Computer Architecture

Image Processing in CUDA

Project Report

Group Members:

Shakeeb Hussain Siddiqui	(k164043)	(Section GR-3)
Mehdi Raza Rajani	(k163904)	(Section GR-3)
Mohammad Ahmed Gul	(k163865)	(Section GR-3)
Moazzam Maqsood	(k163868)	(Section GR-)
Asim Iqbal	(k163894)	(Section GR-3)

Abstract

Image Processing is the use of computer algorithms to perform image processing on Digital Images. Image processing may mean either of the following:

1. Feature and Information Extraction.
2. Enhancing the Images.
3. Manipulation and Alteration of Images

Some image processing techniques are:

A. Gaussian Blur.
B. RGB to Grayscale.
C. Average Blur Filter
D. Image Inversion
E. Image Addition

Our project implements two of the above-mentioned techniques: RGB to Grayscale, Image Inversion and Image Addition.

This report puts to the test the power of parallel computing on the GPU against the massive computations needed in image processing of large images. The GPU has long been used to accelerate 2D and 3D high-resolution applications. With the advent of high-level programmable interfaces, programming to the GPU is simplified and is being used to accelerate a wider class of applications. More specifically, this report focuses on CUDA as its parallel programming platform.

This report explores the possible performance gains that can be achieved by using CUDA on image processing and implements three of the above-mentioned techniques: **Image Inversion, Image Addition** and **RGB to Grayscale**. Benchmarks are done between the parallel implementation and the sequential implementation.

Table of Contents

Introduction to Technology Used

CUDA (**C**ompute **U**nified **D**evice **A**rchitecture) C++ is just one of the ways you can create massively parallel applications with CUDA. It lets you use the powerful C++ programming language to develop high-performance algorithms accelerated by thousands of parallel threads running on GPUs. Many developers have accelerated their computation- and bandwidth-hungry applications this way, including the libraries and frameworks that underpin the ongoing revolution in artificial intelligence known as Deep Learning.

Why CUDA is ideal for image processing

A single high definition image can have over 2 million pixels. Many image processing algorithms require dozens of floating point computations per pixel, which can result in slow runtime even for the fastest of CPUs. The slow speed of a CPU is a serious hindrance to productivity. In CUDA, we can generally spawn exactly one thread per pixel. Each thread will be responsible for calculating the final colour of exactly one pixel. Since images are naturally two dimensional, it makes sense to have each thread block be two dimensional. 32×16 is a good size because it allows each thread block to run 512 threads. Then, we spawn as many thread blocks in the x and y dimension as necessary to cover the entire image. For example, for a 1024×768 image, the grid of thread blocks is 32×48, with each thread block having 32×16 threads.

Problems Solved Using CUDA

Our project implements two of the image processing techniques: RGB to GrayScale, Image Inversion and Image Addition which are described below with complete methodology, code snippet and example.

RGB to GrayScale:

RGB to GrayScale is to convert a coloured image to a black and white image. An RGBA image has 4 channels Red, Green, Blue and Alpha. (Alpha is for transparency, not used in the process). Each channel Red, Blue, Green and Alpha are represented by one byte. Since we are using one byte for each colour there are 256 different possible values for each colour. This means we use 4 bytes per pixel Grayscale images are represented by a single intensity value per pixel which is one byte in size.

Procedure:

One simple method to convert an image from colour to grayscale is to set the intensity to the average of the RGB channels. But we will use a more sophisticated method that takes into account how the eye perceives colour and weights the channels unequally.

The eye responds most strongly to green followed by red and then blue. The NTSC (National Television System Committee) recommends the following formula for the colour to grayscale conversion:

Grayscale-Pixel = 0.299f * Red + 0.587f * Green+ 0.114f * Blue

Code Snippets:

```
from scipy import misc
import matplotlib.pyplot as plt
import numpy as np

image = misc.imread('a.jpg')

print(type(image))

width_col,height_col,dim_col = image.shape

image_gray  = misc.imread('a.jpg')
image_gray = np.dot(image_gray[...,:3],[0.299, 0.587, 0.114])
width_gray,height_gray = image_gray.shape

# image_gray_dir  = misc.imread('a.jpg',mode="L")

f, axarr = plt.subplots(2)
axarr[0, 0].imshow(image)
axarr[0, 0].set_title('Image Color')

axarr[0, 1].imshow(image_gray,cmap = plt.get_cmap('gray'))
axarr[0, 1].set_title('Image converted to Gray')

# image is read in 2 dimension only ---> black an white image
# axarr[1, 0].imshow(image_gray_dir,cmap = plt.get_cmap('gray'))
# axarr[1, 0].set_title('Image directly read to gray')

plt.show()

misc.imsave("black_and_white_image.jpg",image_gray)
```

Figure 1a: RGB to GrayScale (Without Using GPU)

```
import pycuda.driver as drv
from pycuda.compiler import SourceModule
import numpy as np
import cv2
from PIL import Image
import pycuda.autoinit


mod = SourceModule \
  (
    """
#include<stdio.h>
#define INDEX(a, b) a*256+b

_global_ void bgr2gray(float *d_result,float *b_img, float *g_img, float *r_img)
{
 unsigned int idx = threadIdx.x+(blockIdx.x*(blockDim.x*blockDim.y));
 unsigned int a = idx/256;
 unsigned int b = idx%256;
 d_result[INDEX(a, b)] = (0.299*r_img[INDEX(a, b)]+0.587*g_img[INDEX(a, b)]+0.114*b_img[INDEX(a, b)]);

}
 """
)

h_img = cv2.imread('1.jpeg',1)
h_gray=cv2.cvtColor(h_img,cv2.COLOR_BGR2GRAY)
#print a
b_img = h_img[:, :, 0].reshape(60000).astype(np.float32)
g_img = h_img[:, :, 1].reshape(60000).astype(np.float32)
r_img = h_img[:, :, 2].reshape(60000).astype(np.float32)
h_result=r_img
bgr2gray = mod.get_function("bgr2gray")
bgr2gray(drv.Out(h_result), drv.In(b_img), drv.In(g_img),drv.In(r_img),block=(1024, 1, 1), grid=(64, 1, 1))

h_result=np.reshape(h_result,(300,200)).astype(np.uint8)

a=Image.fromarray(h_result)
a.save('/home/r2d2/PycharmProjects/untitled2/out.jpeg')

a.show()
```

Figure 1b: RGB to GrayScale (Using GPU)

Example:

Figure 2: RGB to GrayScale Input Image

Figure 3: RGB to GrayScale Result Image

Figure 4: RGB to GrayScale Result Image

Image Inversion:

Code Snippet:

```
import pycuda.driver as drv
import numpy as np
import cv2
import pycuda.gpuarray as gpuarray
import pycuda.autoinit
from PIL import Image

img = cv2.imread('out.jpeg',0)
h_img = img.reshape(60000).astype(np.float32)
d_img = gpuarray.to_gpu(h_img)
d_result = 255- d_img
h_result = d_result.get()
h_result=np.reshape(h_result,(300,200)).astype(np.uint8)

a=Image.fromarray(h_result)
a.save('/home/r2d2/PycharmProjects/untitled2/out1.jpeg')
```

Figure 5: Image Inversion

Examples

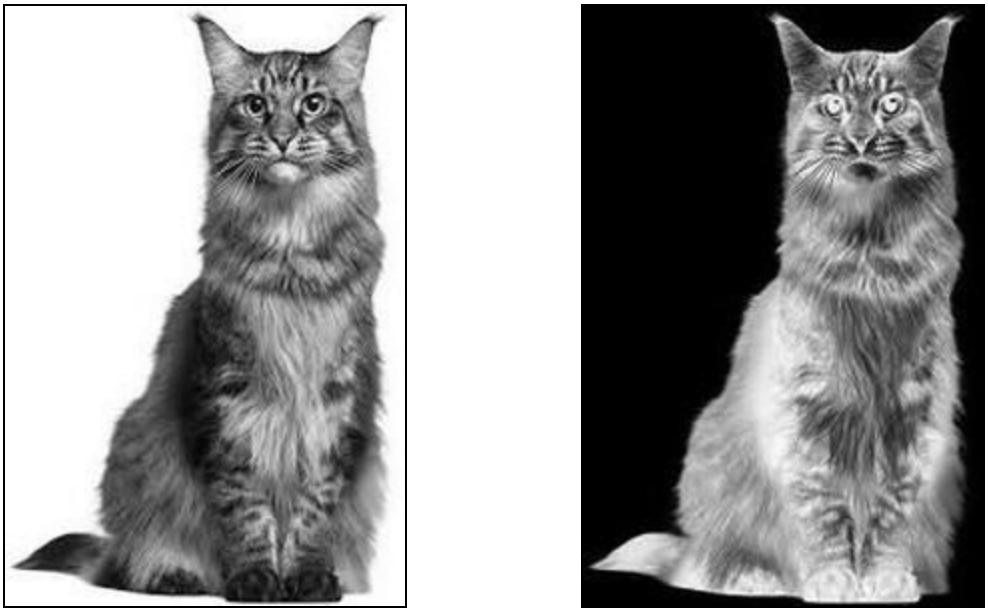


Figure 6: Image Inversion Result Image

Image Addition:

The addition of the two images can be performed when two images are of the same size. It performs a pixel-wise addition of two images.

Code Snippet:

```
import pycuda.driver as drv
from pycuda.compiler import SourceModule
import numpy as np
import cv2
from PIL import Image
import pycuda.autoinit

mod = SourceModule \
 (
"""
 _global_ void add_num(float *d_result, float *d_a, float *d_b,int N)
{
 int tid = threadIdx.x + blockIdx.x * blockDim.x;
 while (tid < N)
  {
 d_result[tid] = d_a[tid] + d_b[tid];
 if(d_result[tid]>255)
 {
 d_result[tid]=255;
 }
 tid = tid + blockDim.x * gridDim.x;
}
}
"""
)
img1 = cv2.imread('2.jpeg',0)
img2 = cv2.imread('out.jpeg',0)
h_img1 = img1.reshape(60000).astype(np.float32)
h_img2 = img2.reshape(60000).astype(np.float32)
N = h_img1.size
h_result=h_img1
add_img = mod.get_function("add_num")
add_img(drv.Out(h_result), drv.In(h_img1), drv.In(h_img2),np.uint32(N),block=(1024, 1, 1), grid=(64, 1, 1))
h_result=np.reshape(h_result,(300,200)).astype(np.uint8)

a=Image.fromarray(h_result)
a.save('/home/r2d2/PycharmProjects/untitled2/out3.jpeg')

a.show()
```

Figure 7: Image Addition

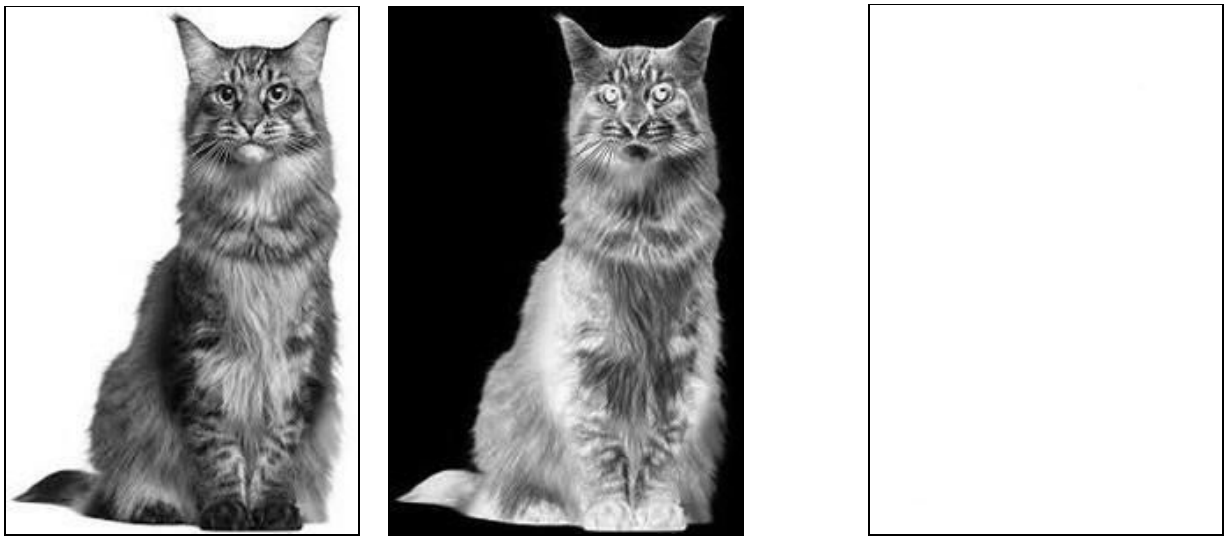
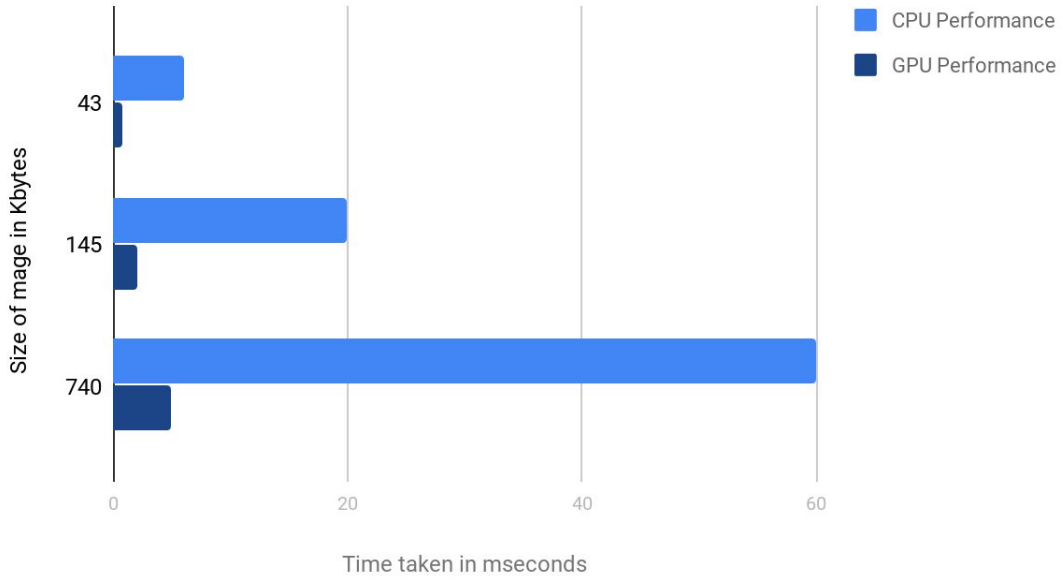


Figure 8: Image Addition Result Image

Result

RGB to Grayscale:

Figure 14: RGB to GrayScale Speed Comparison



Conclusion:

Graphics cards have widely been used to accelerate gaming and 3D graphical applications. High-level programmable interfaces now allow this technology to be used for general purpose computing. CUDA is the first of its kind from the NVidia tech chain. It is fundamentally sound and easy to use. This report gives an introduction to the type of performance gains that can be achieved by switching over to the parallel programming model.

References:

1. https://core.ac.uk/download/pdf/62876379.pdf
2. https://www.researchgate.net/publication/320894716_Parallel_Computing_With_Cuda_in_Image_Processing
3. http://supercomputingblog.com/cuda/intro-to-image-processing-with-cuda/
4. https://devblogs.nvidia.com/even-easier-introduction-cuda/
5. http://www.coldvision.io/2015/11/17/rgb-grayscale-conversion-cuda-opencv/
6. https://www.packtpub.com/mapt/book/application_development/9781789348293/12/ch12lvl1sec104/basic-computer-vision-operations-using-pycuda