitmapImage[imageIdx + 2] = tempRGB;
}
```

Now that the array is in proper order, it is ready to be encrypted. In this illustration, we used a custom made encryption method, which works in the following way.
The user has assigned a key as “8000” to encrypt the image. The array that holds the image data is divided into blocks that contains “key” (8000 in this case) number of elements. In case the last block cannot accommodate 8000 elements, that block is ignored. Each of these blocks is then flipped such that, the 1st element and 8000th element are swapped, the 2nd element and 7999th element, and so on. This is done to each of the blocks. This is done with the following code segment.

```
void encrypt(unsigned char *bitmapImage, int size, int key)
{
    int count;
    unsigned char mid, temp;
    for(int i=0; i<size; i+=key)
    {
        if(i+key>size)
            break;

        for(int j=0; j<key/2; j++)
        {
            temp = bitmapImage[i+j];
            bitmapImage[i+j] = bitmapImage[(((i+j)/key)*key)+key-((i+j)%key)-1];
            bitmapImage[(((i+j)/key)*key)+key-((i+j)%key)-1] = temp;
        }
    }
}
```

With the array encrypted, it is converted back to a BMP file, and all that the user sees is an unrecognizable image.

To decrypt the image, the encrypted image along with the same key is given as input. The data array undergoes the same procedure as above, which is basically undoing of flipping of blocks, and the original image is obtained.

Note that the level of encryption depends on the key provided by the user. Larger the value of key, more is the image encrypted. However, the value of key should not exceed the size of the data.

Complexity

Swapping of R and B has the complexity of O(size/3).

Both the encryption and decryption use two loops, the first running ‘size/key’ time and the second loop runs ‘key/2’ times.
Thus, the complexity is O(size/2).

Scope of parallelism

The following three sections of the code can be parallelized.

- R&B swap
- Encryption
- Decryption

In each of the above cases, the loop can be parallelized among threads, such that each swap is done by one thread.

Effect of problem size

As problem size increases, more and more pixels are needed to be encrypted, i.e., more number of swaps are required to be performed. The following table shows this.

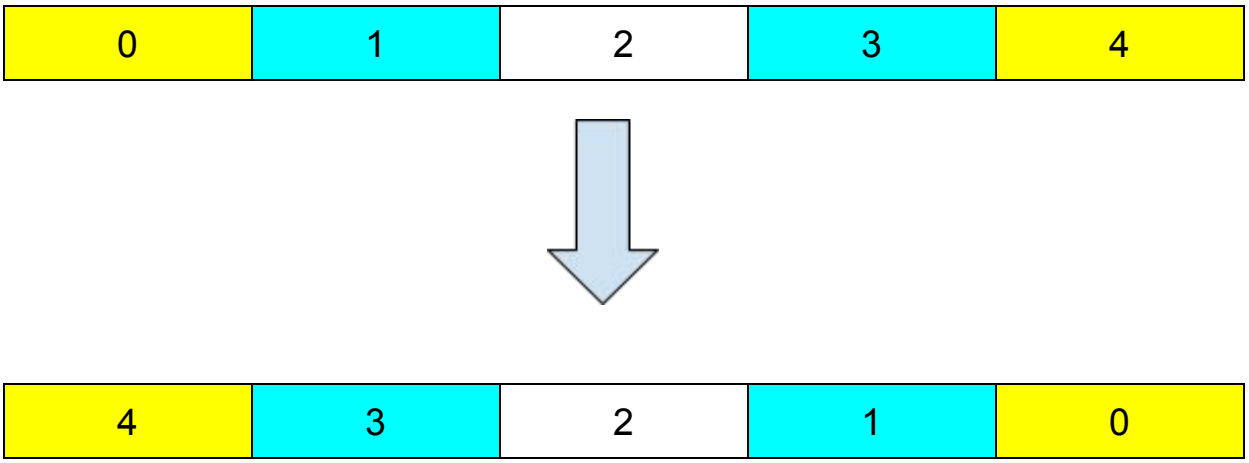
Size of image	Size of problem size	Time in ms (serial) for Encryption/Decryption	TIme in ms (serial) for Swap
500 x 480	720000	5.801984	0.002048
512 x 512	786432	6.050786	0.320224
787 x 576	1361664	11.86505	1.602528
1024 x 768	2359298	27.71142	4.487104

From above, we can observe that as problem size increases, the time taken by serial code increases with problem size.

Parallel Algorithm

Strategy

Consider a small array of size 5. We need to swap the first two elements with the last two, as shown.



This can be done by launching a thread for every swap that takes place, i.e., one thread for swapping 0 and 4, and another for swapping 1 and 3.

The above strategy can be applied for R&B swap, Encryption and Decryption, and parallelize them.

Parallelized code

- R&B Swap

```
__global__ void RB_Swap(unsigned char *imageData, int size)
{
    int imageIdx = threadIdx.x+blockIdx.x*blockDim.x;

    if(imageIdx<size/3)
    {
        unsigned char tempRGB;
        imageIdx = imageIdx*3;
        tempRGB = imageData[imageIdx];
        imageData[imageIdx] = imageData[imageIdx + 2];
        imageData[imageIdx + 2] = tempRGB;
    }
}
```

- Encryption

```
__global__ void encrypt(unsigned char *bitmapImage, int size, int key)
{
    int threadId = threadIdx.x + blockIdx.x*blockDim.x;
    int half = key/2;
    int index = ((threadId/half)*key) + (threadId%half);
    int swap = index + (key - (2*(index%half)) - 1);

    if((swap)<size)
    {
        unsigned char temp;
        //unsigned mid = bitmapImage[((index/half)*key) + half];

            temp = bitmapImage[index];
            bitmapImage[index] = bitmapImage[swap];
            bitmapImage[swap] = temp;
    }
}
```

- Decryption

```
__global__ void decrypt(unsigned char *bitmapImage, int size, int key)
{
    int threadId = threadIdx.x + blockIdx.x*blockDim.x;
    int half = key/2;
    int index = ((threadId/half)*key) + (threadId%half);
    int swap = index + (key - (2*(index%half)) - 1);

    if((swap)<size)
    {
        unsigned char temp;
        //unsigned mid = bitmapImage[((index/half)*key) + half];

            temp = bitmapImage[index];
            bitmapImage[index] = bitmapImage[swap];
            bitmapImage[swap] = temp;
    }
}
```

Results and Plots

Readings

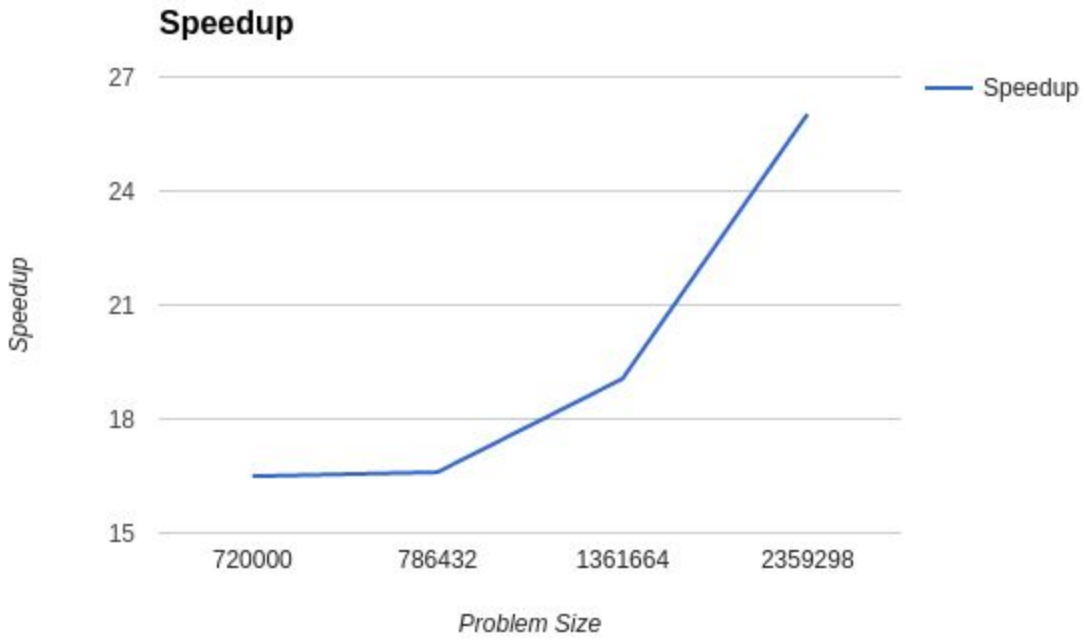
Size of image	Size of problem size	Time in ms (parallel) for Encryption/Decryption	TIme in ms (parallel) for Swap
500 x 480	720000	0.351776	0.187392
512 x 512	786432	0.364512	0.201728
787 x 576	1136160	0.596448	0.30720
1024 x 768	2359298	1.064928	0.565248

Computation Time vs Communication Time

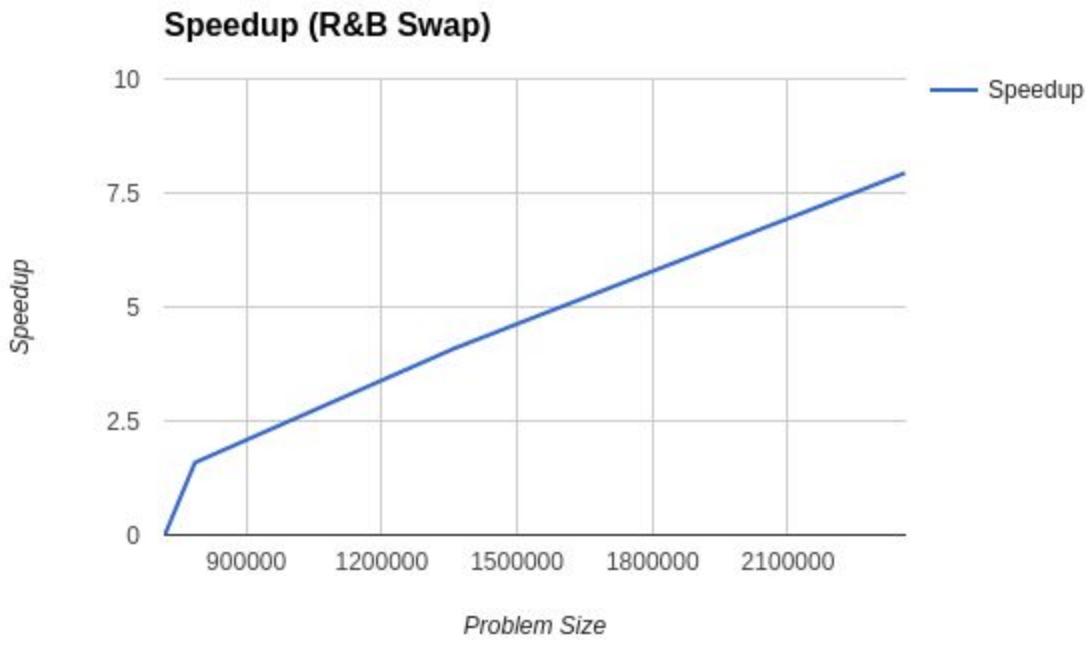
Size of problem size	Computation Time	Communication Time
720000	0.351776	0.303328
786432	0.364512	0.30048
1361664	0.596448	0.165792
2359298	1.064928	0.745536

Speedup Curve

Problem Size	Speedup (Encryption and Decryption)
720000	16.49340489
786432	16.59968945
1361664	19.05849396
2359298	26.0218719



Problem Size	Speedup (R&B Swap)
720000	0.01092896175
786432	1.587404822
1361664	4.093930104
2359298	7.938292572



Observations and Conclusions

From the above plots and readings, we can see that with the increase in problem size, the speedup tends to increase. Hence, image encryption is a very good application that can be implemented in parallel for large images.

Though speedup could be further improved using shared memory of the device, it is not possible to do so in our custom made encryption algorithm because the data in the array are not independent.
Most of the image encryption algorithms require jumbling of pixels, and so using of shared memory is not feasible.

Further Scope

This strategy of parallelization can be further implemented for other encryption algorithms like:

- RSA Algorithm
- Chaos Algorithm etc.

Additional Information

HARDWARE DETAILS

General Information of Device:
Name: NVidia GeForce GT 740M
Clock Rate: 1032500
Memory Information:
Total Global Memory: 2147352576
Total Constant Memory: 65536
MP Information:
Multiprocessor Count: 2
Shared memory per MP: 49152
Registers per MP: 65536
Threads in warp: 32
Max threads per Block: 1024
Max thread dimension:(1024 1024 64)
Max grid dimension: (2147483647 65535 65535)