Assignment 2 report

Gang Chen
Registration No: 180221840

All experimental data comes from a personal laptop, configured as follows: CPU: core i7-7700hq, 4 cores and 8 threads, 2.8Ghz; GPU: Nvidia GeForce GTX1060.

Improvement
There are many imperfections in the original base code, which caused the OPENMP part of the first assignment not be optimized. At the beginning of this assignment, I improved the original code, focusing on the loop structure that traverses each pixel. The loop identifier will not be modified in the loop. At the same time, I also added a method for processing images that are not square.

```
row_quot = image->x / c;
row_rema = image->y % c;
col_quot = image->x / c;
col_rema = image->y % c;

int cells_per_row;
int cells_per_col;

if (row_rema)
   cells_per_row = row_quot + 1;
else
   cells_per_row = row_quot;

if (col_rema)
   cells_per_col = col_quot + 1;
else
   cells_per_col = col_quot;


for ( y_in_col = 0; y_in_col < cells_per_col; y_in_col++)
   {
       if (y_in_col == col_quot)
           col_limit = col_rema;
       else
           col_limit = c;
       for (x_in_row = 0; x_in_row < cells_per_row; x_in_row++)
          {
              if (x_in_row == row_quot)
                  row_limit = row_rema;
              else
                  row_limit = c;

              sumr = 0;
```

```
            sumg = 0;
            sumb = 0;
       for (y_in_cell = 0; y_in_cell < col_limit; y_in_cell++)
          {
            for ( x_in_cell = 0; x_in_cell < row_limit; x_in_cell++)
               {
                  int temp;
                  temp = y_in_col * (c*image->x) + y_in_cell*image->x +
(x_in_row*c + x_in_cell);
                  sumr += image->d[temp].r;
                  sumg += image->d[temp].g;
                  sumb += image->d[temp].b;
               }
          }
     .......
    }
```

CUDA programming

In my program, after reading the completed file, all the information of the image will be stored in a structure called WholeImage. This structure is composed of three parts, two integer values x and y represent the width and length of the image. There is also a structure EveryPixel that stores all rgb values. There are three unsigned chars in EveryPixel, corresponding to the values of r, g, b.

```
struct EveryPixel {
       unsigned char r, g, b;
} ;

struct WholeImage {
       int x, y;
       EveryPixel *d;
} ;
```

At the beginning of the job, I analyzed the original program. Since my original program used a quadruple for loop to locate each pixel value in the entire array of stored pixels, I hope that when I do CUDA programming this quadruple loop can be split. Here I will make a definition. Using the width of the image divided by the value of c input by the user to get the number of mosaics in the x-direction. In the following, I use cell to call each mosaic. A thread in a two-dimensional block can be positioned with threadIdx.x and threadIdx.y. So I decided to use a block to process the entire image, and each thread processes a cell.

First, I need to copy input data from CPU memory to GPU memory. The beginning of this step is to allocate the corresponding storage space for each data in gpu memory.

```
cudaMalloc((void **)&dc, sizeof(unsigned int));
cudaMalloc((void **)&length, sizeof(unsigned int));
cudaMalloc((void **)&width, sizeof(unsigned int));
```

```
cudaMalloc((void **)&d_image, image->x*image->y * sizeof(EveryPixel));
cudaMalloc((void **)&d_image_out, image->x*image->y * sizeof(EveryPixel));
cudaMalloc((void **)&dy_in_cell, sizeof(signed short));
cudaMalloc((void **)&dx_in_cell, sizeof(signed short));
cudaMalloc((void **)&d_sumr, sizeof(unsigned
long*)*cells_per_col*cells_per_row);
cudaMalloc((void **)&d_sumg, sizeof(unsigned
long*)*cells_per_col*cells_per_row);
cudaMalloc((void **)&d_sumb, sizeof(unsigned
long*)*cells_per_col*cells_per_row);
```

The most important of these is the allocation of space to all pixel data. At first, I wanted to convert the *d in the structure into three two-dimensional arrays separated by RGB, but later found that it didn't simplify the operation, but it made it more difficult to locate each pixel because I also did not use two-dimensional arrays in the original program. So here I allocate a contiguous space to hold the entire image->d, the size is `image->x*image->y * sizeof(EveryPixel)`.The next step is to copy the required data into the allocated space.

```
cudaMemcpy(dc, &c, sizeof(int), cudaMemcpyHostToDevice);
cudaMemcpy(length, &image->y, sizeof(int), cudaMemcpyHostToDevice);
cudaMemcpy(width, &image->x, sizeof(int), cudaMemcpyHostToDevice);
cudaMemcpy(dy_in_cell, &y_in_cell, sizeof(signed short),
cudaMemcpyHostToDevice);
cudaMemcpy(dx_in_cell, &x_in_cell, sizeof(signed short),
cudaMemcpyHostToDevice);
cudaMemcpy(d_image, image->d, image->x*image->y * sizeof(EveryPixel),
cudaMemcpyHostToDevice);
```

Then call the kernel by using the CUDA kernel launch syntax. The plan here is to use one block, each thread is responsible for a cell.

```
dim3 blocksPerGrid(1, 1, 1);
dim3 threadsPerBlock(cells_per_row, cells_per_col, 1);
runCuda << <blocksPerGrid, threadsPerBlock >> > (d_image, d_image_out,dc,
width,d_sumr,d_sumg,d_sumb);
```

I designed a global function and a device function, the global function acts as the original outer two-layer loop, just pass threadId.x, threadId.y to the device function, and the device function is equivalent to the original inner two-layer loop. With the outer layer threadId.x, threadId.y value, the device function can locate the value in each cell and calculate the average value in the cell.

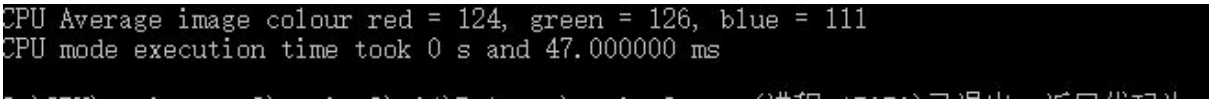
But this method cannot run very quickly, because when I use a graph with a lot of pixels, the number of threads allocated in a block will far exceed the limit of 1024. I need to increase the number of blocks and adjust the number of threads in each block. I tried to use 2D grid, but locating every thread by using threadIdx became really hard. Therefore, I temporarily decided to solve the problem of processing large images by multiplying the number of blocks by the number of threads equal to the number of cells.

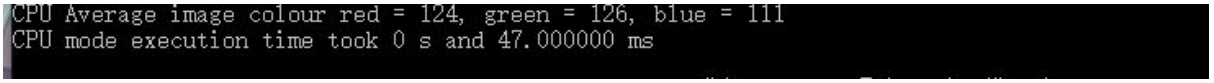
```
dim3 blocksPerGrid(cells_per_col, 1, 1);
dim3 threadsPerBlock(cells_per_row, 1, 1);
```

Since the assignment has a requirement to calculate the average of the entire image, in the first version, a block is used to process the pixels of the entire image. I declare three `__shared__` arrays in the global function to store the RGB sum of all the cells. Because threads in one block can access the same `__shared__` array, it is easier to collect the sum value. Then call `__syncthreads` to ensure shared memory data is populated by all threads. But when I changed the strategy of blocks and threads allocating, because there are too many blocks, I temporarily gave up using shared memory. However, I think there should be one method to reduce the times that access global memory.

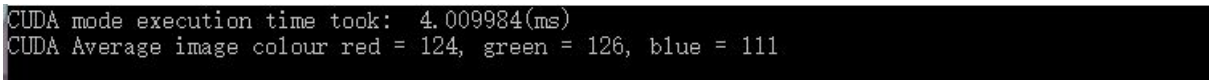
Testing and data

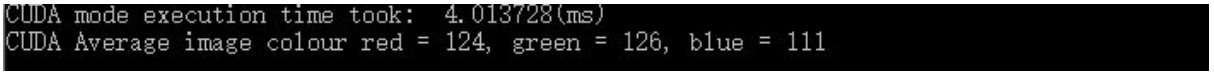
The experiment compares the cpu processing method with the cuda processing method by controlling the variables. The experimental process is as follows:
The picture used in the first test was Dog2048x2048.ppm. When the user input c value is 2, three consecutive tests are performed on the CPU processing method and the Cuda processing method. The screenshots are as follows:
CPU:

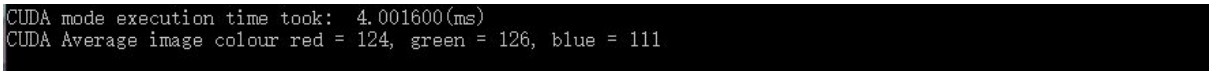




CUDA:







By calculating the average time: when c = 2, CPU takes 48ms, CUDA takes 4ms. Next increase the value of c, which is 4, 8, 64, 128, 256 and 512.

c	2	4	8	64	128	256	512
CPU	48.3ms	33.2ms	24.1ms	20.4ms	17.6ms	19.0ms	16.6ms
CUDA	4.0ms	4.1ms	5.8ms	4.8ms	10.9ms	36.6ms	129ms

The second test: Sheffield512x512.ppm, c = 2, 4, 8, 64, 128, 256 and 512
The test process is the same as the first test. The next test no longer shows the screenshot, and will directly give the final average time table.

c	2	4	8	64	128	256	512
CPU	3.3ms	2.3ms	1.6ms	1.3ms	1.3ms	1.6ms	1ms

CUDA	0.2ms	0.2ms	0.2ms	2.3ms	7.9ms	32.1ms	132.4ms

The last test : Sheffield16x16.ppm, c = 2, 4, 8, 16

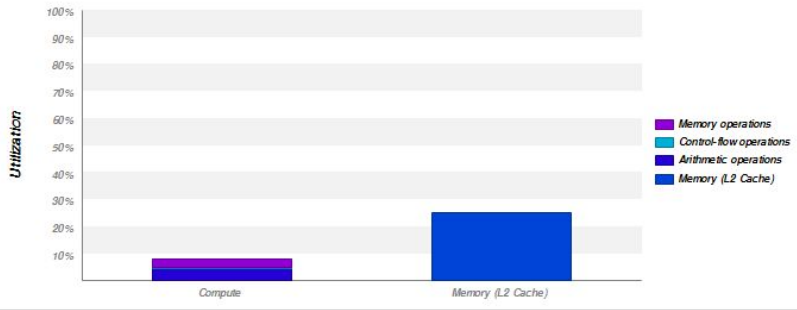
c	2	4	8	16
CPU	0.000003ms	0.000003ms	0.000003ms	0.000003ms
CUDA	0.013ms	0.02ms	0.04ms	0.13ms

From the results of these three experiments, first of all, from the perspective of CUDA, the more the number of cells is used, the shorter the time is. In my design, each thread will process all the pixels in a cell. As the number of cells increases, the number of pixels in each cell will decrease, the workload of each thread will decrease, and the work of each thread will decrease. They are all parallel, so time is reduced. In contrast, if there are more pixels in a thread to process, the processing will take more time and the time will increase.

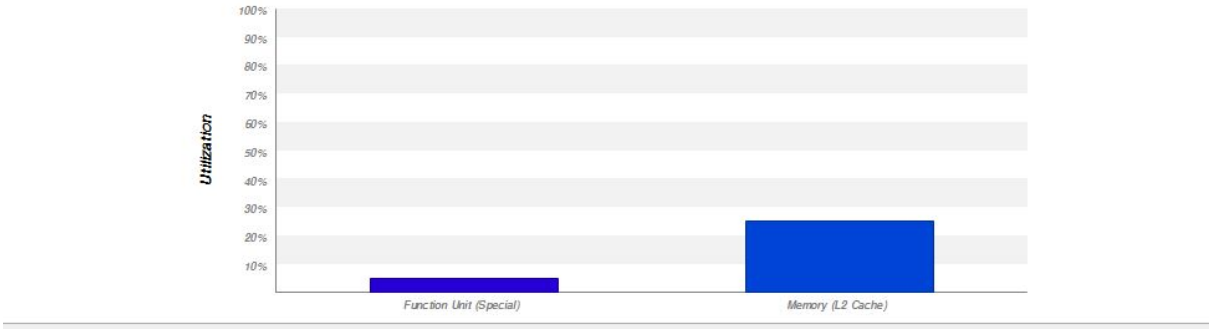
From the comparison of CPU and Cuda, the Cuda program I designed is more suitable for large images, and the user input c value is smaller. The specification of the first picture is 2048*2048. When c is 2, the time difference between CPU and Cuda operation is very obvious, but as c increases, the gap becomes smaller and smaller. When c exceeds 128, the processing speed of CPU Gradually surpassing Cuda's processing speed.
When the image is very small, cuda processing has no advantage. In the last test, the processing time of CPU appears 0.000000ms multiple times. If it is not a problem with my program, it means that the advantages of cpu are much larger than that of cuda when dealing with small images such as Sheffield16x16.ppm.

I can only let the program access an array that exists in global memory every time to store all the sum values.

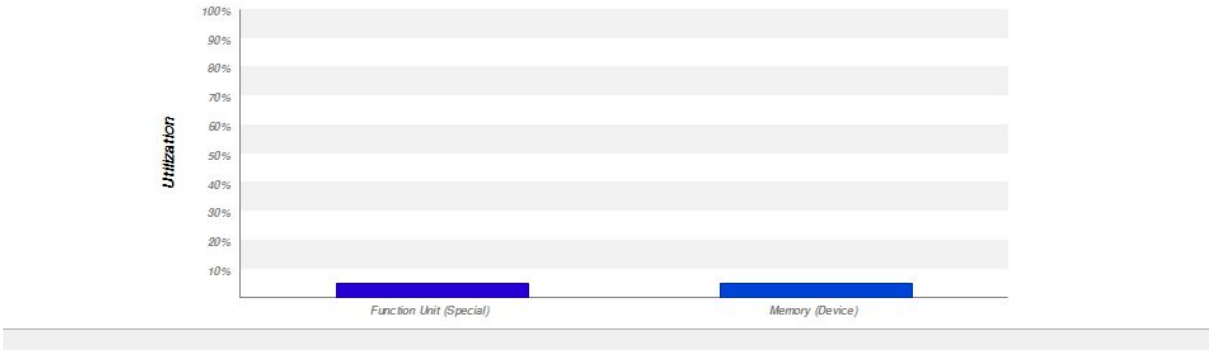
Dog2048*2048 c= 2



Dog2048*2048 c= 32



Dog2048*2048 c= 128



Since the utilization rate is too low, I decided to do some optimization to use shared memory. I tried to use shared memory to sum all cells' sum of RGB, but I failed. The code as follow:

```
__device__ unsigned long long d_sumr, d_sumg, d_sumb;

__syncthreads();

atomicAdd(&d_sumr, sumr);
atomicAdd(&d_sumg, sumg);
atomicAdd(&d_sumb, sumb);



gpuErrchk(cudaMemcpyToSymbol(d_sumr, &sumr2, sizeof(unsigned long long
int)));
gpuErrchk(cudaMemcpyToSymbol(d_sumg, &sumg2, sizeof(unsigned long long
int)));
cudaMemcpyToSymbol(d_sumb, &sumb2, sizeof(unsigned long long int));
```

Due to lack of time, I did not complete the optimization of this part. In my plan, the next step is to use three shared values to accumulate `d_sumr`, `d_sumg`, `d_sumb`respectively. Then send them to CPU and print them.
In theory, using a value in shared memory to add the sum value of each block can speed up the program because it is no longer necessary for each thread to access the global after completing the operation.