EE4C07
Advanced Computing Systems
Lab 2: GPU-based Computing report

Yingli Ni	4515021	Ye Wang	4473582
Javier James	4170253	Luis Garcia Rosario	4062949

Delft University of Technology

Friday 9th October, 2015

TU Delft Delft University of Technology

Challenge the future

Abstraction

This is the report for the lab of the course Advanced Computing Systems at the TU Delft. During this course...

Contents

Chapter 1

Introduction

Improving the speed-up of a software application may sometimes be essential, especially in cases where there are computational intensive algorithms. With all the millions of transistors available on a single chip, it is possible to have more cores running in parallel. Engineering such software is very difficult. During the Lab of Advanced Computing Systems we of Group 07 have analyzed and experimented in depth the speed-up achievable and the performance in a provided Software application running on the **NVIDIA Geforce GTX750Ti**. This Software application uses this NVIDA GPU device to execute function kernels in parallel.

Method and Experiment During the experiments three images were tested individually, image04.bmp, image09.bmp and image15.jpg, having a size of 5.0 MB, 67.1MB, 5.0MB respectively. Each image were given as input to the cpu functions, and then to the gpu functions after. In this way we could analyze the effect of processing the entire image on the cpu vs gpu. The output result image smooth.bmp was then compared to see the difference. The % of error was denoted as the number of pixels above a set threshold, where both outputs differ from one another.

Report Organization The organization of this practical report is as following. Chapter 2 sets the mood by giving background information and analysis of the hardware and software. The potential speed-up and requirements will be discussed in short. Chapter **??** until 6 continues by going in depth analyzing the functions rgb2gray, histogram, contrast, and smoothing in chronological order. In each chapter we will give the speed-up attained for that kernel, the challenges encountered, and the validations of the results. Subsequently, chapter 7 provides the final results testing the entire application on the GPU and the achieved speed-up and image error. Finally, chapter **??** concludes this report by providing a summary of the most important points of the report and providing a discussion.

Chapter 2

Background

This chapter focuses on the hardware and software analysis. Section 2.1 begins analysing the targer GPU hardware providing its specification. Continually, section 2.2 analysis each software function to see its potential speed-up.

2.1 Hardware analysis

The target hardware is the NVIDIA Geforce GTX750Ti/Tesla C2075 GPU, which is present on the similde machine. This GPU uses uses first-generation NVIDIA Maxwell architecture. and the performance up to twice of previous generation graphics cards, however, it is only half the power consumption of previous generation products.the GTX 750TiGPU graphics card consits of 640 CUDA processor core and 1020MHz base frequency as well as 1085MHz enhance the frequency.

2.2 Software analysis

In this section we briefly describe the functionality of each function, display its potential speed-up as given by Amdahl's law in equation 2.2.1. As the number of cores increase to infinity the speed-up is upper bounded by the sequential part, which is given by equation 2.2.2.

$$Speedup = \frac{1}{1-f+\frac{f}{n}} \tag{2.2.1}$$

$$\leq \frac{1}{1-f} \tag{2.2.2}$$

2.2.1 RGB2GRAY algorithm

Function Description This is the original code that can converst the image to gray. According to the code , we can get cuda code and kernel function.

We see the pixel as the matrix which define y and x as the height and width of the image.we assume different color has different pixels function and float type in the same place.So, we can just use function and add these float index together. The most important thing is that we need to static transform the type of the variable because the input image type is char when the r,g,b is float. In the end , we need to change float to char.

```
for ( int y = 0; y < height; y++ )
 { for ( int x = 0; x < width; x++ )
        {    float grayPix = 0.0f;
             float r = static_cast< float >(inputImage[(y * width)
                 + x]);
             float g = static_cast< float >(inputImage[(width *
                height) + (y * width) + x]);
             float b = static_cast< float >(inputImage[(2 * width *
                 height) + (y * width) + x]);
             grayPix = (0.3f * r) + (0.59f * g) + (0.11f * b);
             grayImage[(y * width) + x] = static_cast< unsigned
                char >(grayPix);
     }
}
```

Listing 2.1: ... Yingli put a caption namehere

This code has three parts. First is the for loop. Pointer moves through the image height. Second is the for loop that moves through the image wight,the third part is calculating function and adding the variable.the whole function is based on the loop function . This is the basic theory of the RGB to gray image.we need to use the main function to program kernel function and recall the kernel function at the cuda function.

Max speed-up The for loop which a kernel was created from was 98% of the entire function. Therefore, according to equation 2.2.2 there is a maximum speed is bounded by x50. This is shown in 2.2.3.

$$Speedup_{max-rgb2gray} = \frac{1}{1-0.98} = \frac{1}{0.02} = 50 \tag{2.2.3}$$

2.2.2 Histogram algorithm

Function Description In histogram part, we have to static how many times each grayscale appears in the image. That means, we should have 256 counters for each grayscale and visit every point in the image in order. These counters use a series of continuous addresses, so we could change the value of it easily. But this would take a long time if the size of the image is large. I will show what the function it should have from software aspect by an example in CPU version.

```
  for ( int y = 0; y < height; y++ )
  {
    for ( int x = 0; x < width; x++ )
    {
      histogram[static_cast< unsigned int >(grayImage[(y * width) +
          x])] += 1;
    }
  }
```

Listing 2.2: Histogram algorithm analysis

This code could be seen as three parts. First is the for loop. Pointer moves through the image. Second is visiting memory which has stored grayscale of each point in the image. Third is change the value of the counters. Array histogram is used as the counter to record how many times each grayscale appears in the image.

Other parts of histogram is to visualize statistical results. There is no need to use parallel operation to optimize it.

Max speed-up The for loop which a kernel was created from was 23% of the entire function. Therefore, according to equation 2.2.2 there is a maximum speed is bounded by x50. This is shown in 2.2.4.

$$Speedup_{max-histogram} = \frac{1}{1-0.23} = \frac{1}{0.77} = 1.30 \tag{2.2.4}$$

2.2.3 Contrast algorithm

Function Description The Contrast routines takes as input a grayImage and histogram of length 256 and a contrast threshold value. A contrast is set on the grayImage base on the histogram and the contrast threshold.

The function begins by finding the index for which the index value is no longer less than the threshold value. This is done two times, from firsts index to the last, then from the last to the first. Respectively, these index will be min and max. Once these index are found, each pixel will be updated.

Max speed-up The for loop which a kernel was created from was 46% of the entire function. Therefore, according to equation 2.2.2 there is a maximum speed is bounded by x1.85. This is shown in 2.2.5.

$$Speedup_{max-contrast} = \frac{1}{1-0.46} = \frac{1}{0.54} = 1.85 \tag{2.2.5}$$

2.2.4 Smooth algorithm

Function Description In phase 4 of the image processing pipeline we can see how a triangular smoothing algorithm is used to remove noise from a gray-scale image. Each pixel of an input image is replaced by a weighted average of its neighboring pixels to remove structures from the image.

Max speed-up The for loop which a kernel was created from was 81% of the entire function. Therefore, according to equation 2.2.2 there is a maximum speed is bounded by x5.26. This is shown in 2.2.6.

$$Speedup_{max-smooth} = \frac{1}{1-0.81} = \frac{1}{0.19} = 5.26 \tag{2.2.6}$$

Chapter 3

Converting a color image to grayscale

Slight description of how function works

What was changed to create parallelism

challenges encountered

3.1 Implementation strategy

In RGB2GRAY part, we have to use the function to convert the RGB image to gray image. Firstly, we can use thread to program code. We define the index and the size of it. on the other hand, we can see the input image as the matrix and just use thread to define and calculation. The thread is typical processing by one core of the GPU. And the block is Composed by multiple threads (which can be expressed as a one-dimensional, two-dimensional, three-dimensional, and then elaborate on the specific below). Each block is executed in parallel, inter-block can not communicate, there is no execution order.we need also notice that the number of thread blocks is limited to no more than 65,535 because of the hardware limitations.And the grid is composed by multiply blocks. In this part of code we use thread to define the index. Here is the code.

```
1   unsigned int index = blockIdx.x * blockDim.x + threadIdx.x;
2         if (index < ImageSize)
```

This part code is defined and programmed in the kernel function. it will be easy to calculate because we donâĂŹt defined two or more thread and the calculate function is also linear calculation.the speed of the RGB is depended on the different size of the image.

3.2 Kernel speed-up achieved

Input image	CPU Time (sec)	GPU time (sec)	Speed-up
image 04	0.013690	0.001081	x12.67
image 09	0.483947	0.007024	x68.90
image 15	0.053946	0.001512	x35.67

Table 3.2.1: Kernel speed up using RGB2GRAY on GPU

3.3 Validation of results

Input image	Pixels above threshold	error %
image 04	864	0
image 09	0	0
image 15	624	0

Table 3.3.1: RGB2GRAY error comparison CPU vs GPU

3.4 Optimizations technique used

When we use parallel computing to solve this problem, we want to use threads to finish the computing in O(1). Like following code.

```
1     unsigned int index = blockIdx.x * blockDim.x + threadIdx.x;
```

we can assume that When the use of GPU accelerated image processing, the RGB image format generally 3-channel image, each through one byte, or 24-bit pixel image.we can expand the input image size to 3* width*height. While cuda access data elements, if each thread access 8bit, 16bit, 32bit, when 64bit, corresponding to the length of the data segment can be 32Byte, 64Byte, 128Byte, 128Byte.So we can meet the requirements to access global memory consolidation, improving access to global memory access performance.However, If each thread to access 24bit will not meet the requirements to access global memory consolidation, affecting the performance of the global memory access, so we can find solution that we expand the data size when we deal with the 24bit data as 32bit data access. So in the RGB part we assume the block size is 512 in order to suit for different sizes image.

3.5 Compiler options

Without any optimization option, the compiler's goal is to reduce the cost of compilation and to make debugging produce the expected results. So, for our program, we need to choice the best options.a slightly different set of optimizations may be enabled at each-Olevel than those listed here.But for our program we can find the best option in the make file document,we can find the fellow code:

```
1  debug     := 0
2    Ifneq   ($(debug), 1)
3   CFLAGS     := -O3 -g0
4   NVCCFLAGS  += -O3
5  else
6   CFLAGS     := -O0 -g3 -Wall
7   NVCCFLAGS  += -O0
8  End
```

So -03 is the option that we used during the debug.we can find the definition about -03option is Optimizing yet more.And -O3turns on all optimizations specified by-O2and also turns on the -finline-

functions, funswitch-loops,-fpredictive-commoning,-fgcse-after-reload,-ftree-loop-vectorize,-ftree-loop-distribute-patterns,-ftree-slp-vectorize,-fvect-cost-model,-ftree-partial-preand-fipa-cp-cloneoptions and such functions. We can see -03 use lots of functions and for us this is the best option during the debug. Here is the description of some functions. -ftree-loop-distribute-patterns Perform loop distribution of patterns that can be code generated with calls to a library. This flag is enabled by default at-O3. This pass distributes the initialization loops and generates a call to memset zero. For example, the loop

```
1  DO I = 1, N
2            A(I) = 0
3            B(I) = A(I) + I
4  ENDDO   is transformed to
5           DO I = 1, N
6              A(I) = 0        ENDDO
7           DO I = 1, N
8              B(I) = A(I) + I
9   ENDDO
```

the initialization loop is transformed into a call to memset zero. -ftree-loop-vectorize Perform loop vectorization on trees. This flag is enabled by default at-O3and when-ftree-vectorizeis enabled. -ftree-slp-vectorize Perform basic block vectorization on trees. This flag is enabled by default at-O3and when-ftree-vectorizeis enabled.

Compiling multiple files at once to a single output file mode allows the compiler to use information gained from all of the files when compiling each of them. Conclusion is that we invoke gcc with -03 as our best option because it use and transfer lots of functions that we can promote our program.

Chapter 4

Histogram computation

4.1 Implementation strategy

What we use to implement GPU version is atomic operation. In concurrent programming, an is atomic, linearizable, indivisible or interruptible if it appears to the rest of the system to occur instantaneously. Atomicity is a guarantee of isolation from concurrent processes. When a thread uses atomic operation to read or write a memory unit, this memory unit would be locked during this time. Other threads would have no ability to read or write this memory unit. They have to wait until the first thread have finished its operation and unlock the memory unit. There are lots of atomic operations in cuda. We use atomicAdd() in this part of the program. The code shows below.

```
unsigned int index = blockIdx.x * blockDim.x + threadIdx.x;
        if (index < ImageSize)
        {
                unsigned char Item = d_grayImage[index];
                atomicAdd (&(device_histogram[Item]), 1);
        }
```

Listing 4.1: ..

We will shorten the running time because of using atomicAdd() function. The time complexity of serial computing is $O(n^2/N)$, n is the size of each image and N is the number of threads. This also waste some computing resource of GPU. No matter how many threads can be used, the speed of histogram is limited by the size of the histogram.

4.2 Kernel speed-up achieved

Input image	CPU Time (sec)	GPU time (sec)	Speed-up
image 04	0.003524	0.003260	x1.08
image 09	0.165254	0.078573	x2.10
image 15	0.008705	0.005942	x1.47

Table 4.2.1: Kernel speed up using Histogram on GPU

4.3 Validation of result

Input image	Pixels above threshold	error %
image 04	0	0
image 09	0	0
image 15	0	0

Table 4.3.1: Histogram error comparison CPU vs GPU

4.4 Problems

In histogram part, we are asked to count points number of each grayscale. When we use CPU do serial arithmetic, it is very obvious that we can access memory of the grayscale of each point in the image in order and then add the counter. The time complexity of serial computing is O(n$\hat{2}$), n is the size of each image.

When we use parallel computing to solve this problem, we want to use threads to finish the computing in O(1). Like following code.

unsigned int index = blockIdx.x * blockDim.x + threadIdx.x;

histogram[index]++;

But this is only an illusion. It could not be so easy. The first challenge we met is the memory conflict. To explain this challenge, we have to show the principle of parallel computing at first. In one unit time, every thread would have their own work. They would access the memory, process the data or modify the memory at the same time. In this case, we have n$\hat{2}$ memory units to read and 256 memory units to write. Because these are only 256 kinds of grayscale in enactment, some points may have the same grayscale. If we operate what we write above, it must have a lot of memory conflict. This means more than one threads would write in a memory at same time. We wonâĂŹt get accurate result after these chaotic operations.

4.5 Solutions

So what we use to solve this problem is atomic operation. In concurrent programming, an is atomic, linearizable, indivisible or uninterruptible if it appears to the rest of the system to occur instantaneously. Atomicity is a guarantee of isolation from concurrent processes. When a thread uses atomic operation to read or write a memory unit, this memory unit would be locked during this time. Other threads would have no ability to read or write this memory unit. They have to wait until the first thread have finished its operation and unlock the memory unit. There are lots of atomic operations in cuda. We use atomicAdd() in this part of the program. The code shows below. unsigned int index = blockIdx.x * blockDim.x + threadIdx.x; —-atomicMax() would be added later—- We will shorten the running time because of using atomicAdd() function. The time complexity of serial computing is O(), n is the size of each image and N is the number of threads.

Chapter 5

Contrast enhancement

5.1 Implementation strategy

The pixel update which is done for each pixel using a for loop of $O(n^2)$ have been used as the function kernel. Each thread on the GPU will be responsible for updating one pixel. This function was executed with no problem because they were no data dependencies.

5.2 Kernel speed-up achieved

The baseline speed for the cpu function was 0.061069 seconds. After we parallelized this for loop we got a speed up of

Input image	CPU Time (sec)	GPU time (sec)	Speed-up
image 04	0.013598	0.001125	x12.09
image 09	0.514815	0.011060	x46.547
image 15	0.063011	0.001814	x34.73

Table 5.2.1: Kernel speed up using Contrast on GPU

5.3 Validation of result

Input image	Pixels above threshold	error %
image 04	0	0
image 09	0	0
image 15	0	0

Table 5.3.1: Contrast error comparison CPU vs GPU

5.4 Challenges encountered

This routine was trivial. Calculations could have been done using a 1D array on the kernel. Furthermore, there was no data dependencies, between the indexes.

Chapter 6

Image smoothing

6.1 Implementation Strategy

In the image smoothing phase the application takes a noisy gray-scale image as an input, remove small-scale unwanted structure, and return the resulting filtered/smoothed image as output. The sequential time complexity of the smoothing algorithm only is O(n^4) due to the 4 layered for-loop in the function. This is a computational intensive function for the CPU, but then due to its parallel nature (operations performed per pixels) it's ideal for the GPU.

The easiest approach was to map the image's width and height to CUDA's two-dimensional index for threads where and operation on each pixel is approached by at least one thread. We started with a 256 threads (a 16x16 block) thread block. The number of blocks per grid was dynamically determined by the width and the height of the input gray-scaled noisy image. After copying the image from host to device memory with cuda memcopy function, the kernel was invoked with a number of thread blocks equal to $width/threadIdx.x$ in one dimension and a number of $height/threadIdx.y$ thread blocks in the other dimension.

6.2 Kernel speed-up achieved

The execution times of the tests for the cases with CUDA parallelizations and the cases without any CUDA parallelizations are given in Table 6.2.1. The speed-up achieved was calculated by: Speedup = T_old/T_new = $0.547109/0.009494 \approx 57.6$.

Input image	CPU Time (sec)	GPU time (sec)	Speed-up
image 04	0.129558	0.002254	x57.48
image 09	6.103695	0.078858	x77.40
image 15	0.547109	0.009494	x57.63

Table 6.2.1: Kernel speed up using Smooth on GPU

6.3 Validation of result

The % of error was denoted in the table below as the number of pixels above a set threshold. We validated the resulting output image using the 'compare' application that tests the sequential output vs the parallel output.

Input image	Pixels above threshold	error %
image 04	0	0
image 09	0	0
image 15	0	0

Table 6.3.1: Smoothing error comparison CPU vs GPU

Chapter 7

Final Results

In this chapter we will show show the final results of our experiments.

7.1 Software validation

To test the software, each cpu (gpu) function output will go to the input of the next cpu (gpu) function in the pipeline. However, the same input image is given two both cpu and gpu processing pipeline. The output image, smooth.bmp, from the cpu and gpu was compared with one another.

Input image	CPU Time (sec)	GPU time (sec)	Speed-up
image 04	0.511147	1.539530	x0.33
image 09	15.718547	29.692304	x0.52
image 15	1.523320	3.626021	x0.42

Table 7.1.1: Overal speed up

7.2 Validation of result

Input image	Pixels above threshold	error %
image 04	197562	3
image 09	28792137	14
image 15	238785	1

Table 7.2.1: all functions error comparison CPU vs GPU

Appendix A

Code for Converting a color image to grayscale

```
__global__ void rgb2grayCudaKernel(unsigned char *d_inputImage,
    unsigned char *d_grayImage, int ImageSize)
{
        unsigned index = blockIdx.x * blockDim.x + threadIdx.x;

  if(index < ImageSize)

            {
                        float grayPix = 0.0f;
                        float r = static_cast< float >(d_inputImage[
                            index]);
                        float g = static_cast< float >(d_inputImage[
                            ImageSize + index]);
                        float b = static_cast< float >(d_inputImage
                            [(2 * ImageSize) + index]);
                        grayPix = (0.3f * r) + (0.59f * g) + (0.11f *
                             b);
                        d_grayImage[index] = static_cast< unsigned
                            char >(grayPix);
        }
    }

void rgb2grayCuda(unsigned char *inputImage, unsigned char *
    grayImage, const int width, const int height)
{

        NSTimer kernelTime = NSTimer("kernelTime", false, false);
        NSTimer timer_setUp_Comm = NSTimer("
            setUp_Communication_Timee", false, false);
        memset(reinterpret_cast< void * >(grayImage), 0, width *
            height * sizeof(unsigned char));
```

```
23
24          unsigned char* d_grayImage= NULL;
25          unsigned char* d_inputImage= NULL;
26          int ImageSize = width * height;
27
28    timer_setUp_Comm.start();
29          checkCudaCall(cudaMalloc( (void **) &d_inputImage, (3*
               ImageSize) ));
30          checkCudaCall(cudaMalloc((void **) &d_grayImage, ImageSize)
               );
31          checkCudaCall(cudaMemcpy(d_grayImage, grayImage, ImageSize,
                cudaMemcpyHostToDevice));
32          checkCudaCall(cudaMemcpy(d_inputImage, inputImage, 3*
               ImageSize, cudaMemcpyHostToDevice));
33    timer_setUp_Comm.stop();
34
35    time_setup_comm[0]= timer_setUp_Comm.getElapsed();
36
37          kernelTime.start();
38          dim3 dimBlock(512);
39          dim3 dimGrid(ImageSize/(int)dimBlock.x);
40          rgb2grayCudaKernel<<<dimGrid, dimBlock>>>(d_inputImage,
               d_grayImage, ImageSize);
41          cudaDeviceSynchronize();
42          kernelTime.stop();
43
44    kernelGpuTime[0]= kernelTime.getElapsed();
45
46    timer_setUp_Comm.start();
47          checkCudaCall(cudaMemcpy(grayImage, d_grayImage, ImageSize,
                cudaMemcpyDeviceToHost));
48    timer_setUp_Comm.stop();
49
50    time_setup_comm[0] += timer_setUp_Comm.getElapsed();
51
52          checkCudaCall(cudaFree(d_grayImage));
53          checkCudaCall(cudaFree(d_inputImage));
54
55
56          cout << fixed << setprecision(6);
57          cout << "rgb2gray (gpu): \t\t" << kernelTime.getElapsed()
               << " seconds." << endl;
58
59  }
```

Appendix B

Code for Histogram computation

```
1  __global__ void histogram1DCudaKernel(int ImageSize, unsigned int *
       device_histogram, unsigned char *d_grayImage)
2  {
3    // set the pointer to every element in d_grayImage
4          unsigned int index = blockIdx.x * blockDim.x + threadIdx.x;
5          if (index < ImageSize)
6          {
7                  unsigned char Item = d_grayImage[index];
8                  //use atomic operation to solve problem of memory
                       conflict
9                  atomicAdd (&(device_histogram[Item]), 1);
10         }
11 }
12
13
14
15 void histogram1DCuda(unsigned char *grayImage, unsigned char *
       histogramImage,const int width, const int height,
16                                 unsigned int *histogram, const
                                       unsigned int HISTOGRAM_SIZE,
17                                 const unsigned int BAR_WIDTH)
18 {
19   // set the number of threads in a single block
20         dim3 threadBlockSize(BLOCK_MAX_THREADSIZE);
21         unsigned int max = 0;
22         int ImageSize = width * height;
23         NSTimer kernelTime = NSTimer("kernelTime", false, false);
24   NSTimer timer_setUp_Comm = NSTimer("setUp_Communication_Timee",
       false, false);
25
26
27         timer_setUp_Comm.start();
```

```
28          memset(reinterpret_cast< void * >(histogram), 0,
               HISTOGRAM_SIZE * sizeof(unsigned int));
29          // copy histogram to device_histogram
30          unsigned int* device_histogram = NULL;
31          checkCudaCall(cudaMalloc((void **) &device_histogram,
               HISTOGRAM_SIZE* sizeof(unsigned int)));
32          checkCudaCall(cudaMemcpy(device_histogram, histogram,
               HISTOGRAM_SIZE* sizeof(unsigned int),
               cudaMemcpyHostToDevice));
33          // copy grayImage to d_grayImage
34          unsigned char* d_grayImage = NULL;
35          checkCudaCall(cudaMalloc((void **) &d_grayImage, ImageSize)
               );
36          checkCudaCall(cudaMemcpy(d_grayImage, grayImage, ImageSize,
                cudaMemcpyHostToDevice));
37    timer_setUp_Comm.stop();
38
39    time_setup_comm[1]= timer_setUp_Comm.getElapsed();
40
41          kernelTime.start();
42          // set the number of blocks
43          dim3 BlockNum(width*height/threadBlockSize.x+1);
44          histogram1DCudaKernel<<<BlockNum, threadBlockSize>>>(
               ImageSize, device_histogram, d_grayImage);
45          cudaDeviceSynchronize();
46          kernelTime.stop();
47
48
49    kernelGpuTime[1]= kernelTime.getElapsed();
50
51    timer_setUp_Comm.start();
52          checkCudaCall(cudaMemcpy(histogram, device_histogram,
               HISTOGRAM_SIZE* sizeof(unsigned int),
               cudaMemcpyDeviceToHost));
53         timer_setUp_Comm.stop();
54
55    time_setup_comm[1] += timer_setUp_Comm.getElapsed();
56
57     checkCudaCall(cudaFree(device_histogram));
58    // find the largest number in histogram
59    for ( unsigned int i = 0; i < HISTOGRAM_SIZE; i++ )
60    {
61      if ( histogram[i] > max )
62      {
63        max = histogram[i];
64      }
65    }
```

```
66
67    for ( int x = 0; x < HISTOGRAM_SIZE * BAR_WIDTH; x += BAR_WIDTH )
68    {
69      unsigned int value = HISTOGRAM_SIZE - ((histogram[x / BAR_WIDTH
          ] * HISTOGRAM_SIZE) / max);
70
71      for ( unsigned int y = 0; y < value; y++ )
72      {
73        for ( unsigned int i = 0; i < BAR_WIDTH; i++ )
74        {
75          histogramImage[(y * HISTOGRAM_SIZE * BAR_WIDTH) + x + i] =
              0;
76        }
77      }
78      for ( unsigned int y = value; y < HISTOGRAM_SIZE; y++ )
79      {
80        for ( unsigned int i = 0; i < BAR_WIDTH; i++ )
81        {
82          histogramImage[(y * HISTOGRAM_SIZE * BAR_WIDTH) + x + i] =
              255;
83        }
84      }
85    }
86
87    cout << fixed << setprecision(6);
88    cout << "histogram1D (gpu): \t\t" << kernelTime.getElapsed() << "
        seconds." << endl;
89
90  }
```

Appendix C

Code for Contrast enhancement

```
__global__ void contrast1DKernel(unsigned char *grayImage, const int width, const int height,int min, int max, int diff, int grayImageSize)
{

  unsigned int index  = blockIdx.x * blockDim.x + threadIdx.x;

    //ensure we dont use more threads than image size
    if(index < grayImageSize)
    {

    unsigned char pixel = grayImage[index];

          if ( pixel < min )
          {
            pixel = 0;
          }
          else if ( pixel > max )
          {
            pixel = 255;
          }
          else
          {
            pixel = static_cast< unsigned char >(255.0f * (pixel - min) / diff);
          }
          grayImage[index] =  pixel;
    }
}

void contrast1DCuda(unsigned char *grayImage, const int width, const int height,
        unsigned int *histogram, const unsigned int HISTOGRAM_SIZE,
```

```
30          const unsigned int CONTRAST_THRESHOLD)
31 {
32
33   unsigned int i = 0;
34   NSTimer kernelTime = NSTimer("kernelTime", false, false);
35   NSTimer timer_setUp_Comm = NSTimer("setUp_Communication_Timee",
        false, false);
36
37
38   while ( (i < HISTOGRAM_SIZE) && (histogram[i] <
        CONTRAST_THRESHOLD) )
39   {
40     i++;
41   }
42   unsigned int min = i;
43
44   i = HISTOGRAM_SIZE - 1;
45   while ( (i > min) && (histogram[i] < CONTRAST_THRESHOLD) )
46   {
47     i--;
48   }
49   unsigned int max = i;
50   float diff = max - min;
51
52          int grayImageSize= width * height;
53
54         timer_setUp_Comm.start();
55          // Allocate device memory for grayImage
56          unsigned char *d_grayImage;
57          checkCudaCall(cudaMalloc((void **)&d_grayImage,
              grayImageSize));
58
59     // Copy host memory to device
60           checkCudaCall(cudaMemcpy(d_grayImage,grayImage,
                grayImageSize,cudaMemcpyHostToDevice));
61
62          // Setup execution parameters
63       dim3 threads(BLOCK_MAX_THREADSIZE);
64       dim3 grid(grayImageSize/threads.x);
65   timer_setUp_Comm.stop();
66
67   time_setup_comm[2]= timer_setUp_Comm.getElapsed();
68
69   // Kernel launch
70   kernelTime.start();
71   contrast1DKernel<<<grid,threads>>>(d_grayImage,width,height,min,
        max,diff,grayImageSize);
```

```
72        cudaDeviceSynchronize();
73    kernelTime.stop();
74
75    kernelGpuTime[2]= kernelTime.getElapsed();
76
77    timer_setUp_Comm.start();
78          // Copy result from device to host
79          checkCudaCall(cudaMemcpy(grayImage,d_grayImage,
              grayImageSize,cudaMemcpyDeviceToHost));
80    timer_setUp_Comm.stop();
81
82    time_setup_comm[2] += timer_setUp_Comm.getElapsed();
83
84
85    cout << fixed << setprecision(6);
86    cout << "contrast1DCUDA (gpu): \t\t" << kernelTime.getElapsed()
        << " seconds." << endl;
87
88          // clean device memory
89          cudaFree(d_grayImage);
90  }
```

Appendix D

Code for Image Smoothing

```
__global__ void triangularSmoothKernel(unsigned char *grayImage,
    unsigned char *smoothImage, const int width, const int height,
                    const float *filter)
{
  // The total number of threads per block times the number of
     blocks
  unsigned int i = blockDim.x * blockIdx.x + threadIdx.x; // read-
     only variable i
  unsigned int j = blockDim.y * blockIdx.y + threadIdx.y; // read-
     only variable j

  if(i < width && j < height)
  {
    unsigned int filterItem = 0;
    float filterSum = 0.0f;
    float smoothPix = 0.0f;

    for ( int fy = j - 2; fy < j + 3; fy++ )
    {
      for ( int fx = i - 2; fx < i + 3; fx++ )
      {
        if ( ((fy < 0) || (fy >= height)) || ((fx < 0) || (fx >=
           width)) )
        {
          filterItem++;
          continue;
        }
        smoothPix += grayImage[(fy * width) + fx] * filter[
           filterItem];
        filterSum += filter[filterItem];
        filterItem++;
      }
    }
```

```
28      smoothPix /= filterSum;
29      smoothImage[(j * width) + i] = static_cast< unsigned char >(
            smoothPix);
30    }
31  }
32
33  void triangularSmoothCuda(unsigned char *grayImage, unsigned char *
        smoothImage, const int width, const int height,
34                const float *filter)
35  {
36    NSTimer kernelTime = NSTimer("kernelTime", false, false);
37
38    // 2a. Allocate the memory on the GPU
39    unsigned char *d_grayImage;
40    unsigned char *d_smoothImage;
41    float *d_filter;
42    checkCudaCall(cudaMalloc((void **)&d_grayImage, width * height));
43    checkCudaCall(cudaMalloc((void **)&d_smoothImage, width * height)
          );
44    checkCudaCall(cudaMalloc((void **)&d_filter, sizeof(filter)/
          sizeof(const float)));
45
46    // 2b. Move data over (host memory to device memory) for the
          function to execute
47    // cudaMemcpy(void *dst, void *src, size_t nbytes, enum
          cudaMemcpyKind direction);
48    checkCudaCall(cudaMemcpy((void *)d_grayImage, (void *)grayImage,
          (cudaMemcpyKind)width*height, cudaMemcpyHostToDevice));
49    checkCudaCall(cudaMemcpy((void *)d_smoothImage, (void *)
          smoothImage, (cudaMemcpyKind)width*height,
          cudaMemcpyHostToDevice));
50    checkCudaCall(cudaMemcpy(d_filter, filter, (cudaMemcpyKind)sizeof
          (filter)/sizeof(const float), cudaMemcpyHostToDevice));
51
52    // 3. Modify the function call in order to enable it to launch on
           the GPU
53    // Kernel invocation with one block of width * height * 1 threads
54    dim3 threadsPerBlock(16, 16);
55    dim3 numberOfBlocks(width/threadsPerBlock.x, height/
          threadsPerBlock.y);
56
57    kernelTime.start();
58    //Kernel invocation with N threads that executes it
59    triangularSmoothKernel<<<numberOfBlocks, threadsPerBlock>>>(
          d_grayImage, d_smoothImage, width, height, d_filter);
60    cudaDeviceSynchronize();
61    // /Kernel
```

```
62    kernelTime.stop();
63
64    // 4. Move data back over (device memory to host memory)
65    checkCudaCall(cudaMemcpy((void *)grayImage, (void *)d_grayImage,
          (cudaMemcpyKind)width*height, cudaMemcpyDeviceToHost));
66    checkCudaCall(cudaMemcpy((void *)smoothImage, (void *)
          d_smoothImage, (cudaMemcpyKind)width*height,
          cudaMemcpyDeviceToHost));
67
68    cout << fixed << setprecision(6);
69    cout << "triangularSmooth (cpu): \t" << kernelTime.getElapsed()
          << " seconds." << endl;
70
71    // Free up device memory
72    cudaFree(d_grayImage);
73    cudaFree(d_smoothImage);
74    cudaFree(d_filter);
75 }
```