# The report of part2

## 1. Implementation of code

Successfully generated 3 completed versions of the code during the implementation of the function. At the same time, the other 1 versions were incomplete which is more reasonable on distributing threads and which is a better algorithm.

### 1.1 version 1

The first version aims to parallelise each input pixel. The size of the block set to c^2 so that threads will within the same block and responsible for calculating a mosaic. So the size of the grid is up to the size of mosaic (c witch input from the user).

First, I get data of r,g,b  from PPMrgb structure and store them into three arrays respectively. And these three arrays be passed into the kernel. In the kernel, I index every pixel and add them within every block by atomicAdd(). There are two drawbacks to this method. However, the method cannot calculate the overlarge inputting cell size(c). Because the maximum thread is 1024. In this method, each block will be divided to c^2  threads so that the maximum c is 32. Thus, the limitation of this method is the size of the picture is limited. On another hand, each thread is responsible for a pixel and is not responsible for any calculations, and it only computes the sum of the values in all threads in a block (i.e., the RGB of each pixel). Therefore, this method does not do parallel calculations to complete the addition in the mosaic, resulting in no higher computational speed with parallel computation.


*Figure 1.1*

I have tried some optimization methods on this method. On the basis of this, c is set to be larger than 32, and a mosaic unit is allocated to a plurality of blocks for calculation. For example, when c is equal to 64, 4 blocks compute a mosaic; when c is equal to 128, 16 blocks compute a mosaic. Then, you get a general formula. When c is greater than 32, 2^(c/32), blocks are responsible for computing a mosaic cell. Then, it can handle the case where require c bigger than 32.

## 1.2 version 2

Version 2 is the basic version that implements all the features. Version 2 parallelise each mosaic cell that can solve the problem of the limitation of the number of threads in the Version 1 block. Similarly, Version 2 passes RGB data into the GPU through an array. As Figure 2.1 shows that each block is responsible for calculating the superposition operation in each mosaic unit, and there is only one thread in one block. That is, this thread in each block does the calculation for each mosaic.

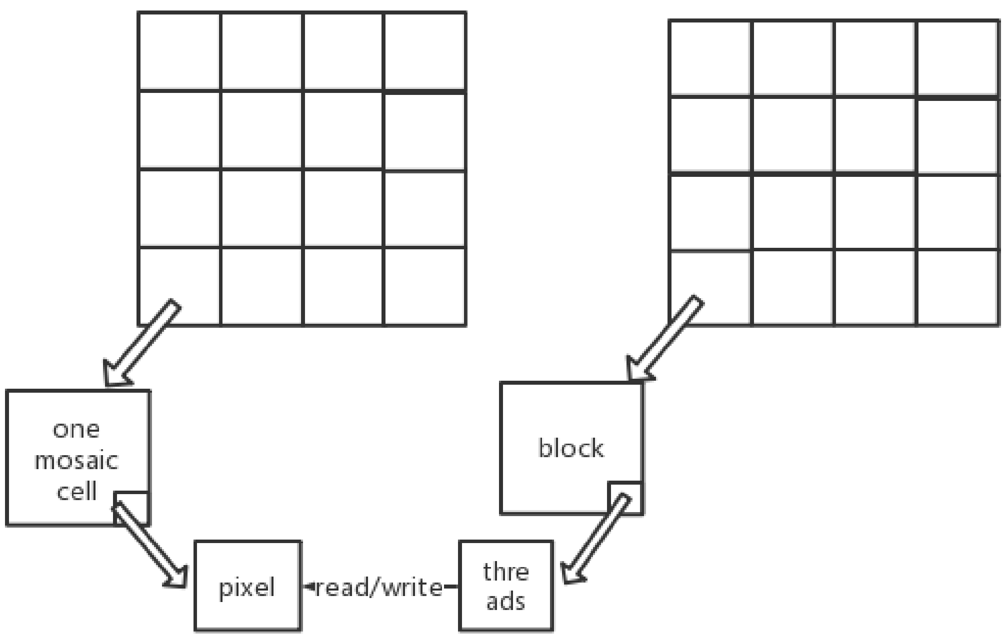
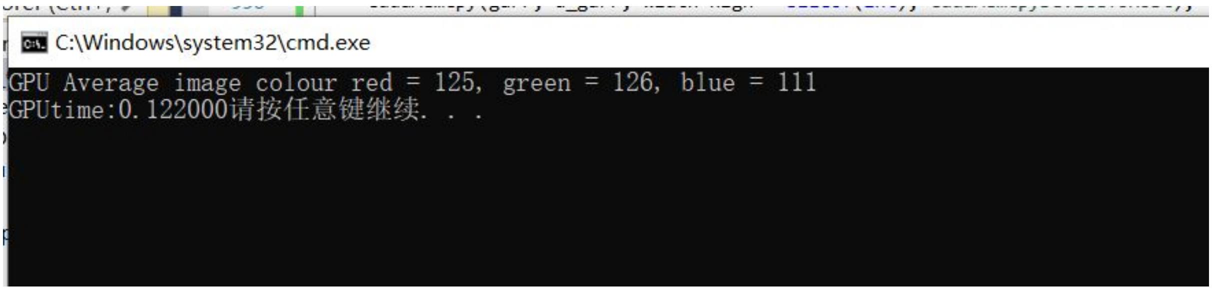



*Figure 1.2.1 the structure of Version 2*

In this version, the smaller the c value, the shorter the runtime is as shown in Figure c for 256 and 64 respectively (Figure 2.2 a and b). The reason is that the larger the c value, the more computational amount of each block (ie, each thread), so the running time will increase.



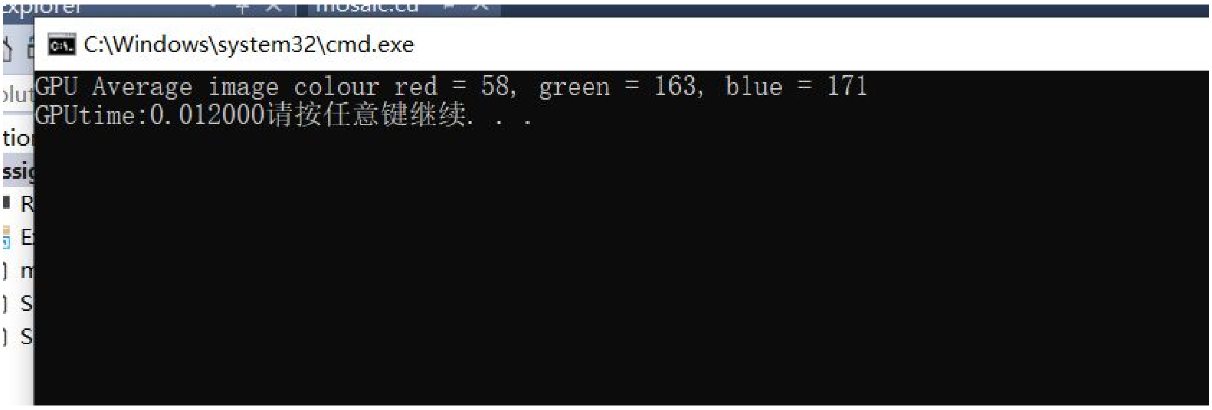

a


b

*Figure 1.2.2 a) when c=256, the time of GPU running is 0.122000s; b) when c=64, the time of GPU running is 0.109000s.*

However, for the case of the same c value for different pictures, the smaller the picture size, the shorter the running time. As shown in Figure 2.3 a and b, c is 64 and the pictures are 2048 * 2048 and 512 * 512 respectively.


a




b

*Figure 1.2.3 In case of c=64, a) when the picture size is 2048*2048, the time of GPU running is 0.109000s; b) when picture size is 512*512, the time of GPU running is 0.0120000s.*

## 1.3 version 3

I think version 3 is the best version of all the versions I can implement. Based on Version 2, Version 3 changes the way of representing RGB data. In version 2, a one-dimensional array is used. In this version, Structures of Arrays is used instead. Compared with Version 2, it greatly improves the running speed and shortens the running time.

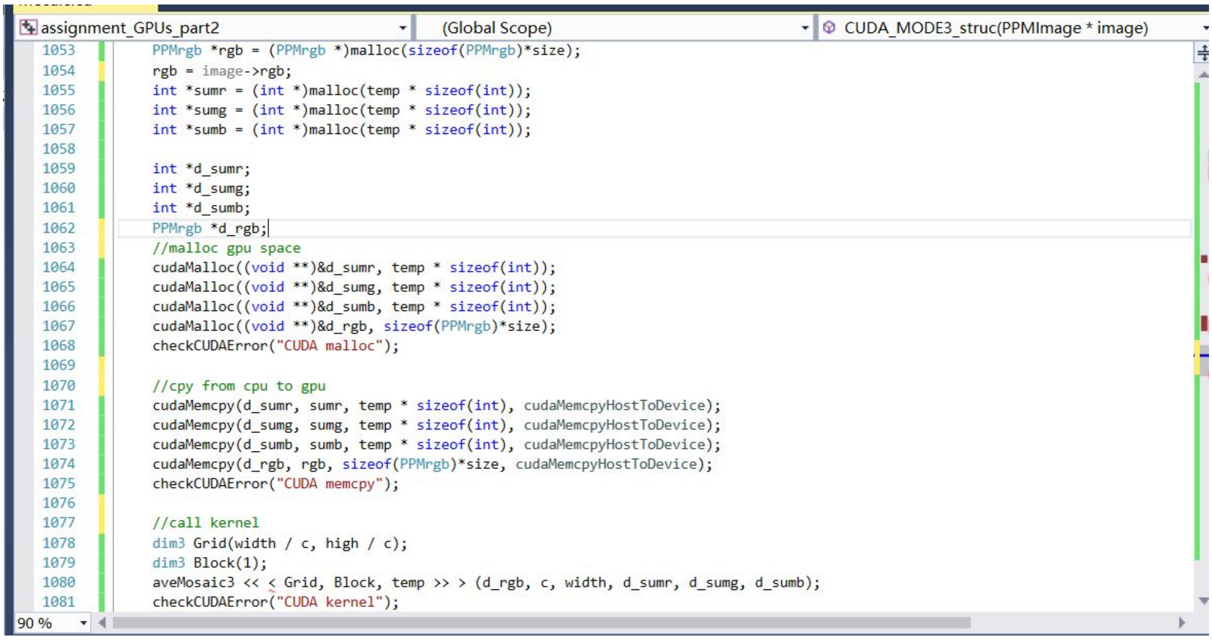



*Figure 1.3.1 code of Structures of Array*

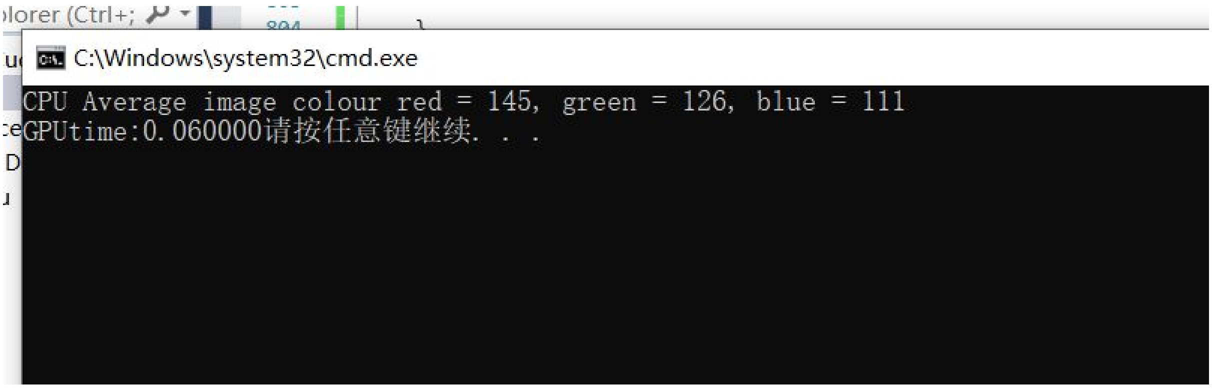


*Figure 1.3.2 runtime of  v3*

Compared with the version 2, the speed has been improved.

## 1.4 version 4

Version 4 is based on the optimization of Version 2. This is an incomplete version. The way to transfer data is the same as that of version 2. It is also to transfer and represent RGB data into GPU by an array. Each block is responsible for calculating the superposition operation in each mosaic unit. The difference is that there are c threads in a block to do the superposition operation instead of one block and only one thread to do the operation. Each thread in each block is responsible for the superposition of a row in the mosaic cell. After all the threads in the same block have completed the operation, perform another superposition operation in each block (Figure 1.4.1).

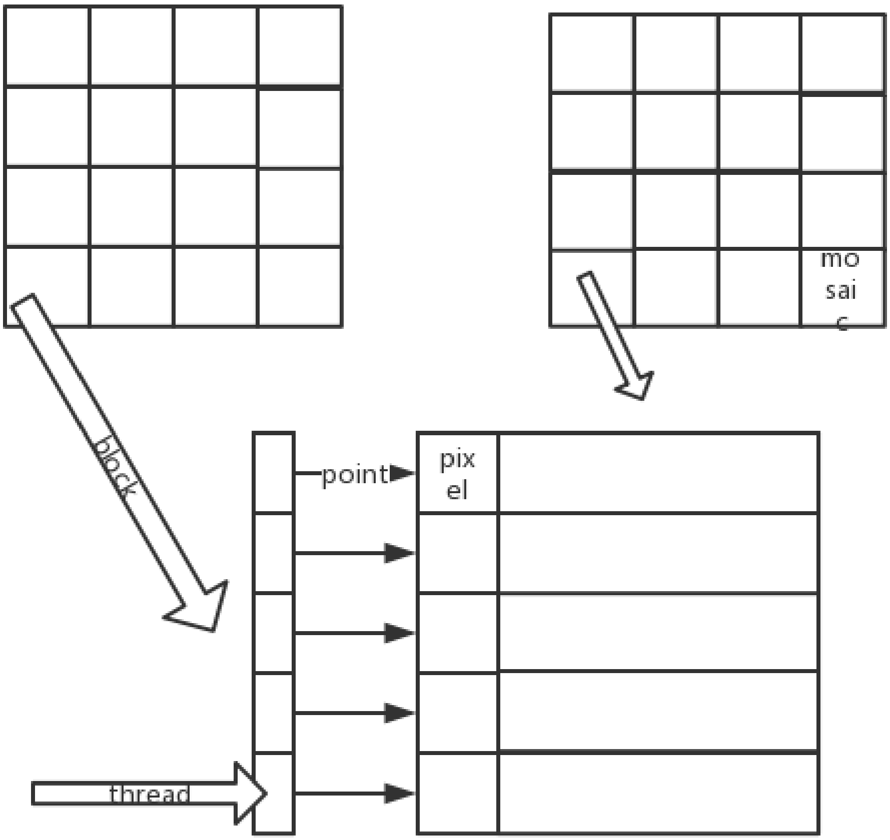



*Figure 1.4.1 the structure of Vision 4*

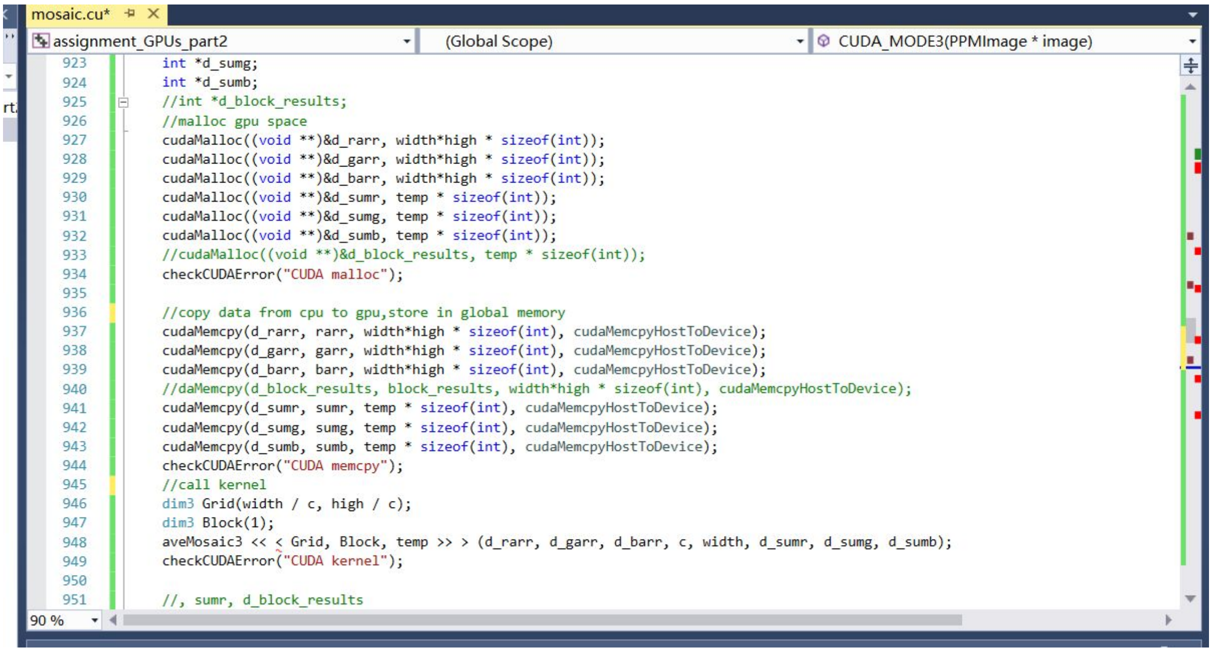
```
    int sum = 0;
    __shared__ int sumRed;
    __shared__ int sumGreen;
    __shared__ int sumBlue;
    __shared__ int aveRed;
    __shared__ int aveGreen;
    __shared__ int aveBlue;

    //int index = threadIdx.x + threadIdx.y * blockDim.x * c  + blockIdx.x * c + blockIdx.y * c  * width;
    int index = threadIdx.x + threadIdx.y*width + blockIdx.x*c + blockIdx.y * c  * width;

    for (int i = 0; i < c; i++) {
        sumRed = sumRed + image->rgb[index + i].red;
        sumGreen = sumGreen + image->rgb[index + i].green;
        sumBlue = sumBlue + image->rgb[index + i].blue;
    }

    if (threadIdx.x == 0) {
        aveRed = sumRed / (c*c);
        aveGreen = sumGreen / (c*c);
        aveBlue = sumBlue / (c*c);
    }

    for (int i = 0; i < c; i++) {
        image->rgb[index + i].red = aveRed;
        image->rgb[index + i].green = aveGreen;
        image->rgb[index + i].blue = aveBlue;
    }
```

*Figure 1.4.2*

# 2. Optimisations

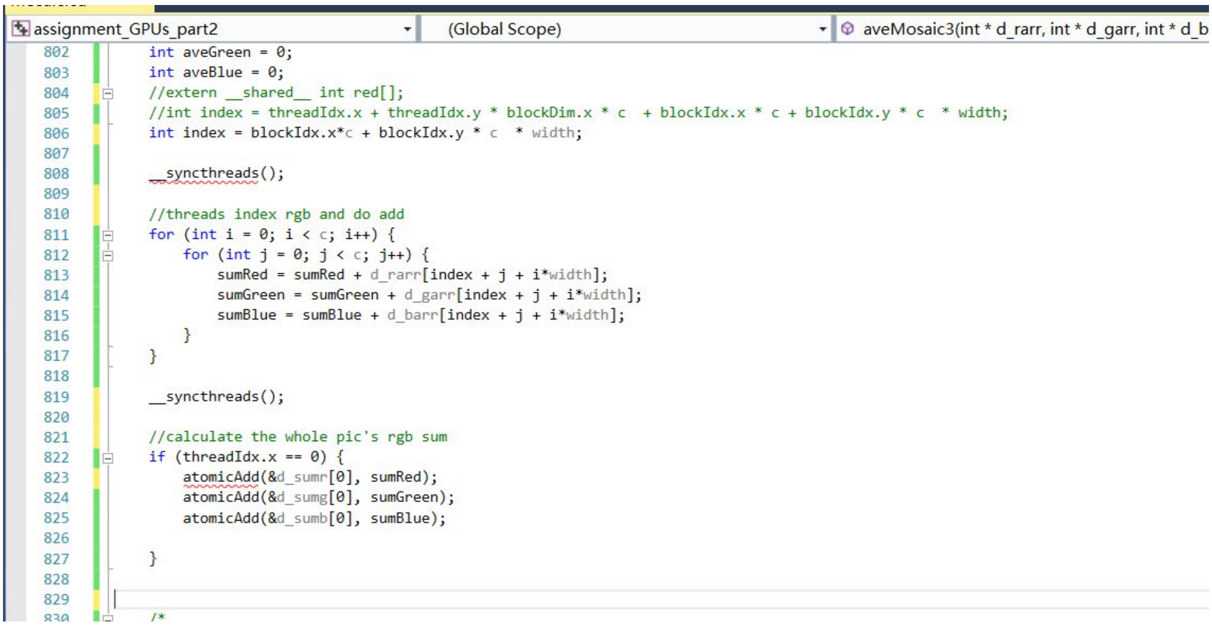
## 2.1 Different methods to layout or represent your data

there are two different methods to layout the data of rgb which are array and structures of arrays.which are correspond to Version 2 and Version 3 respectively. The speed of operation of using structures of arrays is greatly improved. The code and results are compared as follows:




```
    int *d_sumg;
    int *d_sumb;
    //int *d_block_results;
    //malloc gpu space
    cudaMalloc((void **)&d_rarr, width*high * sizeof(int));
    cudaMalloc((void **)&d_garr, width*high * sizeof(int));
    cudaMalloc((void **)&d_barr, width*high * sizeof(int));
    cudaMalloc((void **)&d_sumr, temp * sizeof(int));
    cudaMalloc((void **)&d_sumg, temp * sizeof(int));
    cudaMalloc((void **)&d_sumb, temp * sizeof(int));
    //cudaMalloc((void **)&d_block_results, temp * sizeof(int));
    checkCUDAError("CUDA malloc");

    //copy data from cpu to gpu,store in global memory
    cudaMemcpy(d_rarr, rarr, width*high * sizeof(int), cudaMemcpyHostToDevice);
    cudaMemcpy(d_garr, garr, width*high * sizeof(int), cudaMemcpyHostToDevice);
    cudaMemcpy(d_barr, barr, width*high * sizeof(int), cudaMemcpyHostToDevice);
    //daMemcpy(d_block_results, block_results, width*high * sizeof(int), cudaMemcpyHostToDevice);
    cudaMemcpy(d_sumr, sumr, temp * sizeof(int), cudaMemcpyHostToDevice);
    cudaMemcpy(d_sumg, sumg, temp * sizeof(int), cudaMemcpyHostToDevice);
    cudaMemcpy(d_sumb, sumb, temp * sizeof(int), cudaMemcpyHostToDevice);
    checkCUDAError("CUDA memcpy");
    //call kernel
    dim3 Grid(width / c, high / c);
    dim3 Block(1);
    aveMosaic3 << < Grid, Block, temp >> > (d_rarr, d_garr, d_barr, c, width, d_sumr, d_sumg, d_sumb);
    checkCUDAError("CUDA kernel");

    //, sumr, d_block_results
```

*Figure 2.1.1 code of using array 1*




```
    int aveGreen = 0;
    int aveBlue = 0;
    //extern __shared__ int red[];
    //int index = threadIdx.x + threadIdx.y * blockDim.x * c  + blockIdx.x * c + blockIdx.y * c  * width;
    int index = blockIdx.x*c + blockIdx.y * c  * width;

    __syncthreads();

    //threads index rgb and do add
    for (int i = 0; i < c; i++) {
        for (int j = 0; j < c; j++) {
            sumRed = sumRed + d_rarr[index + j + i*width];
            sumGreen = sumGreen + d_garr[index + j + i*width];
            sumBlue = sumBlue + d_barr[index + j + i*width];
        }
    }

    __syncthreads();

    //calculate the whole pic's rgb sum
    if (threadIdx.x == 0) {
        atomicAdd(&d_sumr[0], sumRed);
        atomicAdd(&d_sumg[0], sumGreen);
        atomicAdd(&d_sumb[0], sumBlue);

    }

    /*
```

*Figure 2.1.2 code of using array 2*

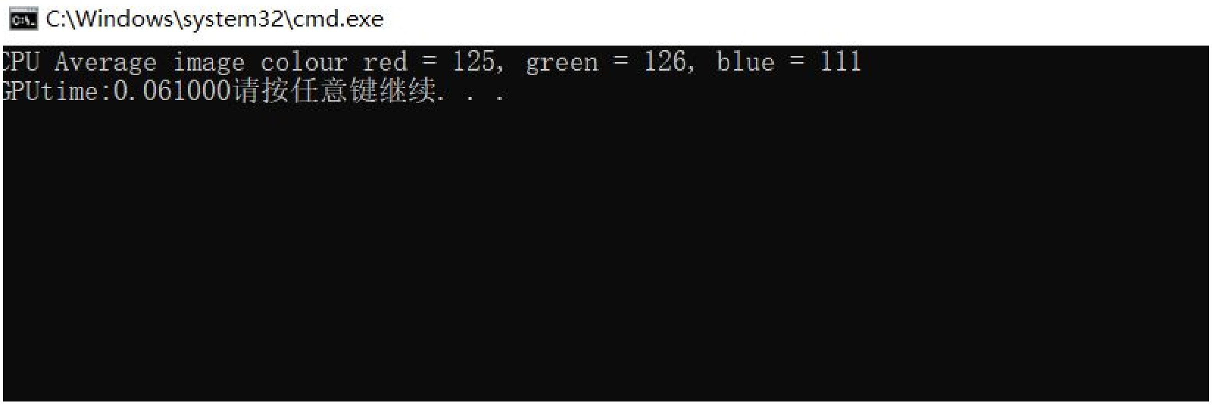
```
int *sumr = (int *)malloc(temp * sizeof(int));
int *sumg = (int *)malloc(temp * sizeof(int));
int *sumb = (int *)malloc(temp * sizeof(int));

int *d_sumr;
int *d_sumg;
int *d_sumb;
PPMrgb *d_rgb;
//malloc gpu space
cudaMalloc((void **)&d_sumr, temp * sizeof(int));
cudaMalloc((void **)&d_sumg, temp * sizeof(int));
cudaMalloc((void **)&d_sumb, temp * sizeof(int));
cudaMalloc((void **)&d_rgb, sizeof(PPMrgb)*size);
checkCUDAError("CUDA malloc");

//cpy from cpu to gpu
cudaMemcpy(d_sumr, sumr, temp * sizeof(int), cudaMemcpyHostToDevice);
cudaMemcpy(d_sumg, sumg, temp * sizeof(int), cudaMemcpyHostToDevice);
cudaMemcpy(d_sumb, sumb, temp * sizeof(int), cudaMemcpyHostToDevice);
cudaMemcpy(d_rgb, rgb, sizeof(PPMrgb)*size, cudaMemcpyHostToDevice);
checkCUDAError("CUDA memcpy");

//call kernel
dim3 Grid(width / c, high / c);
dim3 Block(1);
aveMosaic3 << < Grid, Block, temp >> > (d_rgb, c, width, d_sumr, d_sumg, d_sumb);
checkCUDAError("CUDA kernel");
```

*Figure 2.1.3 code of using structures of arrays 1*

```
int aveGreen = 0;
int aveBlue = 0;

int index = blockIdx.x*c + blockIdx.y * c  * width;

//threads index rgb and do add
for (int i = 0; i < c; i++) {
    for (int j = 0; j < c; j++) {
        sumRed = sumRed + d_rgb[index + j + i*width].red;
        sumGreen = sumGreen + d_rgb[index + j + i*width].green;
        sumBlue = sumBlue + d_rgb[index + j + i*width].blue;
    }
}

__syncthreads();
//calculate the whole pic's rgb sum
if (threadIdx.x == 0) {
    atomicAdd(&d_sumr[0], sumRed);
    atomicAdd(&d_sumg[0], sumGreen);
    atomicAdd(&d_sumb[0], sumBlue);

}

__syncthreads();
//calculate the average rgb of every mosaic
aveRed = sumRed / div;
aveGreen = sumGreen / div;
aveBlue = sumBlue / div;
```

*Figure 2.1.4 code of using structures of arrays 2*


a




b

*Figure 2.1.5 In case of c=64 and same picture a) when using arrays, the time of GPU running is 0.109000s; b) when using structures of arrays, the time of GPU running is 0.062000s.*

## 2.2 The use of various GPU memory caches

**shared memory**

I try to use shared memory to store the sum of every mosaic cell to reduce the number of global memory reads. because found the sum  of every mosaic cell value to be used too many times when averaging and summing the whole picture.

```
__global__ void aveMosaic3_struc(PPMrgb *d_rgb, int c, int width, int *d_sumr, int *d_sumg, int *d_sumb) {


    int div = d_c*d_c;
    __shared__ int sumRed;
    __shared__ int sumGreen;
    __shared__ int sumBlue;
    int aveRed = 0;
    int aveGreen = 0;
    int aveBlue = 0;

    int index = blockIdx.x*c + blockIdx.y * c  * width;
```

*Figure 2.2.1 code of using shared memory*

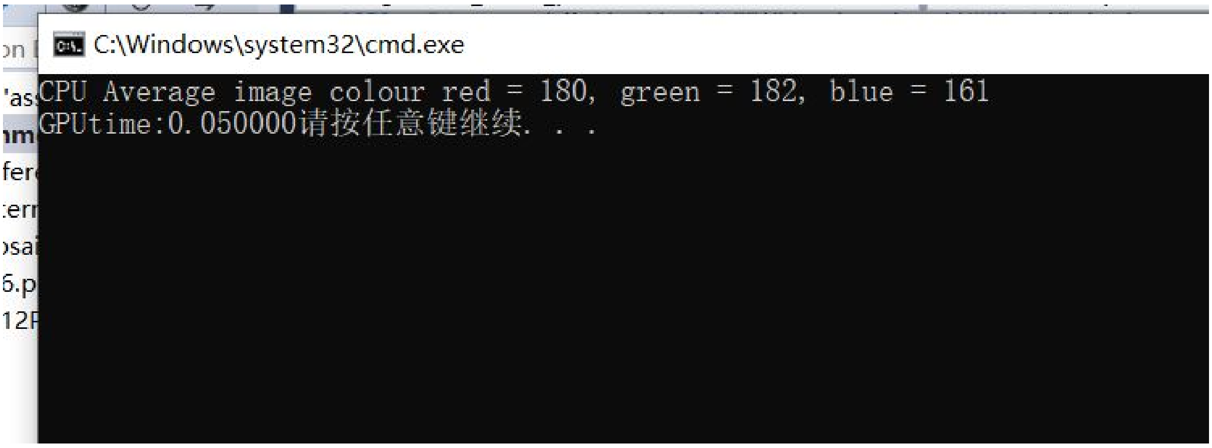



*Figure 2.2.1 runtime of using shared memory*

**constant memory:**

there are three constants required in the calculation process of gpu which are value c and the width of picture. so I use constant memory to store them.The speed of operation is improved a bit.

```
char *ppmfileout;
__constant__ int d_c;
__constant__ int d_WIDTH;
```

*Figure 2.1.4 code of using constant memory*

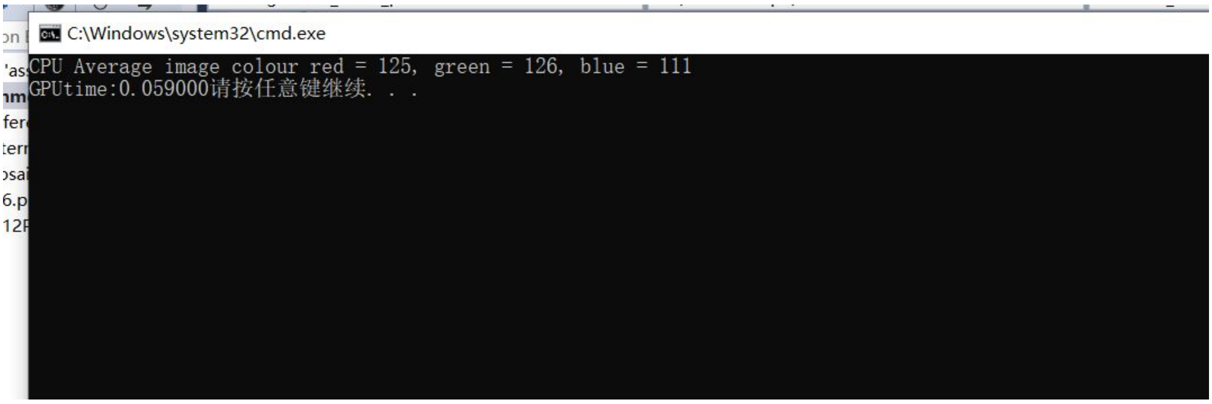



*Figure 2.2.1 runtime of using constant memory*

## 2.3 some GPU optimisations for improving the performance

**syncthreads:**

synchronize before calculating the average value to make sure that a mosaic cell completes all internal superposition operations.

```
for (int i = 0; i < d_c; i++) {
    for (int j = 0; j < d_c; j++) {
        sumRed = sumRed + d_rgb[index + j + i*d_WIDTH].red;
        sumGreen = sumGreen + d_rgb[index + j + i*d_WIDTH].green;
        sumBlue = sumBlue + d_rgb[index + j + i*d_WIDTH].blue;
    }
}

__syncthreads();
//calculate the whole pic's rgb sum
if (threadIdx.x == 0) {
    atomicAdd(&d_sumr[0], sumRed);
    atomicAdd(&d_sumg[0], sumGreen);
    atomicAdd(&d_sumb[0], sumBlue);

}

__syncthreads();
//calculate the average rgb of every mosaic
aveRed = sumRed / div;
aveGreen = sumGreen / div;
aveBlue = sumBlue / div;
```

**Reduction:**

In the method of calculating the sum RGB of the whole picture, I use atomicAdd function. I have a thought that i can calculate it using reduction, so that the continuous block can be released, but the effect is not significant in the current version, because each block has only one thread, even It is not a discontinuous block in the thread to complete the current operation can also be released.

The effect on v4 should be significant, because there are c threads in a block in v4. When c is larger, the superimposed operation after reduction optimization will release continuous threads.

## 3. summary

During the completion of this assignment, I first corrected the function of the cpu part and simply optimized the openmp part. Inspired by the OPENMP part, I first figure out version version 1 and version 2, and finally I got version 3 which is can implemented all the function through continuous optimization as my assignment version. I still have some better method which i think it’s will be ,ore rational use of resources in the gpu calculation process, like version 4, didn’t completed and need to debug.