Delft University of Technology

Advanced Computing System EE4C07

LAB REPORT

Introduction to Image Processing

Group 22

Author: *Email:*

Ade Setyawan Sajim (4608232) A.Sajim@student.tudelft.nl

Priadi Teguh Wibowo (4625862) PriadiTeguhWibowo@student.tudelft.nl

11 October 2016

Contents

1 Introduction

Image processing is an activity of process images using mathematical operations by using any form of signal processing for which the input is an image, a series of images, or a video, while the output of image processing may be either an image or a set of properties related to the image [2].

1.1 CUDA for Image Processing

CUDA® is a parallel computing platform and programming model invented by NVIDIA that enables dramatic increases in computing performance by harnessing the power of the graphics processing unit (GPU) [3]. In this lab, CUDA was chosen as the programming environment to implement the image processing [1]. The task is to use a color image as an input, converts it to gray scale, create the image's histogram and use it to improve the contrast of the gray scale image and finally create a smoothed gray scale version as the final version of the input image. Below are several task needed to be done to complete the lab assignment :

- Phase 1 : Converting a color image to grayscale
  Input images are RGB images (consists of three components : Red, Green and Blue). The gray value of a pixel is given by weighting these three values and then summing them together using formula

$$gray = 0.3 * R + 0.59 * G + 0.11 * B \tag{1}$$

- Phase 2 : Histogram computation
  The histogram shows how often a value of gray is used in an image. To compute the histogram, count the value of every pixel and increment the corresponding counter. There are 256 possible values of gray.
- Phase 3 : Contrast enhancement
  In this phase, the histogram that is computed on previous phase is used to determine which the lowest (darkest) and highest (lightest) gray values that is recorded in the histogram above a certain threshold. Pixels whose values are lower than min are set to black, pixels whose values are higher than max are set to white, and pixels whose values are inside the interval are scaled.
- Phase 4 : Smoothing
  To do smoothing (removal noise from a picture), each point is replaced by a weighted average of its neighbors. In this lab assignment, algorithm that is

used is a triangular smoothing algorithm, which count the maximum weight is for the point in the middle and decreases linearly moving from the center.

2 Implementation

2.1 Converting Color Image to Grayscale

This program simply calculate the combination of red, green and blue value of the image into greyscale in each thread. The kernel has 32*32 threads and width/32 * height/32 blocks to calculate the converting concurrently.

2.2 Histogram Computation

In this part of the lab, the parallel program consist of 2 kernel CUDA function. By dividing the kernel into two parts, reading/writing to same histogram values in Histogram computations could be prevented.

The first function calculates the histogram of every 256 pixel of the picture. Every 256 pixel will be stored in shared memory, so every threads in a block could iterate the shared memory variable to find the histogram value. There will be around height*width/256 of value for each histogram value (0 - 255) and it is stored on new variable to be passed to the second kernel.

The second kernel is also used as global synchronization. In second kernel with 256 threads, the kernel will counts the total each histogram value (0 - 255) in height*width/256 iteration. By doing this, there will be no writing data concurrently on the histogram output. Finally, the results will be stored into histogram output.

2.3 Contrast Enhancement

In this program, each threads calculates each pixel directly using threshold and scaling for the value between the defined max and min value. The kernel has 32*32 threads and width/32 * height/32 blocks to calculate the converting concurrently.

2.4 Smoothing

Because this program calculates based on the 5*5 of surrounding pixel, the total threads would be 25. Each threads will calculates the multiplication between the filter weight and the surrounding pixel. Then each block will sum it up and store the value to the smooth output image.

3 Results

On this assignment, all functions are successfully implemented. Results related to kernel time on CPU are available on table 1, table 2 and table 3. Results related to kernel time on GPU are shown on table 4, table 5, and table 6. Total time results are shown on table 8, table 9 and table 10.

To check the effectiveness of program, speed up comparison is done by using equation 2 and equation 4. The results is shown on table 7.

$$S(OverallSpeedup) = \frac{TotalTimeonCPU}{TotalTimeonGPU} \tag{2}$$

$$TotalTime = ExecutionTime + CommunicationTime \tag{3}$$

$$S(KernelSpeedup) = \frac{KernelExecutionTimeonCPU}{KernelExecutionTimeonGPU} \tag{4}$$

3.1 Environment Set Up

When choosing block dimension, there are several factors needed to be considered. In this lab session, the type of GPU used in running the application is GeForce GTX 750 Ti. This device has several specific characteristics, which are

- Maximum warp size: 32
- Maximum threads per block: 1024
- Max block dimensions: [1024, 1024, 64]
- Max grid dimensions: [2147483647, 65535, 65535]
- Global memory: 2047mb
- Shared memory: 48kb
- Constant memory: 64kb
- Block registers: 65536

Based on this characteristics, 2D block dimension design was chosen because it will be easier to calculate our problem since image itself is a 2D object. The size should be a round multiple of the warp size [4], thus on phase 1 and phase 3, the block size is set to 32x32 threads. The grid dimension will be adapted according to size of the picture (e.g. width of grid = picture width / block width, height of grid = picture height / block height,).

As additional implementation to optimize the GPU performance, the flags "-Xptxas -dlcm=ca" is added on the compiler. This flag will increase the amount of L1 cache to 48KB at compile time,

3.2 Measurement

Table 1: Kernel Function Execution Time on CPU (Image 15) in second

Function	**1**	**2**	**3**	**Average**
rgb2Gray	0.009125	0.008901	0.008888	0.008971
histogram1D	0.007433	0.007638	0.007624	0.007565
contrast1D	0.045262	0.046531	0.046757	0.046183
triangularSmooth	0.475656	0.493252	0.475028	0.481312

Table 2: Kernel Function Execution Time on CPU (Image 9) in second

Function	**1**	**2**	**3**	**Average**
rgb2Gray	0.095983	0.096348	0.120832	0.104388
histogram1D	0.102248	0.103671	0.104198	0.103373
contrast1D	0.373514	0.374070	0.374849	0.374144
triangularSmooth	4.979879	5.211726	5.148846	5.113484

Table 3: Kernel Function Execution Time on CPU (Image 4) in second

Function	**1**	**2**	**3**	**Average**
rgb2Gray	0.002360	0.002477	0.002510	0.002449
histogram1D	0.002782	0.002903	0.002919	0.002868
contrast1D	0.009914	0.010291	0.010363	0.010189
triangularSmooth	0.110917	0.11587	0.116613	0.114467

Table 4: Kernel Function Execution Time on GPU (Image 15) in second

Function	**1**	**2**	**3**	**Average**
rgb2GrayCudaKernel	0.000040	0.000041	0.000010	0.000030
histogram1DCudaKernel	0.000026	0.000023	0.000022	0.000024
contrast1DCudaKernel	0.000024	0.000021	0.000021	0.000022
triangularSmoothCudaKernel	0.000017	0.000016	0.000016	0.000016

Table 5: Kernel Function Execution Time on GPU (Image 9) in second

Function	**1**	**2**	**3**	**Average**
rgb2GrayCudaKernel	0.000039	0.000039	0.000038	0.000039
histogram1DCudaKernel	0.000029	0.000029	0.000029	0.000029
contrast1DCudaKernel	0.000024	0.000025	0.000025	0.000025
triangularSmoothCudaKernel	0.000019	0.000019	0.000019	0.000019

Table 6: Kernel Function Execution Time on GPU (Image 4) in second

Function	**1**	**2**	**3**	**Average**
rgb2GrayCudaKernel	0.000027	0.000030	0.000027	0.000028
histogram1DCudaKernel	0.000017	0.000016	0.000017	0.000017
contrast1DCudaKernel	0.000022	0.000022	0.000022	0.000022
triangularSmoothCudaKernel	0.000018	0.000012	0.000013	0.000014

Table 7: Speed Up on Kernel Function

Function	**Image 15**	**Image 9**	**Image 4**
rgb2GrayCuda	294.70	2691.60	87.65
histogram1DCuda	319.65	3564.56	172.08
contrast1DCuda	2099.24	15168.01	463.15
triangularSmoothCuda	29468.08	269130.72	7986.05

From Table 7, it can be concluded that the speed up is succesfully achieved using CUDA library for various application as expected .

Table 8: Total Execution Time on GPU (Image 15) in second

Function	**1**	**2**	**3**	**Average**
rgb2GrayCuda	0.555244	0.551094	0.538375	0.548238
histogram1DCuda	0.051850	0.052073	0.051813	0.051912
contrast1DCuda	0.004681	0.004593	0.004666	0.004647
triangularSmoothCuda	0.065016	0.064956	0.065483	0.065152

Table 9: Total Execution Time on GPU (Image 9) in second

Function	**1**	**2**	**3**	**Average**
rgb2GrayCuda	0.620804	0.619765	0.625084	0.621884
histogram1DCuda	0.438459	0.452193	0.453404	0.448019
contrast1DCuda	0.037570	0.037742	0.038098	0.037803
triangularSmoothCuda	0.604836	0.604839	0.608933	0.606203

Table 10: Total Execution Time on GPU (Image 4) in second

Function	**1**	**2**	**3**	**Average**
rgb2GrayCuda	0.548215	0.543001	0.539442	0.543553
histogram1DCuda	0.013190	0.013167	0.013170	0.013176
contrast1DCuda	0.001764	0.001826	0.001751	0.001780
triangularSmoothCuda	0.016837	0.016742	0.016735	0.016771

Table 11: Memory Transfer Time on GPU (Image 15) in second

Function	**1**	**2**	**3**	**Average**
rgb2GrayCuda	0.555204	0.551053	0.538365	0.548207
histogram1DCuda	0.051824	0.052050	0.051791	0.051888
contrast1DCuda	0.004657	0.004572	0.004645	0.004625
triangularSmoothCuda	0.064999	0.064940	0.065467	0.065135

Table 12: Memory Transfer Time on GPU (Image 9) in second

Function	**1**	**2**	**3**	**Average**
rgb2GrayCuda	0.620765	0.619726	0.625046	0.621846
histogram1DCuda	0.438430	0.452164	0.453375	0.447990
contrast1DCuda	0.037546	0.037717	0.038073	0.037779
triangularSmoothCuda	0.604817	0.604820	0.608914	0.606184

Table 13: Memory Transfer Time on GPU (Image 4) in second

Function	**1**	**2**	**3**	**Average**
rgb2GrayCuda	0.548188	0.542971	0.539415	0.543525
histogram1DCuda	0.013173	0.013151	0.013153	0.013159
contrast1DCuda	0.001742	0.001804	0.001729	0.001758
triangularSmoothCuda	0.016819	0.016730	0.016722	0.016757

According to table 11, table 12, and table 13, time needed to do memory transfer in rgb2grayCuda, compare to other cuda function, takes longer time. This phenomena happens because in rgb2grayCuda function, CUDA code is called for the first time and it will initialize its environment, thus takes a small period of time [5]. This proven by putting code "cudaSetDevice(1)" before start counting the time. Even though the code is only 1 line but since it involves initializing cuda environment, it takes relatively long time. On average, it takes 0.450706 s to load CUDA environment.

From [4], the GPU (GeForce 750 Ti) has maximum memory bandwidth 86.4 GB/sec and maximum computational 1,305.6 GFLOPS. When running the program, it achieves highest memory usage on rate 38.72 GB/s through global load throughput operation. On the other hand, it also achieves 643.46 GFLOPS. Compare to the highest possible, memory bandwidth achieves 44.81 % peak usage and computational performance has 49.28 % peak usage. There are some interesting result related to this program. Using nvprof profiling option, the program achieves global memory load, global memory store, warp execution and warp non predicated execution 100% efficiency. The implemented program also has a high multiprocessor activity (99.57 %). Facts mentioned above has shown that our program has successfully maximize GPU potential.

4 Conclusion

1. CUDA is a parallel programming library that utilizes NVIDIA GPU to do concurrent computation.
2. CUDA has various advantages as parallel programming library, such as :
   - It has Shared Memory which allocated per thread block and which can be accessed relatively fast.
   - CUDA has a method called __syncthreads() which can synchronize all threads execution in a block.
   - CUDA has plenty of useful tools to measure many kinds of parameter such as, throughput, maximum computation (in GFLOPS), memory usage, etc.
3. One of the disadvantage of CUDA is the memory transfer between GPU and CPU and device initialization. And in some application, it could be a bottleneck.

References

[1] Delft University of Technology. (2016). *Course EE4C07 Lab Manual Advanced Computing Systems*. Delft.

[2] Rafael C. Gonzalez; Richard E. Woods (2008). *Digital Image Processing*. Prentice Hall. pp. 1–3. ISBN 978-0-13-168728-8.

[3] *Nvidia CUDA Home Page*. http://www.nvidia.com/object/ . Retrieved October 5, 2016.

[4] GeForce GTX 750 Ti Specification. *http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-750-ti/specifications* Retrieved October 10, 2016.

[5] Cuda Programming Guide. http://docs.nvidia.com/cuda/cuda-c-programming-guide/initialization. Retrieved October 3, 2016.