EE4C07 - Advanced Computing Systems

Report

Angqi Liu
Wei Sun
Mairin Kroes

Contents

1 Outline

In this report the design process of a set of 4 image processing kernels will be documented. These kernels were derived from sequential algorithms, and have been optimized to run efficiently on NVIDIA GPUs by the use of the CUDA® API.

In order to verify the efficacy of the implemented optimizations, multiple benchmarks were performed to evaluate the speed up over the sequential algorithms. The results of these benchmarks will be analyzed and reflected upon. Finally, some conclusions and recommendations will be made regarding the overall work.

2 Implementation

In this section the design process of the various image processing kernels will be documented.

First, we will provide a brief description of our applied strategy to implement each of the kernels. Subsequently, we will identify the encountered bottlenecks and limitations in each of the kernels. Finally, we will conclude with some optimization we applied to improve the application.

Before introducing the implementation of our CUDA kernels, we should first introduce the CImg library, which is used in this program. The CImg Library is a small, open-source, and modern C++ toolkit for image processing. It provides many useful functions such as getting the size (width and height) of an images, as well as the functionality to store the data of the image to memory, and return a pointer to this stack. CImg stores the color image as a 1-dimensional array, i.e.: RRRRR....GGGGG...BBBBB. For a color image with height (H) and width (W), CImg will allocate 3N = 3*H*W*sizeof(unsigned char) bytes for it. Elements from 0 to N-1 bytes represent the red color channel, N to 2N-1 are allocated for the representation of the green color channel, and the final portion for the blue color channel.

2.1 Grayscale Conversion

In the sequential version, the application is implemented by using nested for loops to convert each pixel from RBG to gray. There is no dependency between each loop, so we can compute all pixels concurrently. The number of total threads of a GPU is N (the size of input image). Because a block can only contain up to 1024 threads, we use N/1024 blocks to do the computations.

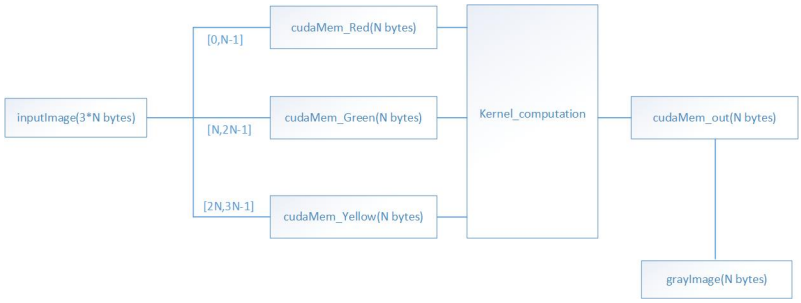

Figure 1: CUDA function

2.2 Histogram Computation

This function is used to compute the histogram of the grayImage from the last function. A histogram is an accurate graphical representation of the distribution of numerical data within an image. In this program, unsigned int histogram[256] is defined. histogram[i]=n means there are n pixels with gray intensity equaling i in the gray image.

In the sequential version, the program uses nested for loops to scan the gray intensity of the image. If grayImage[n]=i then histogram[i] will plus one.

If we apply the same method we used in the last function to implement the CUDA kernel, it will cause conflicts when two or more threads want to wrote to the same index of array(histogram[i]). To avoid this conflict, we have to use `atomicAdd(&histo[histogram[i]], 1)` instead of `histogram[i]]+1`. This atomic operation ensures that only one thread can access histogram[i], and other threads will have to wait. Also, we still need N threads to scan the entire

image. This method is easy to understand and code, but it is much more slower compared to the version implemented by shared memory, which will be introduced later.

2.3 Contrast Enhancement

The main goal of this function is to enhance the contrast of the input image. First we need to select two values (min and max) to represent the threshold of gray intensity. If grayImage[i] $<$ min, the value of the intensity will be set to 0 (black). If grayImage[i] $>$ max , it will be set to 255 (white). The values between min and max will be changed accordingly with the assignment statement `grayImage[i] = static_cast< unsigned char >(255.0f * (grayImage[i] - min) / (max-min))`.

Because there are no dependencies and conflicts, we can accelerate this function easily with CUDA. The parallelization strategy is almost the same as the function of Grayscale Conversion.

2.4 Smoothing

This function is used to smooth the image and reduce the noise. It first calculates the average values of a certain pixel and its neighbors, then replacing its original value with the average one.In this program, the filter size is two.As shown in 2, the yellow block's value with be replaced by the average of the 5*5 elements.

As mentioned in 1, Image data is stored in linear memory or as a vector rather than a matrix. So, accessing neighbors will be more complicated.Given the image width(W) and height(H), the position of element i can be represented as `(i/W,i%w)`.And i is also the thread ID in CUDA kernel. Parallelizing this function is not hard, most of the statements of sequential version do not need to be changed. We just need to unroll the loops.

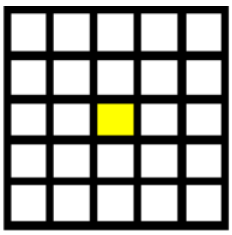

Figure 2: 5x5 matrix

3 Test Methodology

In this section we will give a brief overview of the test setup and methodology we used to obtain our measurement data.

All of the CUDA kernels were tested on the NVIDIA Tesla K40C in the ce-cuda01 server, while the sequential tests were performed on the 12 core / 24 thread CPU in the same system.

4 Performance Analysis

In this section we will show the profiling result, and provide an analysis of the performance of the GPU running the various kernels, in comparison to the host running the sequential algorithms. All tests were performed and assessed according to the methodology laid out in section 3.

By using CUDA profiler(version 8.0), we got the overall performance for executing the program, and we got different operating data from divergent kernel

Picture below, present overall performance of our program, however, the utilization is low, and many cores of the GPU idle, in our opinion, data size may influence the performance, and parallel execution of kernels also play an important role to program occupancy.

4.1 Optimizations

During this part, we ran the improved program, which implemented optimizations with respect to the memory hierarchy. In the original program, we solely accessed global memory, but it takes

substantial time to load data from the global memory on the GPU. Because of this, we implemented methods in the histogram and triangular smoothing algorithm that access the shared memory. The execution time reduced significantly to half of the original one in the histogram algorithm, and 3 times in the triangular smooth algorithm. We can see from the memory access parameters in figures 3 & 4 that the locality of access dramatically changed. Approximately 45% of the memory loads are now directed towards the shared memory, which nets us nearly 10× as much bandwidth.

We also noticed that by implementing shared memory in the histogram kernel, the occupancy increased from 78.9% to 97%. This also reduced the execution time by 76%.

4.2 Benchmarks

Before we started running the benchmarks, we made an estimate of how well the GPU should be able to do compared to a sequential implementation running on the CPU. Taking the theoretical peak floating point throughput of the NVIDIA Tesla K40C into account. The GPU should be capable of performing 4.129 trillion floating point operations per second (as listed by the profile. The CPU on the other hand, running the sequential implementations should have a max of 2.4 GFLOPs. The theoretical floating point throughput is actually based on the scenario where the GPU performs a multiply-add instruction, performing 2 floating point operations per cycle. Seeing that the there are no multiply-add instruction in our applications, this measure will not be accurate, in which case the theoretical max will be 2880 × 745MHz ≈ 2.15 TFLOPs. This means that we should expect at the most a speed-up of about 900×.

Table 1: Benchmark Results - Image 04

	`rgb2gray`	`histogram`	`contrast`	`triangularSmooth`
GPU (Communication)	3.01ms	0.49ms	0.76ms	1.97ms
GPU (Kernel)	0.12ms	1.28ms	0.12ms	0.84ms
CPU (Kernel)	5.68ms	7.67ms	22.2ms	236.3ms
Speed Up (System)	1.81	4.3	25.2	100.76
Speed Up (Kernel)	47.31	5.99	185.27	281.33

Table 2: Benchmark Results - Image 09

	`rgb2gray`	`histogram`	`contrast`	`triangularSmooth`
GPU (Communication)	73.2ms	16.2ms	24.4ms	54.0ms
GPU (Kernel)	2.76ms	48.44ms	3.96ms	32.2ms
CPU (Kernel)	367.2ms	189.5ms	784.2ms	10082.7ms
Speed Up (System)	4.83	2.93	27.7	116.97
Speed Up (Kernel)	133.04	3.91	198.03	313.13

Table 3: Benchmark Results - Image 15

	`rgb2gray`	`histogram`	`contrast`	`triangularSmooth`
GPU (Communication)	7.42ms	1.78ms	2.25ms	6.31ms
GPU (Kernel)	0.35ms	1.65ms	0.35ms	3.43ms
CPU (Kernel)	17.1ms	10.4ms	89.4ms	1013.2ms
Speed Up (System)	2.2	3.03	34.38	104.01
Speed Up (Kernel)	48.85	6.3	255.42	295.39

Now, we will discuss the benchmark results. The results are displayed in tables 1, 2 & 3. In these tests we measured the execution time of purely the kernel, as well as the time necessary for copying data from the host to GPU + execution time as System performance for the GPU. In these benchmarks, the optimizations from section 4.1 have been fully implemented.

As we can see, looking at system performance the speed up factor is the greatest for `triangularSmooth`. This is, because the amount of parallelism and complexity in this algorithm is greater than the other algorithms. This causes the runtime of the kernel to overshadow the communications overhead on the GPU. The GPU performs worst in `histogram` and `rgb2gray`, where the atomic add

L1/Shared Memory		
Local Loads	0	0 B/s
Local Stores	0	0 B/s
Shared Loads	0	0 B/s
Shared Stores	0	0 B/s
Global Loads	12507160	57.495 GB/s
Global Stores	223456	777.371 MB/s
Atomic	0	0 B/s
L1/Shared Total	12730616	58.272 GB/s

Figure 3: Memory access statistics without utilizing shared memory

L1/Shared Memory		
Local Loads	0	0 B/s
Local Stores	0	0 B/s
Shared Loads	5586400	419.653 GB/s
Shared Stores	13966	1.049 GB/s
Global Loads	6937491	103.282 GB/s
Global Stores	223456	2.098 GB/s
Atomic	0	0 B/s
L1/Shared Total	12761313	526.083 GB/s

Figure 4: Memory access statistics with shared memory implemented

operation hampers performance, and the kernel runtime for rgb is simply too short to hide the communications overhead.

When we look at the speed up for solely the kernel, we see that the gains are much higher. Now we see that `histogram` is the only application that is still bottlenecked by the atomic add. The rgb kernel sees some amount of speed up, but not nearly as much as the more complex benchmarks. The greatest performance gain we achieved when looking at purely the kernel runtime is 313× on the largest image (image 09), which is about 33% of the theoretical maximum. With communications overhead we see a maximum speed up of only 117× again on the same image.

5 Conclusion

In conclusion, we processed images in 2 different ways. We started with a sequential implementation and accelerated this, by parallelizing it with the CUDA API. At most, we obtained speed-up factor around 300×, which is about 33% of what we theoretically expected. The main conclusion we can draw from this, is that the more taxing the application, the greater the speed up over the sequential implementation. For small tasks, the communication overhead destroys any gain we would get from using this highly parallel processor.

We mainly used the NVIDIA Visual Profiler to spot bottlenecks based on our first implementation. From the provided information regarding memory accesses, occupancy and type of instructions performed we made changes to our code. The optimization that resulted in the greatest amount of performance improvement was using shared memory. In the histogram kernel we got much higher occupancy and therefore higher performance. In the triangular smoothing algorithm, we mainly saw a significant boost by a 4× in crease in bandwidth.

Some recommendations for further improvement would be to investigate how we can alleviate the bottleneck regarding the atomic add operation in the histogram kernel. We improved performance somewhat by the use of shared memory, but we think there might be better result, if more time is spent investigating this.