**ECE 406**

**Team member: Qingyue Zhang, Haoxiang Hu;**

**Final Project: Scrambled Image Recovery**

**Part A: CPU Version of Scrambled Image Recovery**

Three main steps are implemented in descramble image based on row and column check-sum information.

**1. XOR Calculation:**

First, using “xor” calculation, the check-sum value of entire column and row from both scrambled image and actual image (based on .csv file) are derived. In order to avoid xor intricacy, the first value of both check-sum array are assigned by the initial value of check-sum list just in case.

**2. Column and Row Recovery:**

After two check-sum list are ready, we compare those two respectively to narrow the scope.

In order to record the elimination, a data structure called “foo” is created. “foo” includes two parts: list and counter. For instance, if the check-sum value from scrambled image column 7 is the same with check-sum value of original image column 1, then add “7” to the corresponding “foo” and increment counter.

Two “foo” structure lists are created to record row and column possibility. In the program, “row\_poss” is used to record column possibility, and “col\_poss” is used to record row.

After all the check-sum comparison, another function “de-duplicate” is called to find the duplicate value of different list possibility. For instance, for the column check-sum list, is the list in row\_poss[0] contains column [7], [9], [17] from the scrambled image, and the list in row\_poss[3] contains column [7] only from the scrambled image, then column [7] can be eliminated from row\_poss[0];

**3. Box Recorvery:**

After all the check-sum values of column and row are compared and organized, it’s time to compare the check-sum value of the box. Take the example below as an illustration:

Assuming a 2\*2 box, row\_poss[0].list=[7, 9], col\_poss[0].list=[1, 2], col\_poss[1].list=[3, 5], as shown below:

row\_poss[0].list=[7, 9], col\_poss[0].list=[1, 2], col\_poss[1].list=[3, 5]

In order to eliminate possibilities, first column 7 is chosen from row\_pass. Combined with all the choices from col\_poss[0] and col\_poss[1], the possible pairs are shown below:

|  |  |  |  |
| --- | --- | --- | --- |
| (7, 1) | (7, 1) | (7, 2) | (7, 2) |
| (7, 3) | (7, 5) | (7, 3) | (7, 5) |

After that, we can compare the check-sum of each pair to the check-sum of original box. If non of them matches, then eliminate column “7” from row\_poss.

Repeat that procedure for every columns and rows until minimum possibilities of each pixel has been reached. Ideally the best case is each pixel have unique value, however, imperfections can happen in the process, resulting in less accurate de-scrambled image.

**Image comparison are shown below:**

**Small scrambled:**

C:\Users\Haoxiang Hu\Desktop\matlab_code\small.png_scrambled.png

**Small recovered:**

C:\Users\Haoxiang Hu\Desktop\matlab_code\test3.png

Small Picture CPU Recovery Run Time: **0.02ms**

**Medium scrambled:**

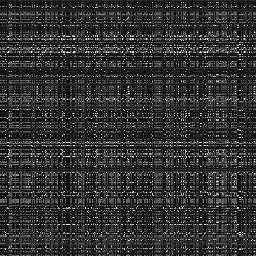


**Medium recovered:**

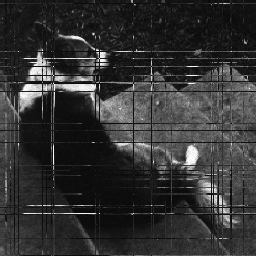


Medium Picture CPU Recovery Run Time: **12ms**

**Large scrambled:**



**Large recovered:**



Large Picture CPU Recovery Run Time: **161ms**

**Part B: GPU Version of Scrambled Image Recovery**

The block size is chosen by the user, and since the total number of threads equals to the 4 times of image height/width, the number of blocks equals to 2\*(height + width)/block size.

**GPU Main**

There are 4 inputs for the main function: filename for the input image, checksum csv name, block size and height/width of the image. We read in the image as a 2d array and checksum as row checksum and column checksum array, which has number of boxes elements. Then we start the GPU part.

**GPU Kernel**

In the GPU version, we have put the part of calculating “xor” of entire column and row from both scrambled image and original image (based on .csv file) into the kernel. The input for the GPU kernel is the descrambled image, the checksum for each boxes’ row and column from the original image, GPU version of large row/column checksum for both original image and descrambled image. There will be four kinds of threads. The first group of threads whose indexes are less than the image height/width will be responsible to calculate the large row checksums of original image, then the next group whose indexes are between the 2 times of image height/width and 3 times of image height/width will do the calculation of large column checksums of original image, then the next group the large row checksums of scrambled image, then the large column checksums of scrambled image.

After generating the checksum for large row and column of scrambled image and original image, passing the GPU data to the CPU data and free the memory, the code will return to the CPU main function and call restore function.

**CPU function restore**

Then we will start to compare the scrambled row/column checksum with all the rest of the row/column checksum of the original image. If there is a match, count increments. At the end of the loop, if count equals to 1, we will store the index of the match column/row. To restore the descrambled image, we assign the pixels of the stored column/row to the original image matrix and output the original image matrix as the final result.

**Hindrance to the performances**

In order to avoid the thread divergence, we have used the idea of group of threads. In each group, the threads will perform exactly the same task, either to calculate the row or column checksum of original or scrambled image. Since the number of threads per group equals to the image height/width, which always larger than the number of threads in a warp, the thread divergence will be avoided.

**The final results are shown below:**

**Small recovered:**



**Medium recovered:**



**Large recovered:**



|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Image size | Block size | Block num | Group of thread size | Box size | Run time |
| 32x32 | 2 | 64 | 32 | 2x2 | 0.05 ms |
| 4 | 32 | 4x4 | 0.05 ms |
| 8 | 16 | 8x8 | 0.06 ms |
| 96x96 | 2 | 192 | 96 | 2x2 | 0.11 ms |
| 4 | 96 | 4x4 | 0.11 ms |
| 8 | 48 | 8x8 | 0.11 ms |
| 256x256 | 2 | 512 | 256 | 2x2 | 0.29 ms |
| 4 | 256 | 4x4 | 0.30 ms |
| 8 | 128 | 8x8 | 0.30 ms |

The number of box is chosen to equal the box size, to make sure the number of thread is always equal to the multiple times of threads doing individual checksum calculation. The speed up for the large image is 5x, and the larger the input pixels, the greater speed up the gpu version can get comparing with the cpu version. The followed screenshot of CUDA\_Occupancy\_Calcultor is the prof of that. When using 512 as the threads per block, the occupancy can reach to a local maximum.

