CMPT 365 Programming Assignment 1

Junchen Li

301385486

2021/2/23

Question 1

Introduction: In this task, we have the input Y U V = 156, 28, 37 and try to print out the value of Y Co Cg and figure out the running time of program. Find the optimal way to implementation this task.

Experiment Design: I design to make three different scenarios to test the running time of program. The basic idea for this part is using given Y U V, try to convert them to the R G B respectively. And using R G B to calculating the Y Co Cg.

According to the slides from the lecture, the basic matrix formulas are:

$$\begin{matrix} Y \\ U \\ V \end{matrix} = \begin{bmatrix} 0.299 & 0.587 & 0.114 \\ -0.299 & -0.587 & 0.886 \\ 0.701 & -0.587 & -0.114 \end{bmatrix} \begin{matrix} R \\ G \\ B \end{matrix} \qquad \begin{matrix} Cg \\ Y \\ Co \end{matrix} = \begin{bmatrix} 1/2 & -1/4 & -1/4 \\ 1/2 & 1/4 & 1/4 \\ 0 & -1/2 & 1/2 \end{bmatrix} \begin{matrix} G \\ B \\ R \end{matrix}$$

I design to run one million times for each scenario, calculate each average running times and analyze the complexity and try to make it as fast as possible

Experiment Result: Y = 160 Co = 4 Cg = -28

Analyzing the program:

① The first one just follow the formulas and use the traditional way to calculate each variable. Firstly, we have the input of Y U V, so we need to find a matrix which can reverse process and get the R G B. [Y U V* matrix1 * matrix 2 = Y Co Cg]

$$\begin{bmatrix} 0.299 & 0.587 & 0.114 & 1 & 0 & 0 \\ -0.299 & -0.587 & 0.886 \mid & 0 & 1 & 0 \\ 0.701 & -0.587 & -0.114 & 0 & 0 & 1 \end{bmatrix} \qquad \begin{bmatrix} 0.299 & 0.587 & 0.114 & 1 & 0 & 0 \\ 0 & 0 & 1 \mid & 1 & 1 & 0 \\ 0.701 & -0.587 & -0.114 & 0 & 0 & 1 \end{bmatrix}$$

$$\begin{bmatrix} 0.299 & 0.587 & 0.114 & 1 & 0 & 0 \\ 0 & 0 & 1 \mid & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 & 1 \end{bmatrix} \qquad \begin{bmatrix} 1 & 0 & 0 & 1 & 0 & 1 \\ 0.299 & 0.587 & 0.114 \mid & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 & 1 & 0 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 0 & 0 & 1 & 0 & 1 \\ 0.509369 & 1 & 0.1942078 \mid & 1.703 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 \end{bmatrix} \qquad \begin{bmatrix} 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0.194 & \mid 1.194 & 0 & -0.51 \\ 0 & 0 & 1 & 1 & 1 & 1 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & \mid 1 & -0.194 & -0.51 \\ 0 & 0 & 1 & 1 & 1 & 0 \end{bmatrix}$$ (Notice than we approximate 0.51 and 0.194)

so the inverse matrix ($matrix^{-1}$) is $\begin{bmatrix} 1 & 0 & 1 \\ 1 & -0.194 & -0.51 \\ 1 & 1 & 0 \end{bmatrix}$. Then we follow the normal matrix calculating rule to get the values of Y Co Cg. The integer result as follow: Y = 159 Co = 4 Cg = -28. Average running time is 1722 ns.

② The second I did some modification for the matrix. I try to only use one matrix which can get the Y Co Cg value directly. [Y U V* matrix_merge = Y Co Cg]

Firstly, change the R G B matrix follow the above formula:

$$R = Y + V$$
$$G = y - 0.194\,U - 0.51\,V$$
$$B = Y + U$$

and we put R G B into other matrix formula:

$$\begin{cases} Y = 0.5Y + 0.5V + 0.25Y - 0.0485U - 0.1275V + 0.25Y + 0.25U \\ Co = -0.5Y - 0.5U + 0.5Y + 0.5V \\ Cg = 0.5Y - 0.097U - 0.255V - 0.25Y - 0.25U - 0.25Y - 0.25V \end{cases}$$

The final matrix is $\begin{bmatrix} 1 & 0.153 & -0.005 \\ 0 & -0.5 & 0.5 \\ 0 & -0.347 & -0.505 \end{bmatrix}$. For this situation, the integer result as follow: Y = 160 Co = 4 Cg = -28. Average running time is 94 ns.

③ For the third situation, I just made a slight change, we not change the value “Y”. Because the value Y stands for the brightness for the original one and the output one. So we can assume the Y value not change. However, it doesn’t have an obvious change for running time. The integer result as follow: Y = 156 Co = 4 Cg = -28. Average running time is 80 ns.

④ As I mentioned before, I have three different situations. However, when I thinking how to make it faster, I notice the timing method also will influence the total running time of the program. I changed the start point and end point from the calling part to the totally loop.

Original

```
for (int i = 0; i< 1000000; i++){

long startTime = System.nanoTime();
result = convert_to_YCoCg(y,u,v);
result = modify_one (y,u,v);
result = modify_two (y,u,v);
long endTime = System.nanoTime();
}
```

Modified

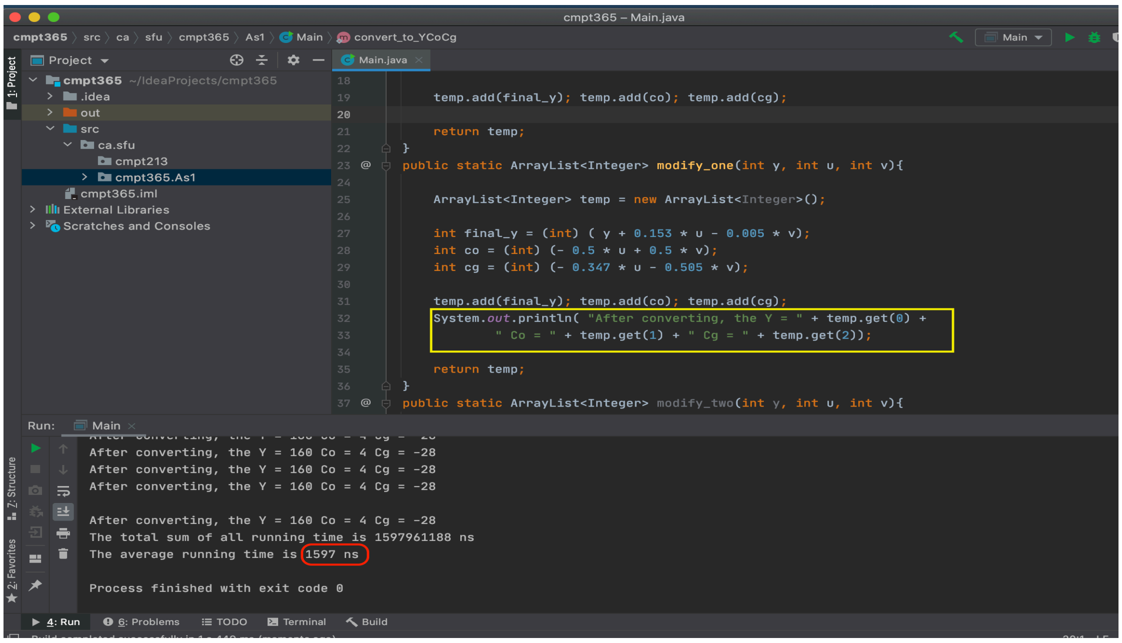
```
long startTime = System.nanoTime();

for (int i = 0; i< 1000000; i++){

result = convert_to_YCoCg(y,u,v);
result = modify_one (y,u,v);
result = modify_two (y,u,v);
}
long endTime = System.nanoTime();
```

The integer result will not change but the running time have a significant difference. I make a table for all data and shown below

Summary:

(All unit is ns)	In the loop	Out the loop
Basic method	1722 ns	66 ns
Merge matrix	94 ns	62 ns
Invariant Y	80 ns	48 ns

For the coding part, all methods include the main function cause the same big-O O(1). For one million times running loop it still cause O(1000000) = O(constant) = O(1). So, they shouldn't have some different. For the machine coding part, if we use less the register (for this program, it will not over the sixteen registers), the program will not cost time to get value from memory or registers. Also, if we have more I/O instructions it will cost more time. Like if we print a result sentence every time in the loop. The running time will increase a lot. Because this operation requires writing a stream of data to a peripheral (screen), the peripheral is in some sense part of the file system. It belongs to an I/O operation. The follow screenshot show the result (I am using the second situation, not print & out loop: 62 ns VS. print & out loop: 1597 ns).



```
        temp.add(final_y); temp.add(co); temp.add(cg);

        return temp;
    }
    public static ArrayList<Integer> modify_one(int y, int u, int v){

        ArrayList<Integer> temp = new ArrayList<Integer>();

        int final_y = (int) ( y + 0.153 * u - 0.005 * v);
        int co = (int) (- 0.5 * u + 0.5 * v);
        int cg = (int) (- 0.347 * u - 0.505 * v);

        temp.add(final_y); temp.add(co); temp.add(cg);
        System.out.println( "After converting, the Y = " + temp.get(0) +
                " Co = " + temp.get(1) + " Cg = " + temp.get(2));

        return temp;
    }
    public static ArrayList<Integer> modify_two(int y, int u, int v){
```

```
After converting, the Y = 160 Co = 4 Cg = -28
After converting, the Y = 160 Co = 4 Cg = -28
After converting, the Y = 160 Co = 4 Cg = -28

After converting, the Y = 160 Co = 4 Cg = -28
The total sum of all running time is 1597961188 ns
The average running time is 1597 ns

Process finished with exit code 0
```

As we can see from the table, the timing position will have a significant impact on the length of the running time. Not changing Y with the out loop timing can have the fastest running time. And the basic method which follows the matrix formula and the timing part stay in the loop will cause the worst case. For the timing position, if we put them inside of the loop. It will increase for every loop, also each time the running time is tiny number. These numbers will have cancelation (epsilon). Secondly, for the cache point of view, each time when we do subtraction between registers the loop will disorganize and change to L1. Therefore, if we want to faster program, it is important to reduce unnecessary steps and change the location of the timing part.

Question 2:

Introduction: Using the NxN matrix as the input picture pixels matrix (each value is [0…255]) and a MxM dither matrix. The range of N is [4…16] and the N is an integer multiple of M. Print out the resultant pictures with halftone printing and with ordered dithering. (Using "0" means not print and "1" means print).

Experiment Design: I want use to input a M value that has the range between 4 and 16. Randomly generating a M*M matrix and each element in the range of [0…M*M +1]. Let program random a coefficient which cannot have coefficient*M > 16 (cannot be smaller than 4) Randomly generating a N*N matrix and each element in the range of [0…255].

Analyzing:

As the example, we let N and M are all equal to 4, so for each of two matrixes, they have total 16 elements each.

For the halftone printing problem:

N matrix:

100	12	182	200
115	22	17	90
239	162	237	64
234	154	36	27

M matrix:

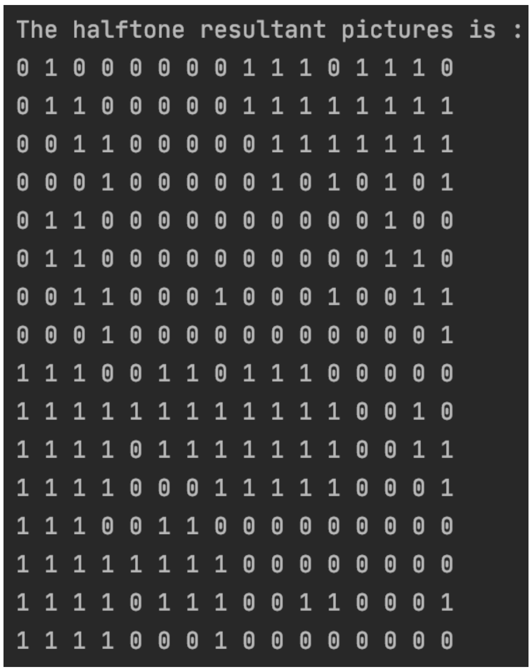
10	4	6	15
7	5	2	8
12	9	1	0
14	11	13	3

After the proportion changed, the N matrix is:

6	0	12	13
7	1	1	6
15	10	15	4
15	10	2	1

All element in this matrix change the range from [0…255] to [0…17]

The halftone printing size should be (N*M) * (N*M) which is 16*16:



```
The halftone resultant pictures is :
0 1 0 0 0 0 0 0 1 1 1 0 1 1 1 0
0 1 1 0 0 0 0 0 1 1 1 1 1 1 1 1
0 0 1 1 0 0 0 0 0 1 1 1 1 1 1 1
0 0 0 1 0 0 0 0 0 1 0 1 0 1 0 1
0 1 1 0 0 0 0 0 0 0 0 0 0 1 0 0
0 1 1 0 0 0 0 0 0 0 0 0 0 1 1 0
0 0 1 1 0 0 0 1 0 0 0 1 0 0 1 1
0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 1
1 1 1 0 0 1 1 0 1 1 1 0 0 0 0 0
1 1 1 1 1 1 1 1 1 1 1 1 0 0 1 0
1 1 1 1 0 1 1 1 1 1 1 1 0 0 1 1
1 1 1 1 0 0 0 1 1 1 1 1 0 0 0 1
1 1 1 0 0 1 1 0 0 0 0 0 0 0 0 0
1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0
1 1 1 1 0 1 1 1 0 0 1 1 0 0 0 1
1 1 1 1 0 0 0 1 0 0 0 0 0 0 0 0
```

For the ordered dithering problem:

N matrix:

173	10	175	195
25	2	166	39
186	219	182	217
195	26	184	247

M matrix:

14	12	9	13
8	4	6	2
11	3	10	5
15	7	0	1

After the proportion changed, the N matrix is:

11	0	11	13
1	0	11	2
12	14	12	14
13	1	12	16

All element in this matrix change the range from [0…255] to [0…17]

The ordered dithering size should still be N * N :

0	0	1	0
0	0	1	0
1	1	1	1
0	0	1	1