**Optimization Report**

**Implementation**

In this program, the mosaic technique has been applied on different images with different size. The mosaic block will be created, then it slips over the image with calculating the average value of RGB in the block. The size of the mosaic block, c, has some specific rules when program running. The first, the c should be less than both size of width and height. The second, the c should be any positive power of number 2, otherwise the c would be processed to the nearest valid number, e.g. c = 3 will be processed to c = 2. The average RGB value of the whole image will be calculated by sum up the average RGB value of each mosaic block and divided by the number of pixels in image. The program could run under two optional methods. The one is under CPU with single process, another one is under CPU with multiple processes.

In the code part, the program could be divided into four parts, reading, pre-processing, calculation, outputting. The program starts at the main function which could receive the argument from the execution command. Then the image data will be read from the input file and be pre-processed. The file will be opened in binary mode and then be processed and stored into 2-D integer array. This process is implemented in the function *read\_data*(). The data in 2-D array will be divided into different blocks according to the size of the mosaic block, meanwhile, if c is not the factor of the number of width or height, the block at the edge will be clipped. All situations have been considered in the program. Then the average RGB value would be calculated and replace each original pixel RGB value in the mosaic block. This process is implemented in the optional function *cpu\_cal()* or *openmp\_cal()*. At the final the function *output()* will write the data after calculation into the specific file with optional format (Binary or Plain Text).

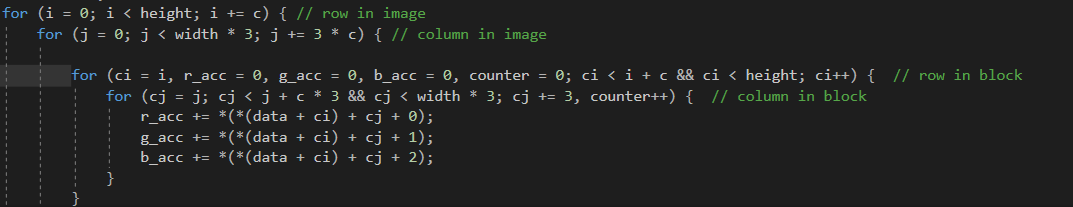
For the choice of data structure, two kinds of data structure have been considered to store the image data (1-D array or 2-D array). Using 1-D array is convenient to store the data just read from the file but when it comes to the calculation part it would be too abstract for the programmer because the index of the array would be complex and would be hard to map the data onto the mosaic block. The final choice is using 2-D array because it is more understandable for programmer, though using this structure requires to do more pre-process. The function *malloc()* is broadly used in the program e.g. allocating the memory for 2-D array to store the image data. Using *malloc()* is a better method than using array declaring (e.g. *array[len]*), because sometime the size of array is unknown during the compiling.

**Optimization**

To get the optimal program, some optimization approaches have to be applied on the program. These approaches are mainly focused on the compute bound and memory bound. In fact, in this program, the compute bound are more important than memory bounds, because processing a high quality image (2048x2048) requiring the memory will not overflow on the most common computer nowadays. So in this program, the compute bound will be optimized at first.

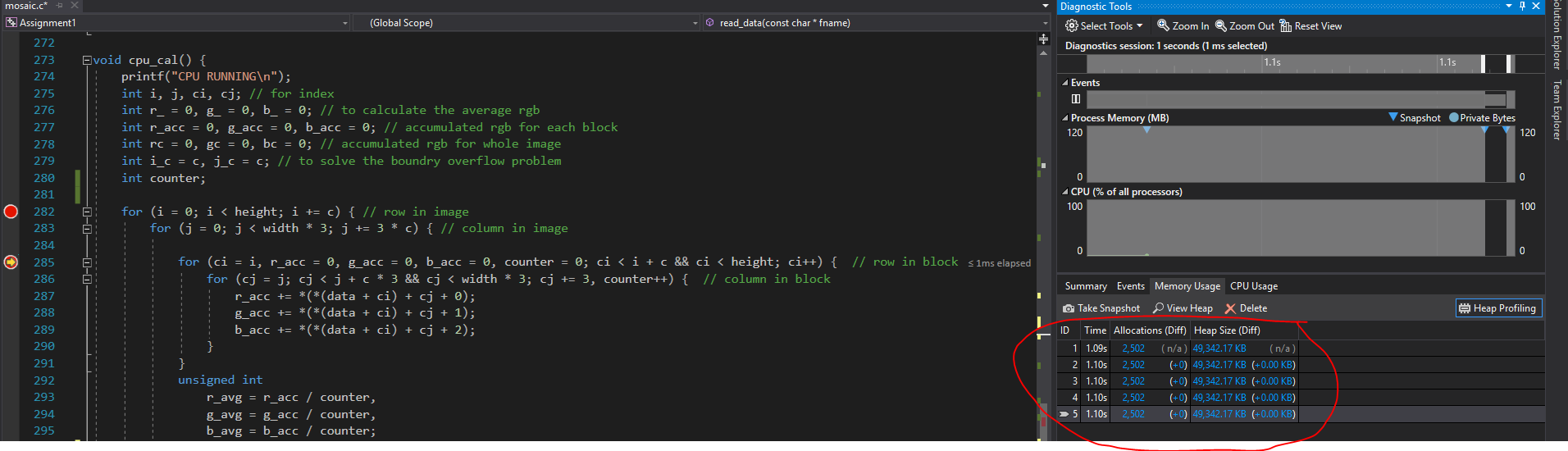
This program is compiled and debugged by the Microsoft Visual Studio 2017. The different optimization version will be managed by Git. The optimization part will be divided into two parts, the first part is aim to improve the performance when using CUP with single process, the second part is aim to improve the performance when using OpenMP to implement calculation on CPU with multiple processes. To show what has been improved, all the debug information and benchmark are based on calculating the image which size is 2048x2048.

The first, CUP with single process, generally cost about 60ms to calculate the average value of whole image and replace the original value with average value. The time consuming is acceptable, through there are four loops connected, shown as following image.



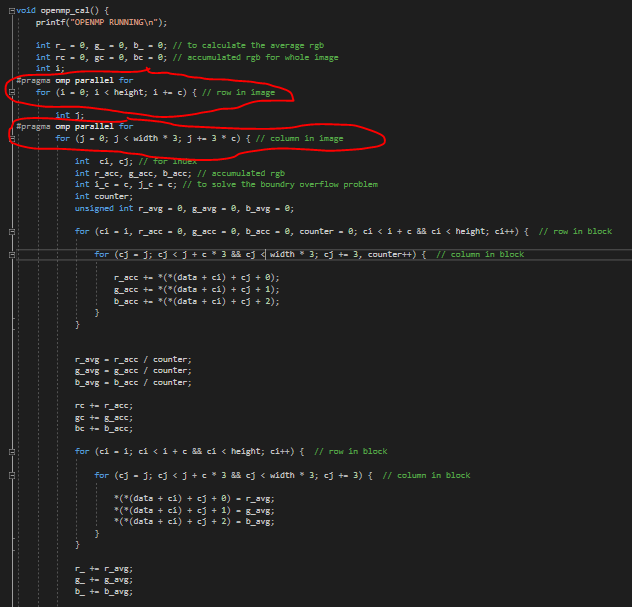
However, the time of running this loop is fixed, even though the size of both the image and the mosaic block is involved. That is because of that the cost of first two loops depends on how many mosaic blocks (width / c \* height / c) will be included in the image, the cost of the inner two loops depends on the size of mosaic block (c). If c increases, the value of width / c \* height / c will decrease, so it will be kept in a fix complexity. Obviously, this thread traverse whole image with taking block as unit. There is another approach which only need three loops have been tried in the implementation, the performance still keep in the same with using four loops, because no matter four loops or three loops are used, it always take one block as moving once unit as well.

For the memory bound optimization in this part, no more heap memory will be applied and it only need some integer variables. Following image shows heap memory has no change in the loops. So, there is no obvious optimization in here.



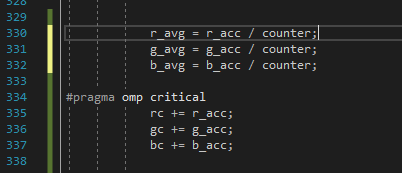
Then, the part of CPU with multiple process is based on the single process, which means they have the same code logic. The code of this part is based on the part of single process and change a little. The main purpose of optimizing this part is saving time and improve the efficiency of use of CPU. Meanwhile, the difficulty of optimization is how to manage the variable sharing or not sharing.

In this optimization, different strategies have be compared and the best one has been applied on the program, so there are a few vision of optimization. In the initial version, the strategies are adding some OpenMP directives before the loop and reconstructing a part of code. The first version code shown in the following image.

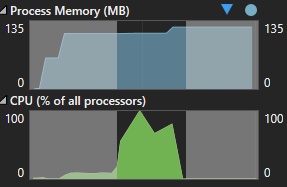


Comparing the code with the part of single process, this code has the declaration of OpenMP and some variables outside the loop have been moved to be inside the loop to become the private variable. This code can run in the most of situation and it actually has the better performance than single process. However, when c comes to small value (e.g. 1 or 2), this program cannot solve the average value correctly. Debugging the program, it could be found that because of the accumulation existing in the parallel loops the threads will share the accumulation variable and this variable would be changed simultaneously by the different threads.

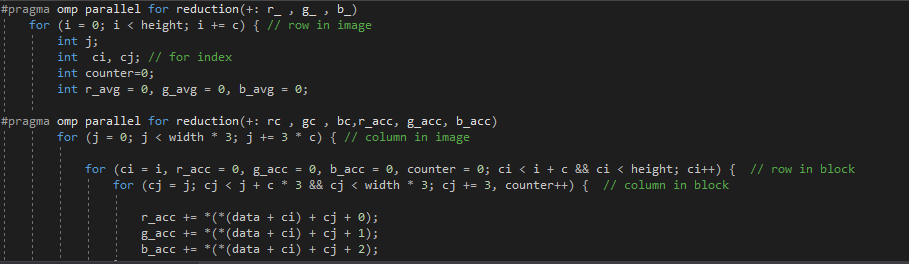
So, the second version would focus on solving this variable sharing problem. The OpenMP directive, *critical*, will be used, shown in the following image.



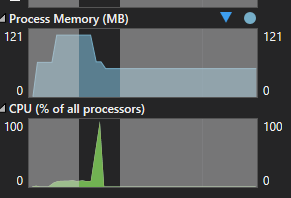
Based on the first version, using *critical* can solve the variable sharing problem, but the performance is lower than single process. The reason might be that the *critical* create the barrier before the accumulation part and all threads have to wait till all tasks done before the accumulation so that would cost time and need to distribute the memory for each thread. The following image shows the memory usage and CPU usage in this case. This is not an effective approach, so the third version comes to solve the problem.



In the third version, based on the first version, the OpenMP directive, *reduction*, will be used and combined with the loop. Following image will show the use of *reduction*.



Using *reduction* can not only improve the performance of the program but also can always solve the correct value. The directive, *reduction*, can copy the reduction variable for each thread and combine the variable at the end of the structured code block then save it to the variable in the master thread. The usage of CPU and Memory in this case is shown in following image.



It is reasonable and expected. Running this multiple process program to process a piece of Image (2048x2048) will cost about 20ms which is much quicker than running the single process (which cost about 60~80s). So, this is the final version for using of OpenMP.