

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.

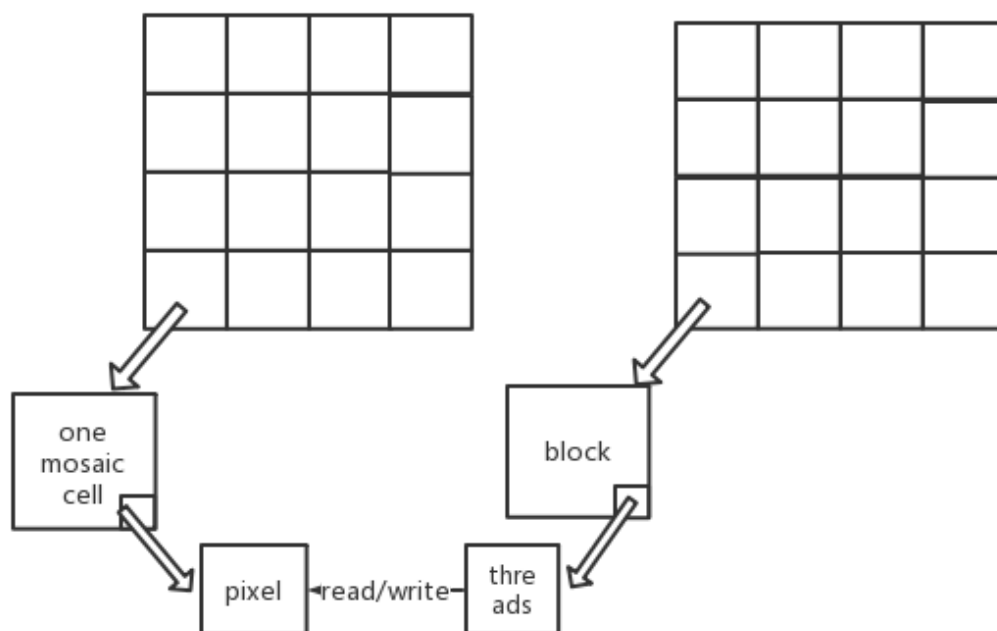
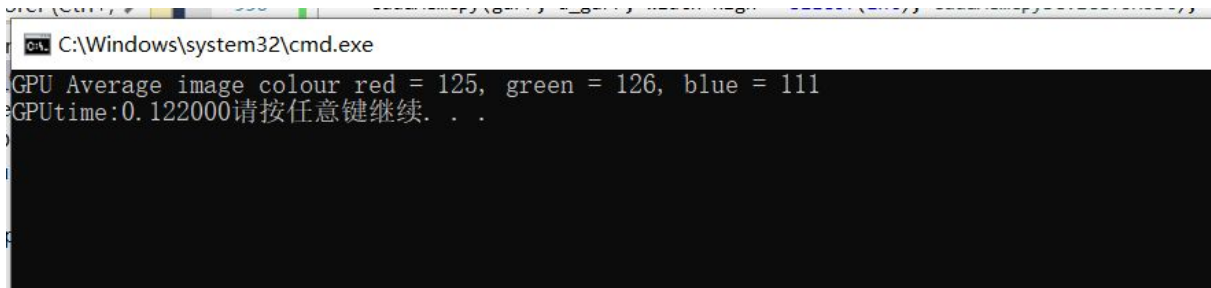


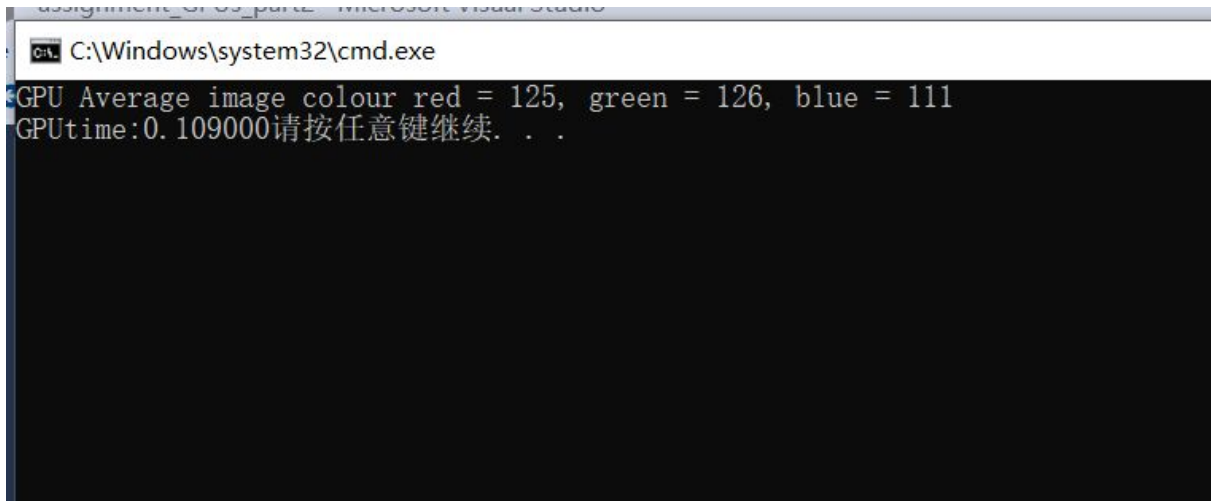
Figure 1.2.1 the structure of Version 2

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.



```
C:\Windows\system32\cmd.exe
GPU Average image colour red = 125, green = 126, blue = 111
GPUtime:0.122000请按任意键继续. . .
```

a



```
C:\Windows\system32\cmd.exe
GPU Average image colour red = 125, green = 126, blue = 111
GPUtime:0.109000请按任意键继续. . .
```

b

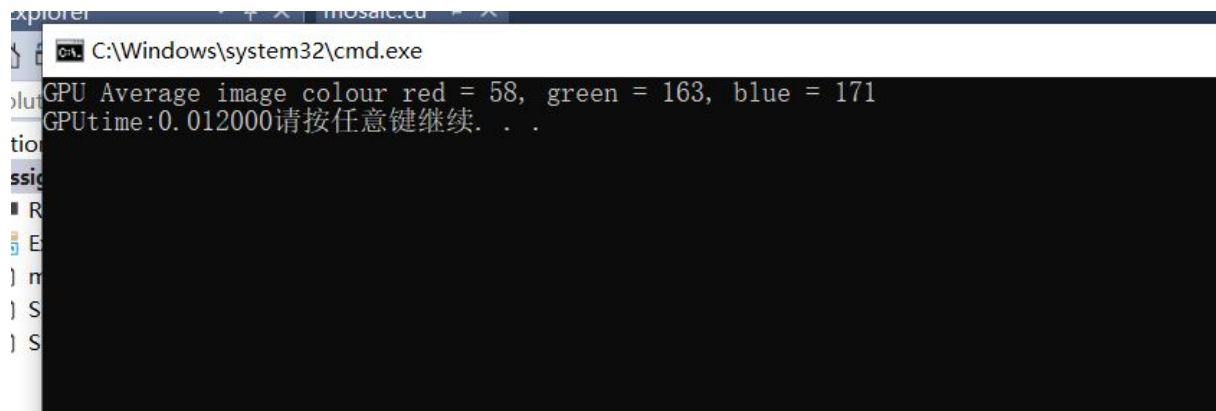
Figure 1.2.2 a) when $c=256$, the time of GPU running is 0.122000s; b) when $c=64$, the time of GPU running is 0.109000s.

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.



```
C:\Windows\system32\cmd.exe
GPU Average image colour red = 125, green = 126, blue = 111
GPUtime:0.109000请按任意键继续. . .
```

a



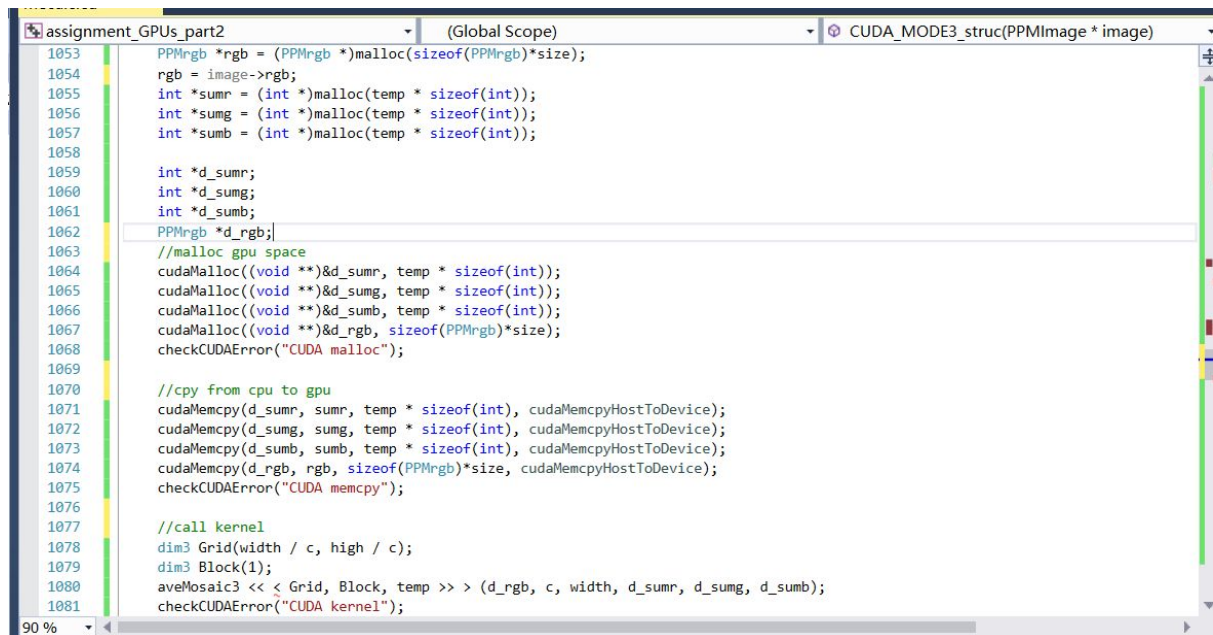
```
C:\Windows\system32\cmd.exe
GPU Average image colour red = 58, green = 163, blue = 171
GPUtime:0.012000请按任意键继续. . .
```

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 0.012000s.

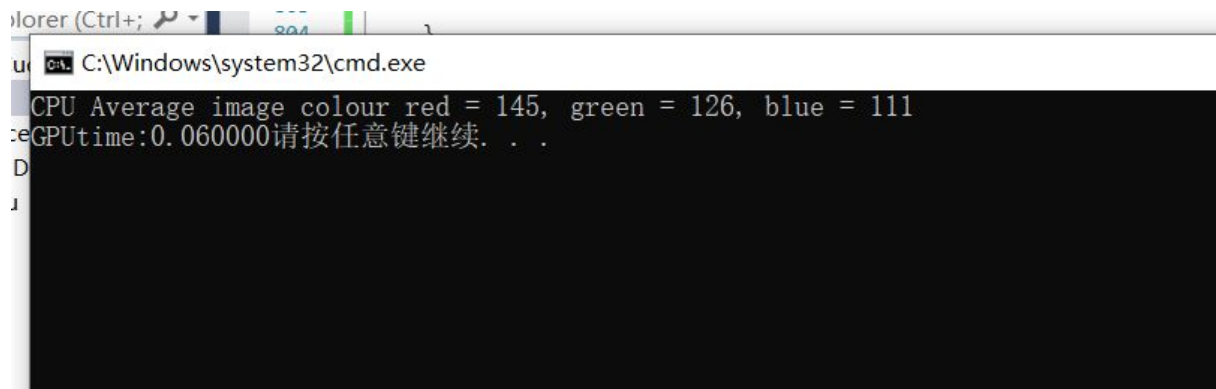
1.3 version 3

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.



```
assignment_GPUs_part2 (Global Scope) CUDA_MODE3_struct(PPMImage * image)
1053 PPMrgb *rgb = (PPMrgb *)malloc(sizeof(PPMrgb)*size);
1054 rgb = image->rgb;
1055 int *sumr = (int *)malloc(temp * sizeof(int));
1056 int *sumg = (int *)malloc(temp * sizeof(int));
1057 int *sumb = (int *)malloc(temp * sizeof(int));
1058
1059 int *d_sumr;
1060 int *d_sumg;
1061 int *d_sumb;
1062 PPMrgb *d_rgb;
1063 //malloc gpu space
1064 cudaMalloc((void **)&d_sumr, temp * sizeof(int));
1065 cudaMalloc((void **)&d_sumg, temp * sizeof(int));
1066 cudaMalloc((void **)&d_sumb, temp * sizeof(int));
1067 cudaMalloc((void **)&d_rgb, sizeof(PPMrgb)*size);
1068 checkCUDAError("CUDA malloc");
1069
1070 //cpy from cpu to gpu
1071 cudaMemcpy(d_sumr, sumr, temp * sizeof(int), cudaMemcpyHostToDevice);
1072 cudaMemcpy(d_sumg, sumg, temp * sizeof(int), cudaMemcpyHostToDevice);
1073 cudaMemcpy(d_sumb, sumb, temp * sizeof(int), cudaMemcpyHostToDevice);
1074 cudaMemcpy(d_rgb, rgb, sizeof(PPMrgb)*size, cudaMemcpyHostToDevice);
1075 checkCUDAError("CUDA memcpy");
1076
1077 //call kernel
1078 dim3 Grid(width / c, high / c);
1079 dim3 Block(1);
1080 aveMosaic3 << < Grid, Block, temp >> > (d_rgb, c, width, d_sumr, d_sumg, d_sumb);
1081 checkCUDAError("CUDA kernel");
```

Figure 1.3.1 code of Structures of Array



```
plorer (Ctrl+; 804 1
uc C:\Windows\system32\cmd.exe
CPU Average image colour red = 145, green = 126, blue = 111
GPUtime:0.060000请按任意键继续. . .
D
J
```

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).

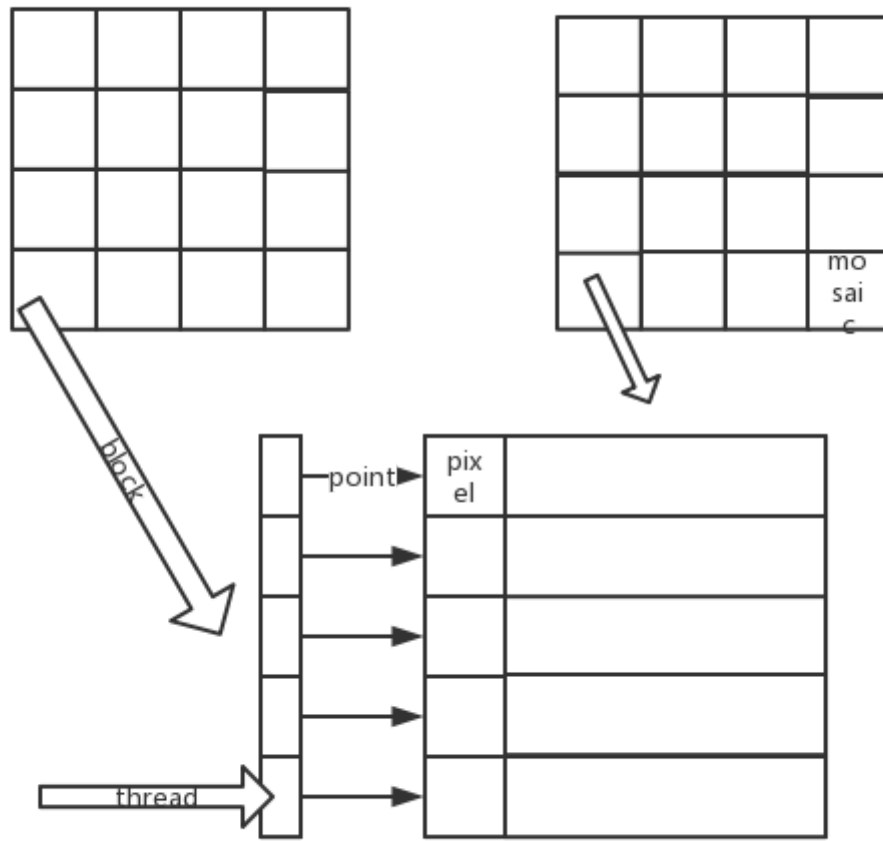


Figure 1.4.1 the structure of Vision 4

```

561
562 int sum = 0;
563 __shared__ int sumRed;
564 __shared__ int sumGreen;
565 __shared__ int sumBlue;
566 __shared__ int aveRed;
567 __shared__ int aveGreen;
568 __shared__ int aveBlue;
569
570 //int index = threadIdx.x + threadIdx.y * blockDim.x * c + blockIdx.x * c + blockIdx.y * c * width;
571 int index = threadIdx.x + threadIdx.y * width + blockIdx.x * c + blockIdx.y * c * width;
572
573 for (int i = 0; i < c; i++) {
574     sumRed = sumRed + image->rgb[index + i].red;
575     sumGreen = sumGreen + image->rgb[index + i].green;
576     sumBlue = sumBlue + image->rgb[index + i].blue;
577 }
578
579 if (threadIdx.x == 0) {
580     aveRed = sumRed / (c * c);
581     aveGreen = sumGreen / (c * c);
582     aveBlue = sumBlue / (c * c);
583 }
584
585 for (int i = 0; i < c; i++) {
586     image->rgb[index + i].red = aveRed;
587     image->rgb[index + i].green = aveGreen;
588     image->rgb[index + i].blue = aveBlue;
589 }

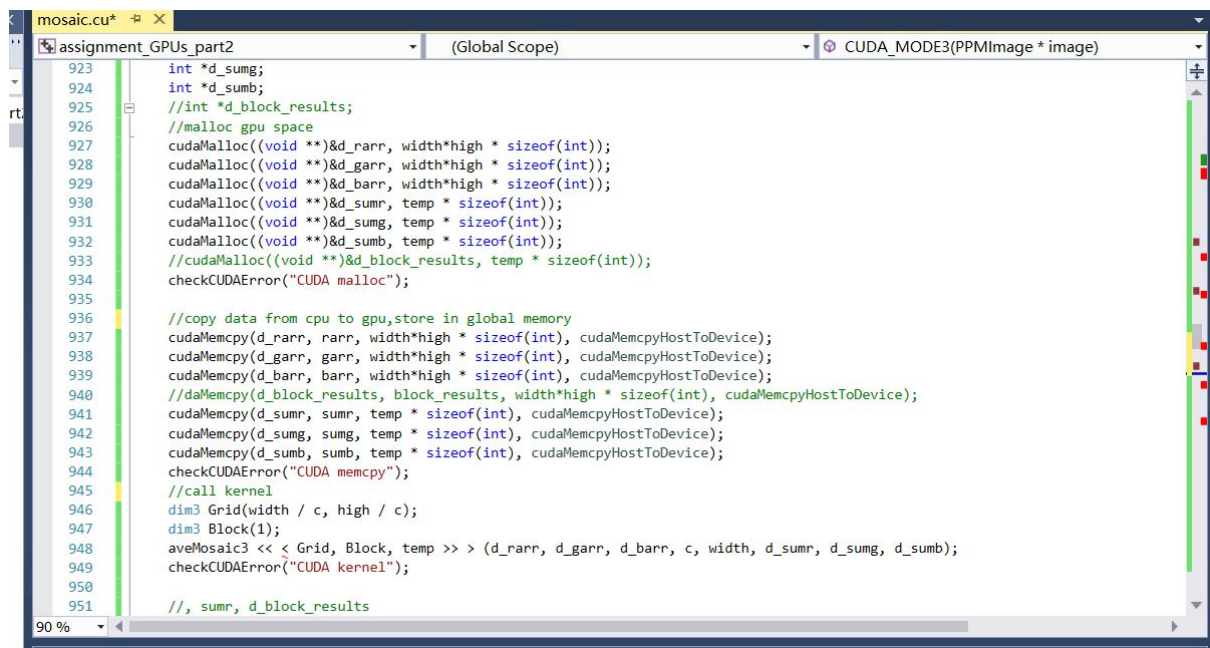
```

Figure 1.4.2

2. Optimisations

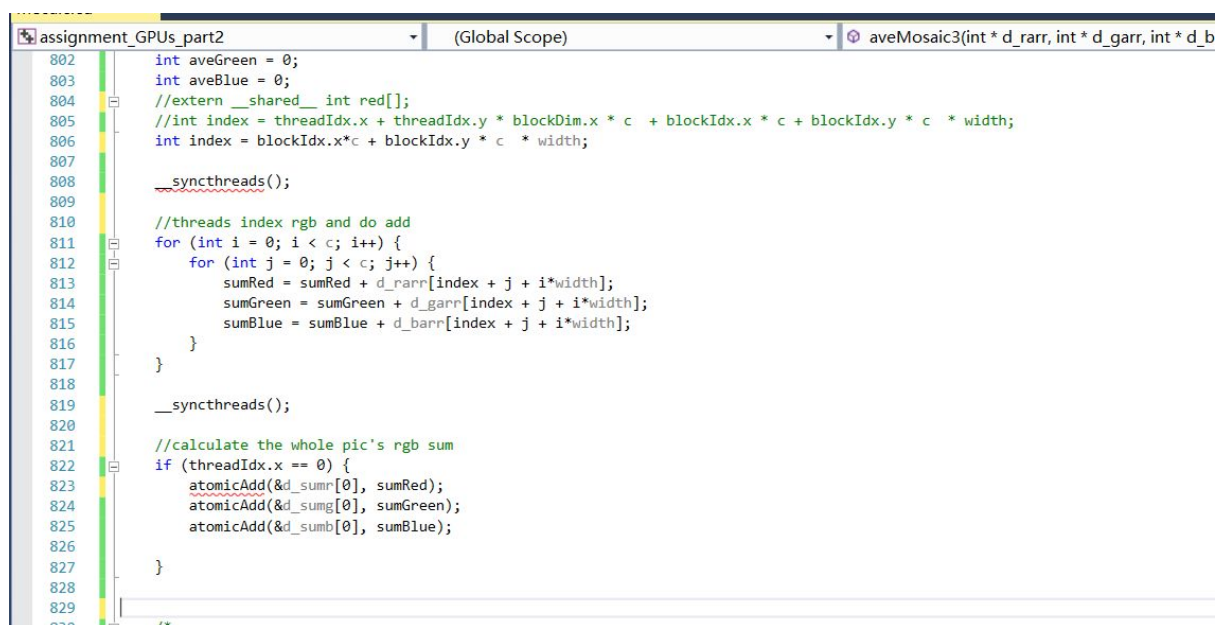
2.1 Different methods to layout or represent your data

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:



```
mosaic.cu* X
assignment_GPUs_part2 (Global Scope) CUDA_MODE3(PPMImage * image)
923 int *d_sumg;
924 int *d_sumb;
925 //int *d_block_results;
926 //malloc gpu space
927 cudaMalloc((void **)&d_rarr, width*high * sizeof(int));
928 cudaMalloc((void **)&d_garr, width*high * sizeof(int));
929 cudaMalloc((void **)&d_barr, width*high * sizeof(int));
930 cudaMalloc((void **)&d_sumr, temp * sizeof(int));
931 cudaMalloc((void **)&d_sumg, temp * sizeof(int));
932 cudaMalloc((void **)&d_sumb, temp * sizeof(int));
933 //cudaMalloc((void **)&d_block_results, temp * sizeof(int));
934 checkCudaError("CUDA malloc");
935
936 //copy data from cpu to gpu,store in global memory
937 cudaMemcpy(d_rarr, rarr, width*high * sizeof(int), cudaMemcpyHostToDevice);
938 cudaMemcpy(d_garr, garr, width*high * sizeof(int), cudaMemcpyHostToDevice);
939 cudaMemcpy(d_barr, barr, width*high * sizeof(int), cudaMemcpyHostToDevice);
940 //daMemcpy(d_block_results, block_results, width*high * sizeof(int), cudaMemcpyHostToDevice);
941 cudaMemcpy(d_sumr, sumr, temp * sizeof(int), cudaMemcpyHostToDevice);
942 cudaMemcpy(d_sumg, sumg, temp * sizeof(int), cudaMemcpyHostToDevice);
943 cudaMemcpy(d_sumb, sumb, temp * sizeof(int), cudaMemcpyHostToDevice);
944 checkCudaError("CUDA memcpy");
945 //call kernel
946 dim3 Grid(width / c, high / c);
947 dim3 Block(1);
948 aveMosaic3 <<< Grid, Block, temp >>> (d_rarr, d_garr, d_barr, c, width, d_sumr, d_sumg, d_sumb);
949 checkCudaError("CUDA kernel");
950
951 //, sumr, d_block_results
```

Figure 2.1.1 code of using array 1



```
assignment_GPUs_part2 (Global Scope) aveMosaic3(int *d_rarr, int *d_garr, int *d_b
802 int aveGreen = 0;
803 int aveBlue = 0;
804 //extern __shared__ int red[];
805 //int index = threadIdx.x + threadIdx.y * blockDim.x * c + blockIdx.x * c + blockIdx.y * c * width;
806 int index = blockIdx.x * c + blockIdx.y * c * width;
807
808 __syncthreads();
809
810 //threads index rgb and do add
811 for (int i = 0; i < c; i++) {
812     for (int j = 0; j < c; j++) {
813         sumRed = sumRed + d_rarr[index + j + i*width];
814         sumGreen = sumGreen + d_garr[index + j + i*width];
815         sumBlue = sumBlue + d_barr[index + j + i*width];
816     }
817 }
818
819 __syncthreads();
820
821 //calculate the whole pic's rgb sum
822 if (threadIdx.x == 0) {
823     atomicAdd(&d_sumr[0], sumRed);
824     atomicAdd(&d_sumg[0], sumGreen);
825     atomicAdd(&d_sumb[0], sumBlue);
826 }
827 }
828
829 /*
```

Figure 2.1.2 code of using array 2


```

assignment_GPUs_part2 (Global Scope) aveMosaic3(int * d_rarr, int * d_garr, int * d_t
1040 int *sumr = (int *)malloc(temp * sizeof(int));
1041 int *sumg = (int *)malloc(temp * sizeof(int));
1042 int *sumb = (int *)malloc(temp * sizeof(int));
1043
1044 int *d_sumr;
1045 int *d_sumg;
1046 int *d_sumb;
1047 PPMrgb *d_rgb;
1048 //malloc gpu space
1049 cudaMalloc((void **)&d_sumr, temp * sizeof(int));
1050 cudaMalloc((void **)&d_sumg, temp * sizeof(int));
1051 cudaMalloc((void **)&d_sumb, temp * sizeof(int));
1052 cudaMalloc((void **)&d_rgb, sizeof(PPMrgb)*size);
1053 checkCUDAError("CUDA malloc");
1054
1055 //cpy from cpu to gpu
1056 cudaMemcpy(d_sumr, sumr, temp * sizeof(int), cudaMemcpyHostToDevice);
1057 cudaMemcpy(d_sumg, sumg, temp * sizeof(int), cudaMemcpyHostToDevice);
1058 cudaMemcpy(d_sumb, sumb, temp * sizeof(int), cudaMemcpyHostToDevice);
1059 cudaMemcpy(d_rgb, rgb, sizeof(PPMrgb)*size, cudaMemcpyHostToDevice);
1060 checkCUDAError("CUDA memcpy");
1061
1062 //call kernel
1063 dim3 Grid(width / c, high / c);
1064 dim3 Block(1);
1065 aveMosaic3 << < Grid, Block, temp >> > (d_rgb, c, width, d_sumr, d_sumg, d_sumb);
1066 checkCUDAError("CUDA kernel");
1067
1068

```

Figure 2.1.3 code of using structures of arrays 1

```

mosaic.cu x
assignment_GPUs_part2 (Global Scope) aveMosaic3__struc(PPMrgb * d_rgb, int c,
987 int aveGreen = 0;
988 int aveBlue = 0;
989
990 int index = blockIdx.x*c + blockIdx.y * c * width;
991
992 //threads index rgb and do add
993 for (int i = 0; i < c; i++) {
994     for (int j = 0; j < c; j++) {
995         sumRed = sumRed + d_rgb[index + j + i*width].red;
996         sumGreen = sumGreen + d_rgb[index + j + i*width].green;
997         sumBlue = sumBlue + d_rgb[index + j + i*width].blue;
998     }
999 }
1000
1001 __syncthreads();
1002 //calculate the whole pic's rgb sum
1003 if (threadIdx.x == 0) {
1004     atomicAdd(&d_sumr[0], sumRed);
1005     atomicAdd(&d_sumg[0], sumGreen);
1006     atomicAdd(&d_sumb[0], sumBlue);
1007 }
1008
1009 __syncthreads();
1010 //calculate the average rgb of every mosaic
1011 aveRed = sumRed / div;
1012 aveGreen = sumGreen / div;
1013 aveBlue = sumBlue / div;
1014
1015

```

Figure 2.1.4 code of using structures of arrays 2


```
C:\Windows\system32\cmd.exe
GPU Average image colour red = 125, green = 126, blue = 111
GPUtime:0.109000请按任意键继续. . .
```

a

```
C:\Windows\system32\cmd.exe
CPU Average image colour red = 125, green = 126, blue = 111
GPUtime:0.061000请按任意键继续. . .
```

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.

```

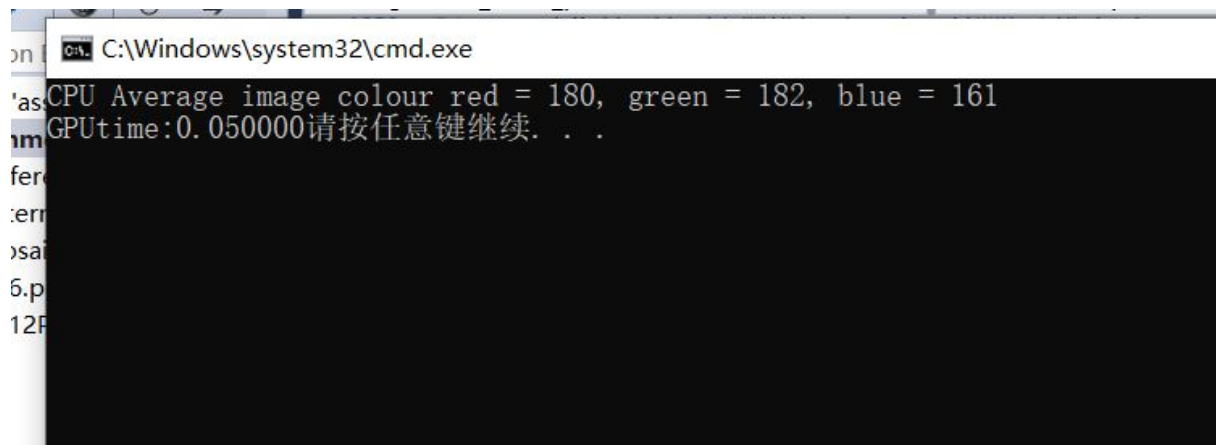
__global__ void aveMosaic3_struct(PPMrgb *d_rgb, int c, int width, int *d_sumr, int *d_sumg, int *d_sumb) {

    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



```

C:\Windows\system32\cmd.exe
CPU Average image colour red = 180, green = 182, blue = 161
GPUtime:0.050000请按任意键继续. . .

```

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

```

Figure 2.1.4 code of using constant memory

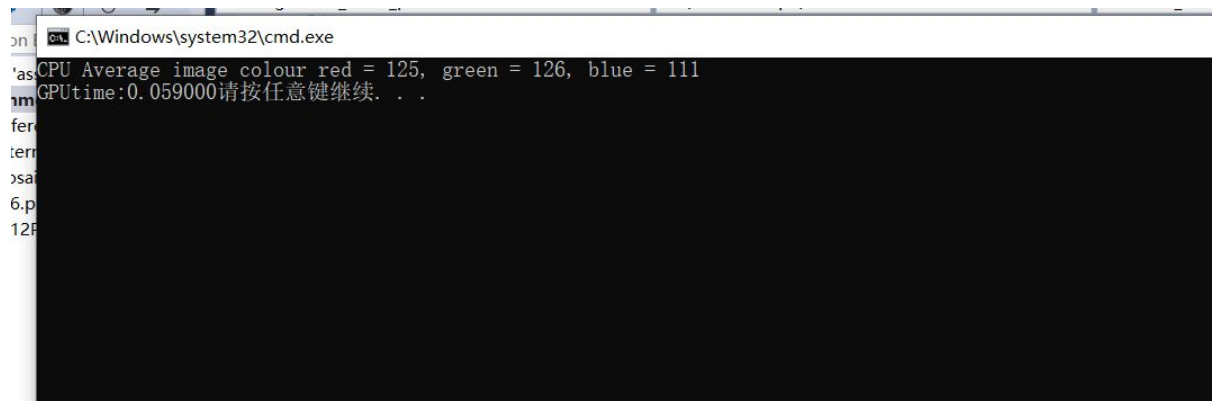


Figure 2.2.1 runtime of using constant memory

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.