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

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

Monday 12th October, 2015

Contents

1 Introduction **1**

I Assesment **2**

2 Background **3**

2.1 Hardware analysis 3

2.1.1 GPU calculation peak performance (GFLOPS) 3

2.1.2 GPU memory peak performance (Bandwidth) 3

2.2 Software analysis 4

2.2.1 rgb2gray algorithm 4

2.2.2 Histogram algorithm 5

2.2.3 Contrast algorithm 5

2.2.4 Smooth algorithm 6

II Implementation **7**

3 Converting a color image to grayscale **8**

3.1 Implementation strategy 8

3.2 Kernel speed-up achieved 8

3.3 Validation of results 9

3.4 Optimization technique used 9

3.5 Compiler options 9

4 Histogram computation **11**

4.1 Implementation strategy 11

4.2 Kernel speed-up achieved 11

4.3 Validation of result 12

4.4 Problems 12

4.5 Solutions 12

5 Contrast enhancement **14**

5.1 Implementation strategy 14

5.2 Kernel speed-up achieved 14

5.3 Validation of result 15

5.4 Challenges encountered 15

6 Image smoothing **16**
6.1 Implementation Strategy 16
6.2 Kernel speed-up achieved 16
6.3 Validation of result 17

7 Final Results Assessment phase **18**
7.1 Software validation 18
7.2 Performance analysis 19
7.2.1 Effective GFLOPS 19
7.2.2 Effective memory Bandwidth 19

III Optimizing **21**

8 Launch configuration/Occupancy analysis **22**
8.1 Occupancy 22
8.2 Launch configuration 23

9 Optimizing the GPU Software **25**
9.1 Launch configuration 25
9.2 Optimizing Contrast 25

10 Conclusion **27**

Bibliography **31**

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 3 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, before optimizing. Chapter 8 continues by analyzing the resources used in a kernel for each function to see where the kernels are bounded. Subsequently, chapter 9 highlights how the application was optimized. Finally, chapter 10 concludes this report by providing a summary of the most important points of the report and providing a discussion.

Part I

Assesment

Chapter 2

Background

This chapter focuses on the hardware and software analysis. Section 2.1 begins analyzing the target GPU hardware providing its specification. Continually, section 2.2 analyzes 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 consists of 640 CUDA processor core and 1020MHz base frequency as well as 1085MHz enhance the frequency.

2.1.1 GPU calculation peak performance (GFLOPS)

The peak GFLOPS for this hardware is 1305.60 GFLOPS. This was calculating using the formula in equation 2.1.1 and 2.1.2

$$GFLOPS_{theoretical} = cores * \frac{core}{socket} * clock * Flops/cycle \tag{2.1.1}$$

$$= 640 * 1020MHz * 2 = 1305.6GFLOPS \tag{2.1.2}$$

2.1.2 GPU memory peak performance (Bandwidth)

CUDA GPU code is limited by the memory transfers between host and device. There is data being transferred also between CPU and CPU memory, and GPU and GPU memory. The CPU memory bandwidth and the GPU memory bandwidth plays an important role on how the overall speedup of the GPU functions will be. In this section we focus on the GPU memory bandwidth.

The peak bandwidth of the GPU memory is 86.4 GB/s. This is found from ....., and can be calculated from the following calculation.

$$BW_{theoretical} = BlockSpeed(Gbps) * memorypumprate * \frac{buswidth}{8} \tag{2.1.3}$$

$$= 5.4Gbps * \frac{128}{8} = 86.4GB/s \tag{2.1.4}$$

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 converts 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++ )
// defined a for loop that pointer moves to the image height
  { for ( int x = 0; x < width; x++ )
 // defined a second for loop that moves to the image width. the
     for loop traverse the whole image.
        {     float grayPix = 0.0f; //pixel is float start from 0.0
              float r = static_cast< float >(inputImage[(y * width)
                   + x]);
              // the pointer moves to the first pixel of inputimage
                  ' red color  and casts the type of data to float.
             float g = static_cast< float >(inputImage[(width *
                height) + (y * width) + x]);
             // the pointer moves to the next pixel of inputimage '
                 green color  and casts the type of data to float.
             float b = static_cast< float >(inputImage[(2 * width *
                 height) + (y * width) + x]);
             // same process with red and green color.
             grayPix = (0.3f * r) + (0.59f * g) + (0.11f * b);
             // using the function about conversing RGB pixel to
                gray pixel.
             grayImage[(y * width) + x] = static_cast< unsigned
                char >(grayPix);
             //cast float graypix to char type.
       }
 }
```

Listing 2.1: Histogram analysis 2

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.

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

Part II

Implementation

Chapter 3

Converting a color image to grayscale

3.1 Implementation strategy

In rgb2gray phase, 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.

```
unsigned int index = blockIdx.x * blockDim.x + threadIdx.x;
      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)	Overall function speedup (comm + kernel)	Kernel Speed-up
image 04	0.013482	0.001078	x1.76	x12.51
.. image 09	0.456235	0.007005	x1.90	x 65.12
.. image 15	0.458113	0.007028	x1.91	x65.18

Table 3.2.1: Kernel speed up using rgb2gray on GPU

Input image	Total GPU time (Communication + Kernel)	% communication	% Kernel
image 04	86.09	13.91	
image 09	97.09	2.91	
image 15	97.07	2.93	

Table 3.2.2: Kernel rgb2gray: Communication vs kernel time

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

3.3 Validation of results

3.4 Optimization 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.

```
    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 phase we assume the block size is 512 in order to suit for different image sizes.

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 optimization 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:

```
debug    := 0
  Ifneq  ($(debug), 1)
 CFLAGS    := -O3 -g0
 NVCCFLAGS += -O3
else
```

```
 CFLAGS     := -O0 -g3 -Wall
 NVCCFLAGS += -O0
End
```

Compiler flag -03 is the option that was used for debugging. We can find in the definition about how the -03 option is Optimizing yet more. And -O3 turns on all optimization specified by-O2 and 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 that -03 uses 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, in this loop

```
 DO I = 1, N
           A(I) = 0
           B(I) = A(I) + I
 ENDDO   is transformed to
         DO I = 1, N
            A(I) = 0       ENDDO
         DO I = 1, N
            B(I) = A(I) + I
  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-O3 and when -ftree-vectorize is enabled. -ftree-slp-vectorize Perform basic block vectorization on trees. This flag is enabled by default at -O3 and when -ftree-vectorize is 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)	Overall function speedup (comm + kernel)	Kernel Speed-up
image 04	0.004636	0.003294	x4.60	x1.41
... image 09	0.151044	0.078660	x4.81	x1.92
... image 15	0.144489	0.078675	x4.77	x1.84

Table 4.2.1: Kernel speed up using Histogram on GPU

Input image	Total GPU time (Communication + Kernel)	% communication	% Kernel
image 04	20.15	79.85	
image 09	37.80	62.20	
image 15	37.77	62.23	

Table 4.2.2: Kernel histogram: Communication vs kernel time

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 uninterrupted 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)	Overall function speedup (comm + kernel)	Kernel Speed-up
image 04	0.014414	0.001150	x8.93	x12.54
... image 09	0.510283	0.011058	x11.17	x46.15
image 15	0.505441	0.011063	x11.11	x45.69

Table 5.2.1: Kernel speed up using Contrast on GPU

Input image	Total GPU time (Communication + Kernel)	% communication	% Kernel
image 04	69.27	30.73	
image 09	88.96	11.04	
image 15	88.92	11.18	

Table 5.2.2: Kernel contrast: Communication vs kernel time

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

5.3 Validation of result

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. As compiler option we invoke gcc with -03 as our best option.

Input image	CPU Time (sec)	GPU time (sec)	Overall function speedup (comm + kernel)	Kernel Speed-up
image 04	0.131464	0.002310	x14.68	x 56.90
.. image 09	5.932225	0.078823	x22.36	x75.25
.. image 15	5.969224	0.077140	x22.64	x77.38

Table 6.2.1: Kernel speed up using Smooth on GPU

Input image	Total GPU time (Communication + Kernel)	% communication	% Kernel
image 04	79.43	20.57	
image 09	75.00	35.00	
image 15	75.34	24.66	

Table 6.2.2: Kernel smooth: Communication vs kernel time

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 Assessment phase

In this chapter we will show the final results of our experiments, after implementing our cuda parallel code. To this point no optimization has been done to the code. The results in this chapter will later on be compared to the results attained in the optimization version.

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 the CPU's and the GPU's processing pipeline. The output image, smooth.bmp, from the CPU and GPU was compared with one another.

Overall speed-up The overall speed of the application is considered the ration of the execution time of the all functions running on the CPU vs those running on the GPU. The execution time of the GPU takes in account the setup + communication + kernel time as shown in table 7.1.1. The setup time is where the application used cudaMalloc to create memory for the variables, and is very time consuming and taking this into account will affect the overall speedup results. However, for multiple runs of the kernel this setup time is negligible and can also be ignored.

Validation We have tested the results to ensure that the results were okay. This validation is shown in table7.1.2. The largest error we received was 14 % and the best we have done was 1 %, leaving us with an average of 4.5 % error.

=

Input image	CPU time	GPU Time (Setup + Communication + Kernel)	Application Overall Speed-up above th
image 04	0.348393	1.429215	x0.24
image 09	15.518030	28.863783	x0.54
image 15	15.884527	28.539865	x0.56

Table 7.1.1: all functions speedup comparison CPU vs GPU

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

Table 7.1.2: all functions error comparison CPU vs GPU

Table 7.2.1: Effecitive vs Peak GFLOPS analysis

GFLOPS Analysis	**image 04**		**image 09**		**image 15**	
	effective	**%**	**effective**	**%**	**effective**	**%**
rgb2gray	12.403014	14.36	76.481398	88.52	37.954546	43.93
histogram 0	0	0	0	0	0	0
contrast	4.321759	5.00	18.181130	21.04	11.832858	13.70
smooth	0.173609	0.20	0.032511	0.04	0 0.086506	0.10

7.2 Performance analysis

7.2.1 Effective GFLOPS

The achieved GFLOPS was calculated according to equation 7.2.1

$$GFLOPS_{effective}(GB/s) = \frac{numOfThreads * numOfFloatingPointsOperations}{Kernelexecutiontime * 10^9} \tag{7.2.1}$$

Evaluating bandwidth efficiency Looking at table 7.2.2 it can be seen that rgb2gray achieved high % GFLOPS from the hardware peak of 1305.6GFLOPS as shown in equation 2.1.3. In general we have not achieved not much GFLOPS for the functions, but this could probably me limited by the number of threads used. Probably a better launch configuration will improve the GFLOPS a bit for the calculation bounded functions such as histogram, contrast, and smooth.

7.2.2 Effective memory Bandwidth

CUDA GPU code is limited by the memory transfers between host and device. There is data being transfered also between CPU and CPU memory, and GPU and GPU memory. The CPU memory bandwidth and the GPU memory bandwidth plays an important role on how the overall speedup of the GPU functions will be. In this section we focus on the GPU memory bandwidth.

The achieved bandwidth was measured running the application with image15. The throughput of the application will be given by equation 7.2.2 . Br and Bw is the total amount of bytes read and written in the kernel respectively. The achieved bandwidth is given in table 7.2.2.

$$BW_{effective}(GB/s) = \frac{Br(GB) + Bw(GB)}{Kernelexecutiontime(s)} \tag{7.2.2}$$

$$\tag{7.2.3}$$

Table 7.2.2: Effecitive vs Peak Bandwidth analysis

Bandwidth Analysis	**image 04**		**image 09**		**image 15**	
	effective	**%**	**effective**	**%**	**effective**	**%**
rgb2gray	6.234286	7.22	38.728179	44.82	18.944498	21.93
histogram	0.508842	0.59	0.853037	0.99	1.212488	1.40
contrast	2.802892	3.24	12.134791	14.04	7.899438	9.14
smooth	1.471311	1.70	1.699408	1.97	1.506127	1.74

Evaluating bandwidth efficiency Table 7.2.2 shows the comparison of each function bandwidth with respect to the peak bandwidth, when tested on the three different input image. The best we did was 44.82 % from the peak bandwidth, and the worst 0.59 %. It can be seen that it is difficult to achieve even 50 % of the maximum bandwidth. Most functions are bandwidth bounded. Data transfers between CPU and GPU is also determined by the CPU hardware which is not high and can play a role on the achieved bandwidth.

Part III

Optimizing

Chapter 8

Launch configuration/Occupancy analysis

In order to improve our performance even more we analyzed the possibility of altering the blocksize for the kernel. This was done using the NVIDA Occupancy calculator.

Occupancy is limited by three factors. register, shared memory, and block size. In the calculator we sued as reference compute capability 3.0 which has a shared memory per block of 48KB

8.1 Occupancy

rgb2gray This function used Used 12 registers, 340 bytes cmem[0], 8 bytes cmem[2]. Furthermore, only 512 threads per block was used. Results were 2048 Active threads, 64 active warps per multiprocessor, 4 threads block per multiprocessor, and an occupancy of 100 %. The occupancy limiter is te max warp or max blocks per multiprocessor. 64 warps per SM

So to increase the number of blocks per multiprocessor we will have to reduce the memory usage.

histogram This function used Used Used 6 registers, 344 bytes cmem[0]. Furthermore, only 512 threads per block was used. Results were 2048 Active threads, 64 active warps per multiprocessor, 4 threads block per multiprocessor, and an occupancy of 100 %. The occupancy limiter is te max warp or max blocks per multiprocessor. 64 warps per SM

So to increase the number of blocks per multiprocessor we will have to reduce the memory usage.

contrast This function used Used Used 11 registers, 352 bytes cmem[0], 8 bytes cmem[2]. Furthermore, only 512 threads per block was used. Results were 2048 Active threads, 64 active warps per multiprocessor, 4 threads block per multiprocessor, and an occupancy of 100 %. The occupancy limiter is te max warp or max blocks per multiprocessor. 64 warps per SM

So to increase the number of blocks per multiprocessor we will have to reduce the memory usage.

smooth This function used Used 16 registers, 352 bytes cmem[0], 8 bytes cmem[2]. Furthermore, only 512 threads per block was used. Results were 2048 Active threads, 64 active warps per multiprocessor, 4 threads block per multiprocessor, and an occupancy of 100 %. The occupancy limiter is te max warp or max blocks per multiprocessor. 64 warps per SM

So to increase the number of blocks per multiprocessor we will have to reduce the memory usage.

8.2 Launch configuration

In order to set a good launch configuration we have made used of the occupancy-based launch configuration API in our software. This API automatically calculates a launch configuration with the best occupancy, and is portable for each hardware. **??**

```
        int blockSize;   // The launch configurator returned block
           size
        int minGridSize; // The minimum grid size needed to achieve
            the
        // maximum occupancy for a full device launch
        int gridSize;    // The actual grid size needed, based on
           input size

      cudaOccupancyMaxPotentialBlockSize( &minGridSize, &blockSize,
                                          rgb2grayCudaKernel,
                                             0, 0);
      // Round up according to array size
     gridSize = (ImageSize + blockSize - 1) / blockSize;

        dim3 dimBlock(blockSize);
        dim3 dimGrid(gridSize);


        kernelTime.start();
        rgb2grayCudaKernel<<<dimGrid, dimBlock>>>(d_inputImage,
           d_grayImage, ImageSize);
        cudaDeviceSynchronize();
        kernelTime.stop();


        // calculate theoretical occupancy
        int maxActiveBlocks;
        cudaOccupancyMaxActiveBlocksPerMultiprocessor( &
           maxActiveBlocks,
                                                 rgb2grayCudaKernel
                                                    , blockSize,
                                                 0);

                int device;
        cudaDeviceProp props;
        cudaGetDevice(&device);
        cudaGetDeviceProperties(&props, device);

        float occupancy = (maxActiveBlocks * blockSize / props.warpSize) /
                    (float)(props.maxThreadsPerMultiProcessor /
                        props.warpSize);
```

Listing 8.1: Occupancy-based Launch configuration code

For rgb2gray, histogram, and contrast the blocksize have been increased to 1024 threads per block.

Chapter 9

Optimizing the GPU Software

To Optimize the code we kept in consideration of the memory and computation bound for each function as shown in table 7.2.1 and 7.2.2. Furthermore, what helped us in our decision is the occupancy result as shown in chapter 8

9.1 Launch configuration

Kernel	Old block size	New block size
rgb2gray	512	1024
histogram	512	1024
contrast	512	1024
smooth	16x16	16x16

Table 9.1.1: Launch configuration optimization

9.2 Optimizing Contrast

This kernel was memory an computation bounded. At its best it only achieved 14 % and 21 % from the peak bandwidth and GFLOPS respectively. Seeing that it is more bandwidth bounded, we focused first on optimizing the memory loads.

Using the new launch configuration we achieved a 3.4 % from the peak bandwidth and a 5.2% from peak GFLOPS, which was better than before.

Pinned memory was our initial approach as an optimization technique. In doing this we aimed to ensure that the execution time time of of data transfer be reduced. Testing on image 04 we achieved an effective bandwidth which was 1.94 % from the peak bandwidth. This result is less than the initial result in 7.2.2 which was. We believe this was caused due to the fact the kernel does not access the memory often, and thus it was not beneficial introducing the extra execution time creating the pinned memory. Despite this fact this method helps improving the PCI bandwidth. Previously we an application overall speed-up of x.24, now we have 0.x26. Not this is taking in consideration the setup time using cudamalloc and cuda memcopy, which is determined by the PCI bandwidth.

We did not do shared memory because we believed this will have diminishing returns. Not much memory fetches in done in this kernel. So copying the data from global the shared will just introduce extra execution time which will not be benefited from. We therefore focus now on improving the computation bound.

Chapter 10

Conclusion

Without optimization we have attained a maximum kernel speed-up of x65.12, overall kernel speed-up of x1.90. The overall kernel speed up took into account the ratio of the execution time running the application on the CPU vs on the GPU and looking only at the kernel time + the communication time. Including the start up time using data allocation we got a maximum speed-up of x56.

Each kernel was tested to see what type of bound it had, whether bandwidth or computational. This was done using the occupancy calculator and also manually from the source code. The results are shown in table **??** and **??**. It can be seen that most were memory bounded.

We began to optimize the code using the occupancy calculator and the occupancy-base latency launch configuration API. Using these methods we have optimized the the rgb2gray, histogram, and contrast function to automatically select the best block size where we will have optimum occupancy. Before we had a block size of 512, and now we have 1024. Testing the contrast function this gave a speed-up improvement.

For the contrast function we have also tried to optimize using pinned memory, but this did not improve the memory bandwidth or GFLOPS. It has however, improved the application overall speed-up, taking into account the start-up + communication + kernel time from the GPU code.

When optimizing the kernels it is important to know which metric one we'll like to improve. Improving one may cause the next to decrease. Depend which is of importance, the choice will be a good one. As in the case in using pinned memory for contrast.

Table 10.0.1: Effective vs Peak Bandwidth analysis

Bandwidth Analysis	**image 04**		**image 09**		**image 15**	
	effective	**%**	**effective**	**%**	**effective**	**%**
rgb2gray	6.234286	7.22	38.728179	44.82	18.944498	21.93
histogram	0.508842	0.59	0.853037	0.99	1.212488	1.40
contrast	2.802892	3.24	12.134791	14.04	7.899438	9.14
smooth	1.471311	1.70	1.699408	1.97	1.506127	1.74

Ze
Table 10.0.2: Effective vs Peak GFLOPS analysis

GFLOPS Analysis	**image 04**		**image 09**		**image 15**	
	effective	**%**	**effective**	**%**	**effective**	**%**
rgb2gray	12.403014	14.36	76.481398	88.52	37.954546	43.93
histogram 0	0	0	0	0	0	0
contrast	4.321759	5.00	18.181130	21.04	11.832858	13.70
smooth	0.173609	0.20	0.032511	0.04	0 0.086506	0.10

Bibliography

[1] *http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-750-ti/specifications*

[2] *http://bomadeno.com/2010/10/27/calculating-the-flops-of-your-nvidia-gpu/*

[3] *http://bomadeno.com/2010/10/27/calculating-the-flops-of-your-nvidia-gpu/*

[4] *http://devblogs.nvidia.com/parallelforall/cuda-pro-tip-occupancy-api-simplifies-launch-configuration/*

[5] *http://devblogs.nvidia.com/parallelforall/cuda-pro-tip-occupancy-api-simplifies-launch-configuration/*