Report

# The technique and implement

1. Struct

After searching online, I found using struct in C can easily handle so many pixels. I defined two structs named ‘EveryPixel’ for store every pixel’s RGB value and ‘WholeImage’ for store the input picture’s width(x), height(y) and every pixel’s RGB value.

|  |
| --- |
| **typedef** **struct** {  **unsigned** **char** r, g, b; } EveryPixel;  **typedef** **struct** {  **int** x, y;  EveryPixel \*d; } WholeImage; |

1. fscanf and fread methods

To read two different types of image files, there should be two different ways. For P3 image file, because it was composed for ASCII numbers, I chose fscanf method to read every pixel one by one by using a for loop.

|  |
| --- |
| **int** i; **for** (i = 0; i < (image->x\*image->y); i++) { fscanf(f, **"%d%d%d"**, &image->d[i].r, &image->d[i].g, &image->d[i].b); } |

For P6 image file, using fread method is efficient.

|  |
| --- |
| fread(image->d, 3 \* image->x, image->y, f); |

After reading, all information of input image will be stored in pointer \*image.

3.for loop for calculating

According to the value c was given by the user, I designed a nest for loop to calculate the average colour value in every c\*c area. Then using another for loop to replace the colour values by the average colour in the same area.

4.fprint and fwrite methods

According to the output file option was given by the user, my program need to be able to output P3 and P6 type files. For P3, a for loop and a fprint method can finish the output work.

**for** ( i = 0; i < (img->x\*img->y); i++) {  
 fprintf(fp, **"%d %d %d "**, img->d[i].r, img->d[i].g, img->d[i].b);   
 **if** ((i + 1) % img->x == 0) {  
 fprintf(fp, **"\n"**);  
 }   
}

For P6, I used fwrite method, and there is no need to print “\n” manually.

|  |
| --- |
| **for** (i = 0; i < (img->x\*img->y); i++) {  fwrite(&img->d[i].r, **sizeof**(**unsigned** **char**), 1, fp);  fwrite(&img->d[i].g, **sizeof**(**unsigned** **char**), 1, fp);  fwrite(&img->d[i].b, **sizeof**(**unsigned** **char**), 1, fp); } |

# Optimisations and impact

Since my calculation process is a bit complicated, in the for loop that calculates the average, some variables in the loop initialization statement and the end statement are dynamically changing. Therefore, with my current understanding of OpenMP, it is impossible to perform the parallel computing on the for loop in the calculation process.

However, I manually added parallel sections between statements that can be parallelized.

|  |
| --- |
| #pragma omp parallel #pragma omp sections nowait  { #pragma omp section  allaver = allsumr / (image->x\*image->y); #pragma omp section  allaveg = allsumg / (image->x\*image->y); #pragma omp section  allaveb = allsumb / (image->x\*image->y);   } |

These three statements are respectively calculating the rgb value of the entire picture. I used three sections to let them calculate at the same time. Figure 1 is the result of using CPU and Figure 2 is the result of using OpenMP.



Figure 1. CPU result



Figure 2. OpenMP result

To my surprise, the program that was processed in parallel computing was slower than the original program. I got similar results after many tests. Then I decided to use Visual Studio 2017's own performance profiler to observe the CPU usage of two different modes of operation. In order to rule out the special case, I performed four repeated tests on the CPU and OPENMP modes. In the process of processing the same Dog2048\*2048.ppm and calculating the average value, the CPU average time is about 1 millisecond shorter than the OpenMP average time. Then I checked the CPU usage chart in every test. In CPU mode, each time the resulting graph is similar to Figure 3.

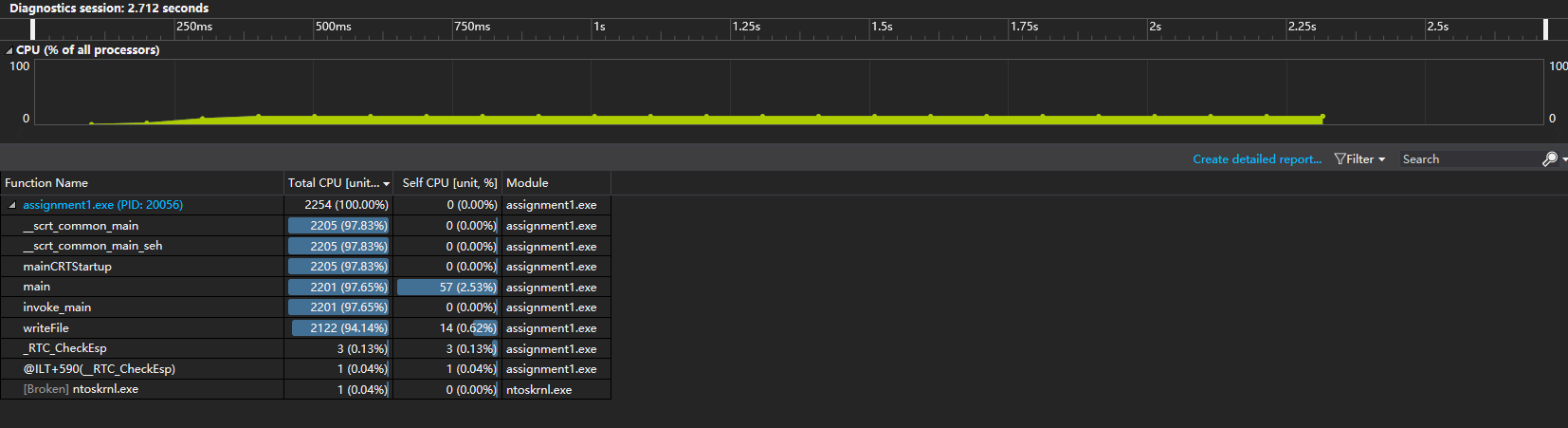


Figure 3. CPU usage in CPU mode

In OpenMP mode, each time the resulting graph is similar to Figure 4.

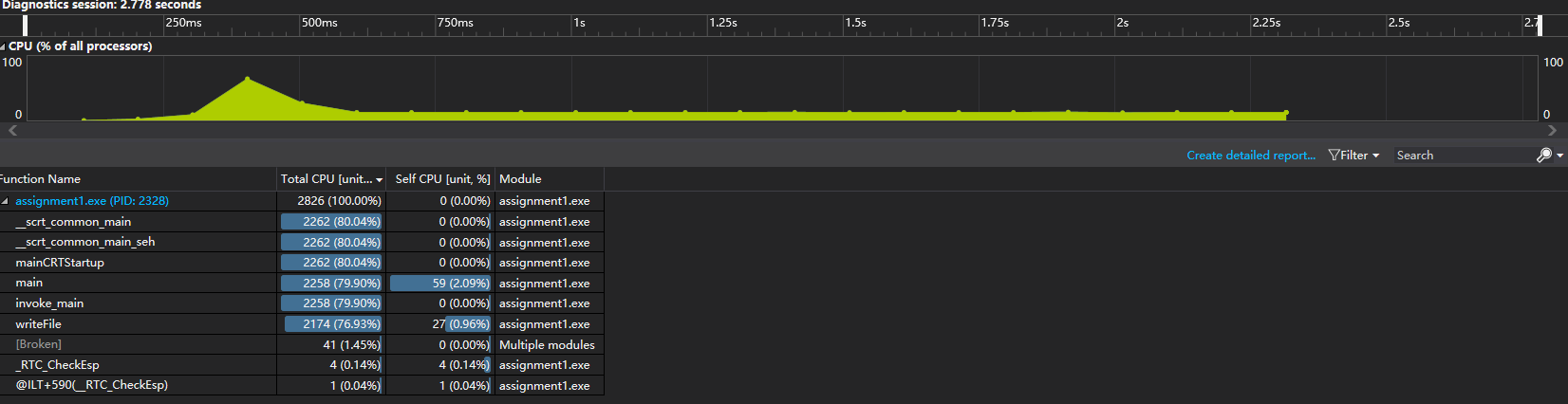


Figure 4. CPU usage in OpenMP mode

From the comparison of the two figures, it is easy to find that the CPU usage in the CPU mode is at a stable value when the entire program is running, and a very obvious peak appears in the resulting graph of the OpenMP mode. There is the program running to the parallel area, the program opened up multiple threads for parallel computing, which has more CPU usage.

Then I tried to add similar parallel calculation sections in other areas, but the result is that the CPU peak time increases and the calculation speed are slower.

# Conclusion

It can be seen from the above test results that the use of OpenMP does improve the utilization of the CPU in the specified parallel area. However, my method of setting a parallel calculation may have problems, and it leads to an increase in calculation time and does not play a role in optimizing the program.

After searching some related research reports and forum articles, I suspect that the reason for the negative optimization of the parallel region may be that I did not pay attention to the concurrency of the instructions in the process of writing the program, so that the variables in the instructions in the parallel section are associated with values in some statements outside the parallel section. Even if the parallel section is set, the variables in the parallel section are waiting for the value of the statement outside the parallel section to pass.

To solve this problem, I think that I need to modify the way I calculate the average and save the results. In particular, the for loop that gets each value when calculating the average value needs to make the loop initial value and the end condition static so that the for loop can perform OpenMP parallel optimization.

# 