Report

Introduction

This is a report of the implementation of the mosaic filter based on the parallel computing with the graphic processing units (GPUs).

Parallel strategy

The image data in the project is being pre-processed into 1D array from the 2D array.

In this project, there are two parallel strategy. The first strategy called the pixel based strategy and it is mapping each pixel onto each thread regardless of the size of the mosaic cell. Two kernel functions are used in this strategy. The first kernel function is used to add up the value of RGB in each mosaic cell and stored it into an array. The architecture of this kernel is 1D grid with 1D block. All of threads are flatten into 1D array corresponding to the pixel array and each threads process a pixel. To add up RGB value of each mosaic cell, in this strategy, the operation *atomicAdd* is adopted on the variable in the global memory. The second kernel function is used to calculate the average RGB value and then replace the original value with the average value. The dimensions of the grid and block are both 1D. A pixel in the image is processed by a thread in the kernel.

The second strategy is optimised and based on the first strategy, called the mosaic cell based strategy because it is setting the size of block according to the size of the mosaic cell and then mapping each pixel onto each thread in the corresponding block which means one thread processes one pixel. Two kernel functions are used in this strategy. The first kernel function is used to add up the value of RGB in each mosaic cell and stored it into an array. The architecture of this kernel is 2D grid with 1D block. Specifically, the dimension of the block is depending on *c* which is the size of the mosaic cell and the dimension of the grid is depending on how many mosaic cells the image can be divided into. To add up RGB value of each mosaic cell, in this strategy, the operation *atomicAdd* is adopted on the variable in the shared memory. The second kernel function is the same as the first strategy.

Implementation

Before executing kernel function, the image data need to be transfer to 1D array type from 2D array type. Then, copy the data from the host to device. Here, it requires an array which would be used to store the accumulated value of each mosaic cell and an array which would be used to store the number of the pixels processed in the mosaic cell.

In the first strategy, the kernel function, *add_up()*, calculates the index of the threads which is also the index of the pixel in the array. By using the index of the pixel, the id of the mosaic cell which contains the pixel could be computed. Then the RGB value of the pixel will be accumulated into the corresponding location in the mosaic cell array. All of the calculation only require the variables (in global memory) which is passed in when the function is called.

After the accumulation, each value in the mosaic cell array is the accumulated value of pixel in the mosaic cell. In the *avg()* kernel, the value in the mosaic cell array is divided by the number of the pixels in this mosaic cell. Then the original RGB value of the pixel in the image would be replaced by the calculated value. All of the calculation only require the variables (in global memory) which is passed in when the function is called.

Test

The test on the first version takes different size of the image with different size. The time consumption has been recorded and displayed in the following table.

Width	Height	C (Mosaic cell size)	Cost
2048	2048	32	18.339296 ms
2048	2048	64	18.536064 ms
2048	2048	128	20.188000 ms
6000	4000	128	103.30291 ms
6000	4000	256	98.186012 ms
6000	4000	512	123.04262 ms

Table 1. Cost of program with different size of image and mosaic cell

There is no huge gap in the same image with different mosaic cell size, because the mosaic cell size would not be considered in the computation of kernel function. The performance analysis tool, NVDIA Visual Profiler, is applied on the program. Individual kernel testing focus on the utilization of compute and memory. Following image shows the examination of the kernel function *add_up()*.

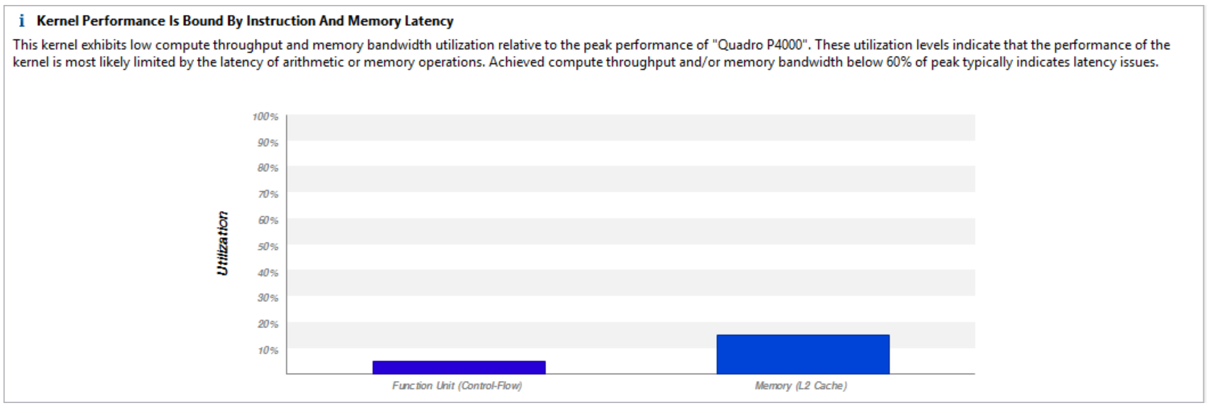

Kernel Performance Is Bound By Instruction And Memory Latency
This kernel exhibits low compute throughput and memory bandwidth utilization relative to the peak performance of "Quadro P4000". These utilization levels indicate that the performance of the kernel is most likely limited by the latency of arithmetic or memory operations. Achieved compute throughput and/or memory bandwidth below 60% of peak typically indicates latency issues.

Figure 1. Examination of the kernel function *add_up()* in first version

This examination result shows low compute throughput and memory bandwidth utilization. No memory operation exist in this kernel. So, this kernel need to be optimised.

The examination of the second kernel function *avg()* would be showing in the flowing image.

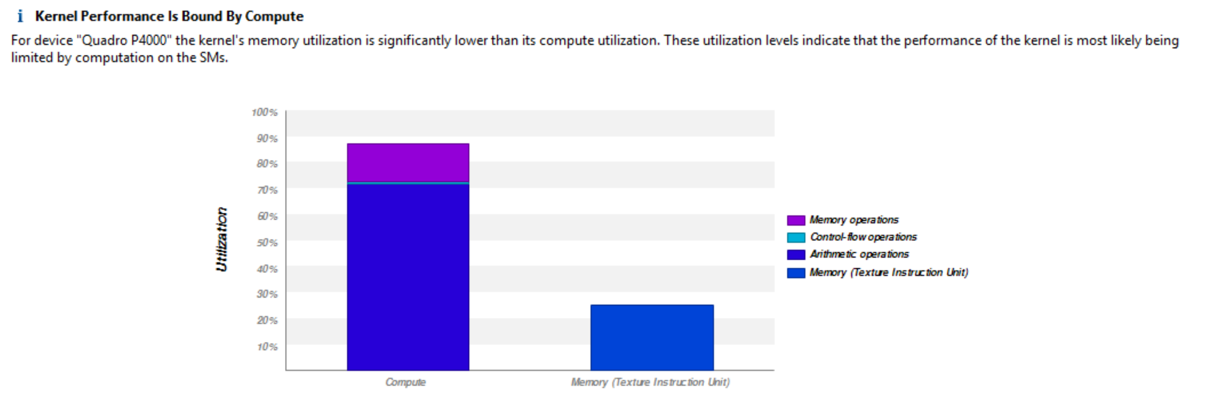

Kernel Performance Is Bound By Compute
For device "Quadro P4000" the kernel's memory utilization is significantly lower than its compute utilization. These utilization levels indicate that the performance of the kernel is most likely being limited by computation on the SMs.

Figure 2. Examination of the kernel function *avg()* in first version

The image shows high compute utilization, though the huge gap is between the compute utilization and the memory utilization.

Optimization

Obviously, the first kernel function, *add_up()*, has huge space of the optimization. The shared memory can be considered to be utilized in the optimization, because in the first strategy no utilization of shared memory cause no memory operation and low compute utilization. If the shared memory can be utilized the rate of the utilization of the compute must be improve and the cost of the program should decrease. Thus, the second strategy would store the accumulated RGB value of each pixel into corresponding block. The difficulty of the implementation of strategy is to calculate the indexes of the pixel in the image or the block. To be more intuitive, following image show the mapping from the image to the kernel.

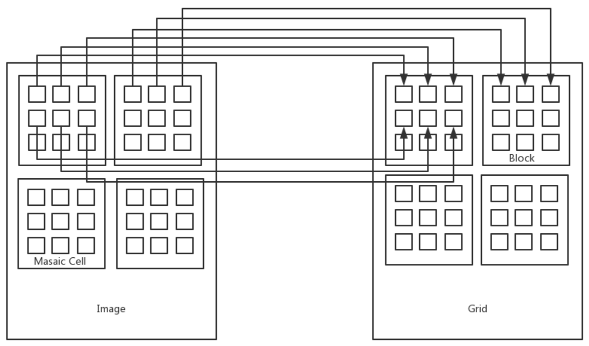
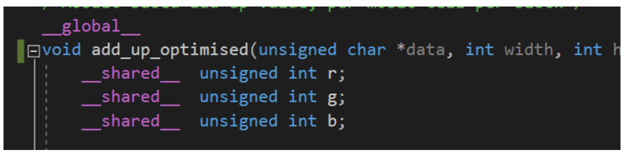


Figure 3. Strategy of second version

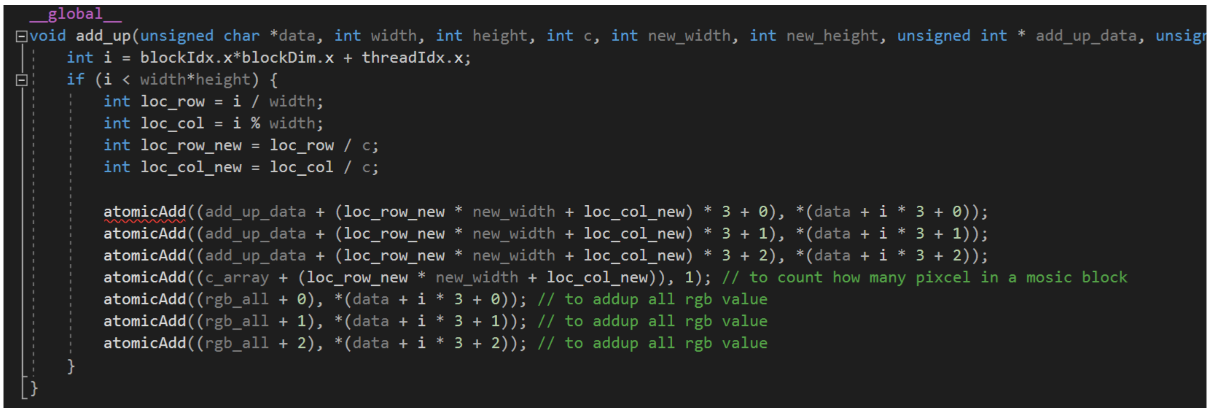
Based on the first strategy, three variable in shared memory are declare. See following image.



```
__global__
void add_up_optimised(unsigned char *data, int width, int h
    __shared__  unsigned int r;
    __shared__  unsigned int g;
    __shared__  unsigned int b;
```

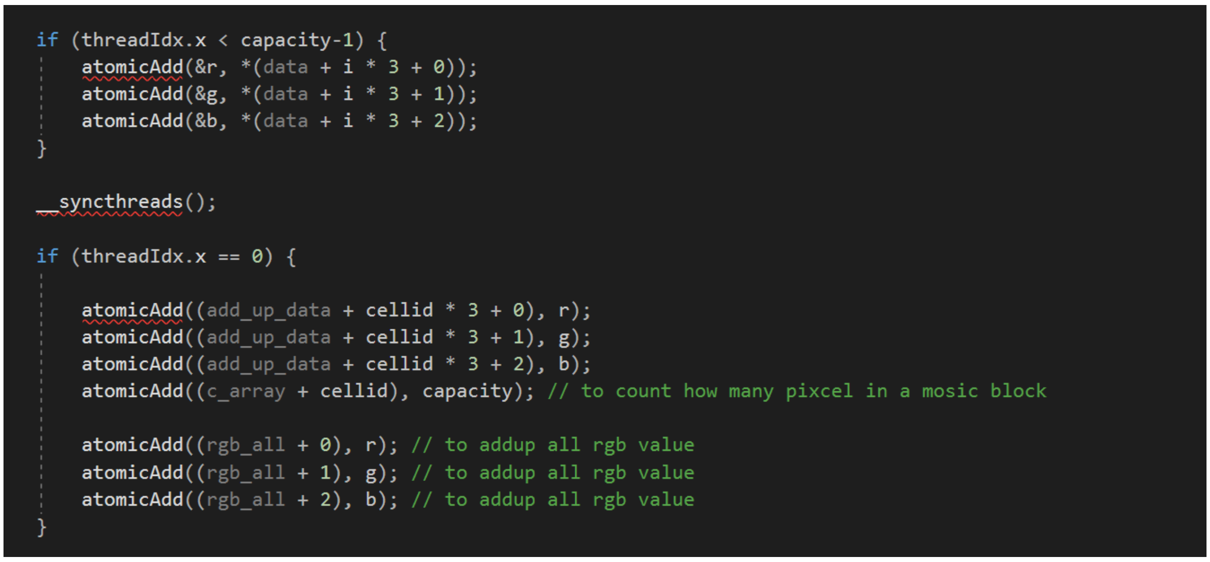
Figure 4. Usage of shared memory in second strategy

The operation *atomicAdd* are also applied on this three variable in the separate blocks. Comparing the first strategy, all threads are applied *atomicAdd* function and referring to the same variable location, which causes increasing time consumption because the add operation need to be done one thread by one thread. In the second strategy, atomicAdd function is applied on the shared memory, the accumulation operation can be run parallel between the blocks. Then, after accumulation, one thread in each block would be chosen to add up the result to the global memory. Specific difference would be shown in following images.



```
__global__
void add_up(unsigned char *data, int width, int height, int c, int new_width, int new_height, unsigned int * add_up_data, unsigr
    int i = blockIdx.x*blockDim.x + threadIdx.x;
    if (i < width*height) {
        int loc_row = i / width;
        int loc_col = i % width;
        int loc_row_new = loc_row / c;
        int loc_col_new = loc_col / c;

        atomicAdd((add_up_data + (loc_row_new * new_width + loc_col_new) * 3 + 0), *(data + i * 3 + 0));
        atomicAdd((add_up_data + (loc_row_new * new_width + loc_col_new) * 3 + 1), *(data + i * 3 + 1));
        atomicAdd((add_up_data + (loc_row_new * new_width + loc_col_new) * 3 + 2), *(data + i * 3 + 2));
        atomicAdd((c_array + (loc_row_new * new_width + loc_col_new)), 1); // to count how many pixcel in a mosic block
        atomicAdd((rgb_all + 0), *(data + i * 3 + 0)); // to addup all rgb value
        atomicAdd((rgb_all + 1), *(data + i * 3 + 1)); // to addup all rgb value
        atomicAdd((rgb_all + 2), *(data + i * 3 + 2)); // to addup all rgb value
    }
}
```

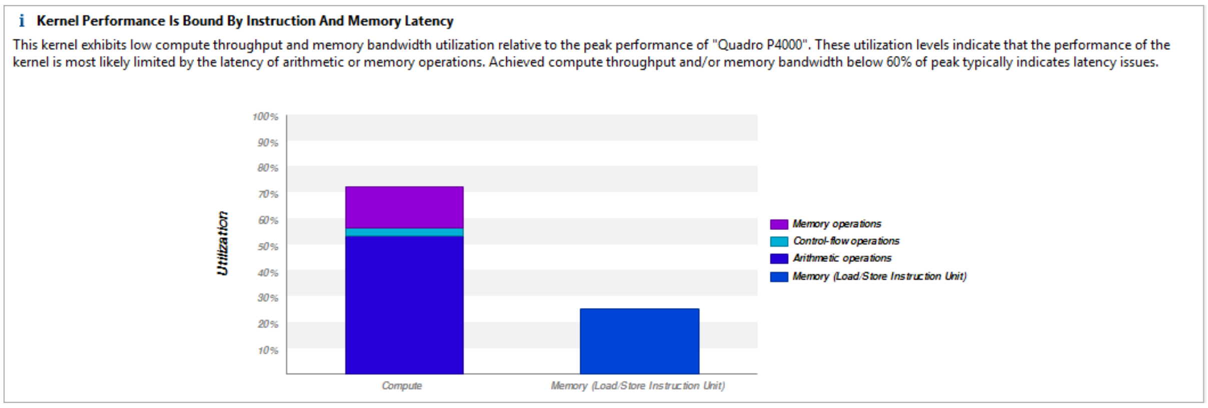
Figure 5. Usage of *atomicAdd* in the first strategy



```
if (threadIdx.x < capacity-1) {
    atomicAdd(&r, *(data + i * 3 + 0));
    atomicAdd(&g, *(data + i * 3 + 1));
    atomicAdd(&b, *(data + i * 3 + 2));
}

__syncthreads();

if (threadIdx.x == 0) {

    atomicAdd((add_up_data + cellid * 3 + 0), r);
    atomicAdd((add_up_data + cellid * 3 + 1), g);
    atomicAdd((add_up_data + cellid * 3 + 2), b);
    atomicAdd((c_array + cellid), capacity); // to count how many pixcel in a mosic block

    atomicAdd((rgb_all + 0), r); // to addup all rgb value
    atomicAdd((rgb_all + 1), g); // to addup all rgb value
    atomicAdd((rgb_all + 2), b); // to addup all rgb value
}
```

Figure 5. Usage of *atomicAdd* in the second strategy

Theoretically, in this optimization, comparing the first strategy less address conflicts exist in the program. Thus, the performance of the accumulation would increase. Using the test tools on the optimised strategy, the result of the examination would be shown in the following images.



i Kernel Performance Is Bound By Instruction And Memory Latency

This kernel exhibits low compute throughput and memory bandwidth utilization relative to the peak performance of "Quadro P4000". These utilization levels indicate that the performance of the kernel is most likely limited by the latency of arithmetic or memory operations. Achieved compute throughput and/or memory bandwidth below 60% of peak typically indicates latency issues.

Figure 6. Examination of the kernel function *add_up()* in second version

Obviously, the rate of the utilization of compute and memory significantly increase in *add_up* kernel, comparing to the first version. The kernel function, *avg(),* does not change, because in the examination of the first version, it performs acceptably. Testing the second version with the different images, the result is shown in the following table. Significant improvement can be found after optimised.

Width	Height	C (Mosaic cell size)	Cost (Old)	Cost (Optimised)
2048	2048	32	18.339296 ms	5.934784 ms
6000	4000	32	98.181763 ms	30.585665 ms

Table 2. Cost of program with different size of image and mosaic cell

Overall, the optimised has been applied on the program. The significant improvement is shown on the utilization of compute. However, the optimised version is not stable always because some unknown bugs might be exist, so the finally version I will keep the code in the first version which is a stable version, the optimised approaches have been displayed in this report.