Homework 1 report
Heng Zhang
0029109755

## Introduction

The two source images used in this homework are Figure 1 (1280* 720) and Figure 2 (1960 * 1080). They are “img0.jpg” and “img1.jpg” in the submission folder.


*Figure 1 1280 * 720*

*Figure 2 1960 * 1080*

There are three source code files in the submission folder: conv.py, main.py, and main.c. The python files are used in part A – C while the c file is used in part D.

## PartA

In total, there are 12 output plots where 6 plots originate from img0.jpg and the other 6 from img1.jpg by applying 6 different configurations to each plots.

### Config 1 (Task 1 kernel 1)

Figure 3 and 4 are generated by applying kernel 1 in task 1. Kernel 1 is a horizontal edge detector. Therefore, the horizontal edges are highlighted in these figures. The output images size in both height and width are decreased by 2 because the kernel has length of 3 and the stride is only 1 by the formula: (original_size – kernel length) / stride + 1

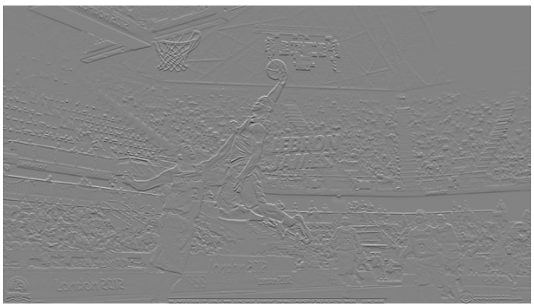

*Figure 3 img0 by Task 1 kernel 1*

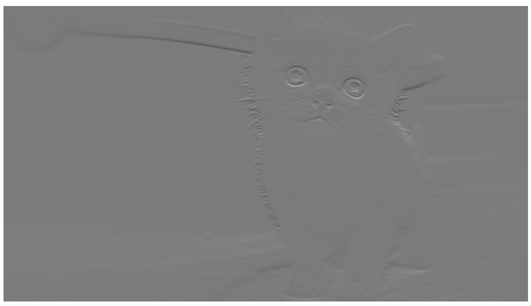

*Figure 4 img1 by Task 1 kernel 1*

## Config 2 (Task 2 kernel 4)

The effect in Config 2 compared with Config 1 is the size of the kernel. Kernel 4 is 5 by 5 which gives stronger horizontal edge detecting effect. The Figure 5 and 6 compared with Figure 3 and 4 have thicker horizontal edges

*Figure 5 img0 by Task 2 kernel 4*

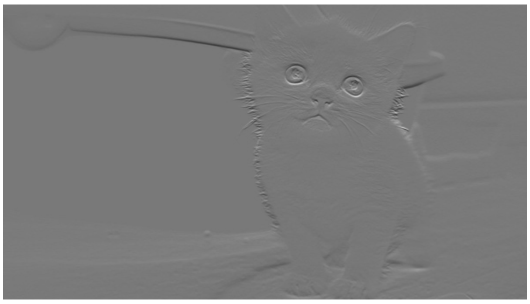

*Figure 6 img1 by Task 2 kernel 4*

## Config 3 (Task 2 kernel 5)

Kernel 5 is a vertical edge detector with size of 5 by 5. Therefore in Figure 7 and 8, we see thick vertical edges are detected.

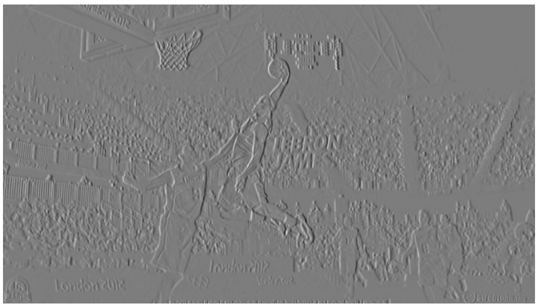

*Figure 7 img0 by Task 2 kernel 5*

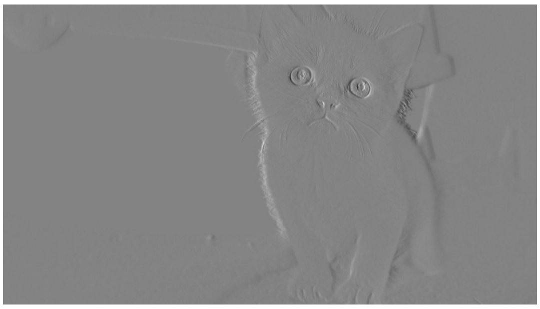

*Figure 8 img1 by Task 2 kernel 5*

## Config 4 (Task 3 kernel 1 stride 2)

Compared with Config 1, the change is the larger stride. By (original_size – kernel length) / stride + 1, the output figure will have smaller height and weight by roughly half of the the plots in Config 1.

*Figure 9 img0 by Task 3 kernel 1*

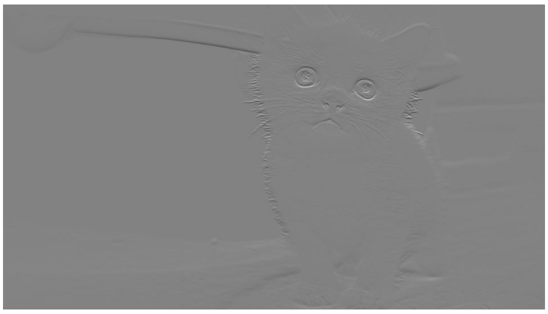

*Figure 10 img1 by Task 3 kernel 1*

## Config 5 (Task 3 kernel 2 stride 2)

Kernel 2 is a vertical edge detector. We see the Figures 11 and 12 are showing clear vertical edges.

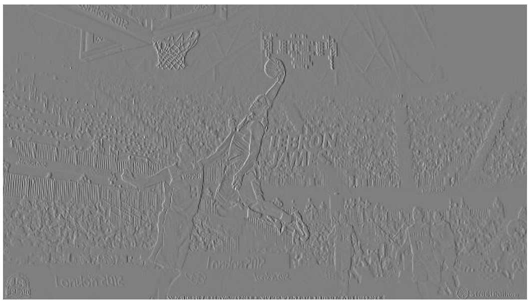

*Figure 11 img0 by Task 3 kernel 2*

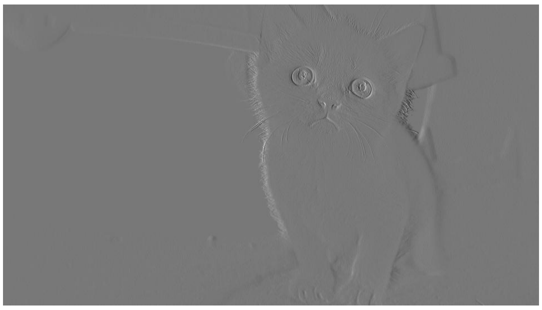

*Figure 12 img1 by Task 3 kernel 2*

## Config 6 (Task 3 kernel 3 stride 2)

Kernel 3 is an average detector because it has all 1s in all elements which means the convolution will be just an average of all three RGB channels. The resulting plots are Figure 13 and 14.


*Figure 13 img0 by Task 3 kernel 3*

*Figure 14 img1 by Task 3 kernel 3*

## Part B

This part is comparing the computation efficiency with respect to different number of output channels for both img0 and img1. The number of output channels is calculated by $2^i$ where i is the x axis of Figure 15 and 16. The computing time is in seconds. Clearly the computation time is exponentially increasing with i.

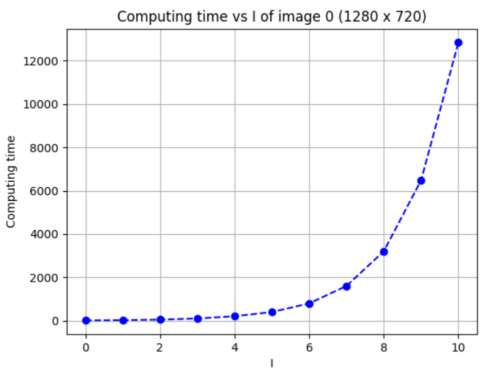


*Figure 15 img0 computation efficiency*

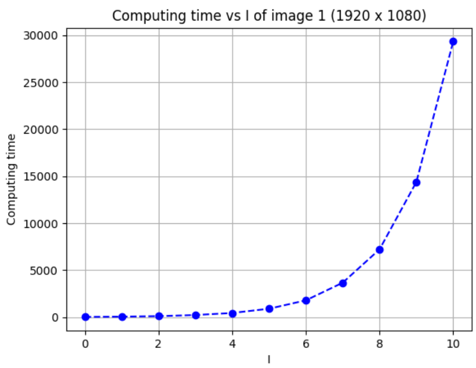


*Figure 16 img1 computation efficiency*

## Part C

This part compares the number of operations with respect to the kernel size. It is clear that the relation is similar to quadratic shape, which is the in correspondence to the total size of kernel because the kernel is 2D and the total elements is quadratic to the length.

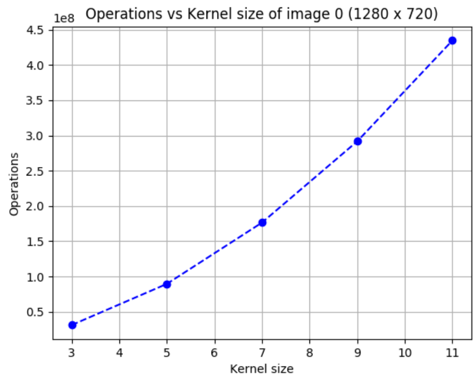


*Figure 17 img0 operations with kernel size*

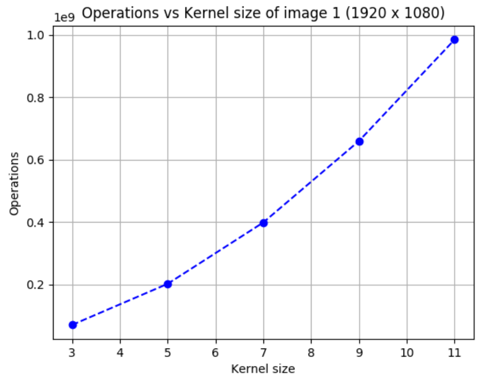


*Figure 18 img1 operations with kernel size*

## Part D

This part is the replication of part B but in C programming language. The running time is still quadratic in terms of the size of output channels but the absolute running time in C compared with Python is much smaller so the C implementation is more efficient.

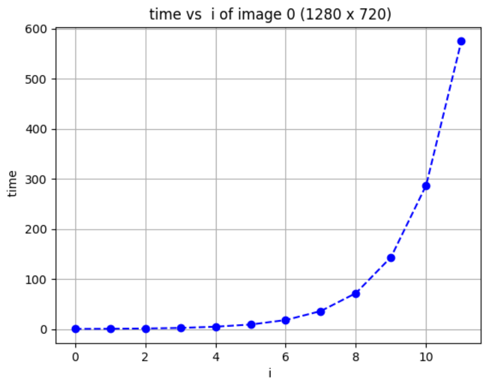


*Figure 19 img0 computation efficiency in c*

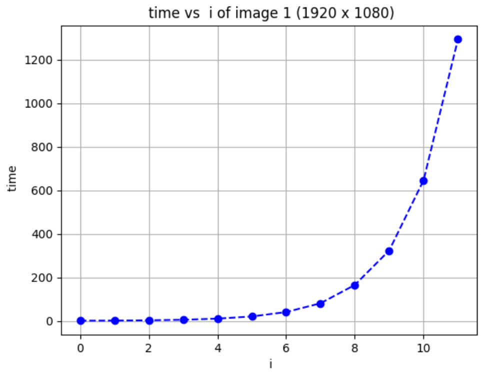


*Figure 20 img1 computation efficiency in c*

# Appendix

1. For replicating results for part ABC:

Run as:
python3 main.py [partA, partB, partC]

2. For replicating results for part D:
   Compile it with:
   ```
   gcc main.c -lm -o main
   ```
   run as
   ./main

3. For generating all the plots:
   python3 plot.py [partA_result.csv, partB_result.csv, partC_result.csv, partC_result.csv]

conv.py

```python
import torch


class Conv2D:
    def __init__(self, in_channel, o_channel, kernel_size, stride, mode='known'):
        self.in_channel = in_channel
        self.o_channel = o_channel
        self.kernel_size = kernel_size
        self.stride = stride
        self.mode = mode
        # predefined kernels
        self.k1 = torch.FloatTensor([[-1, -1, -1], [0, 0, 0], [1, 1, 1]])
        self.k2 = torch.FloatTensor([[-1, 0, 1], [-1, 0, 1], [-1, 0, 1]])

        self.k3 = torch.FloatTensor([[1, 1, 1], [1, 1, 1], [1, 1, 1]])
        self.k4 = torch.FloatTensor([[-1, -1, -1, -1, -1], [-1, -1, -1, -1, -1], [0, 0, 0, 0, 0], [1, 1, 1, 1, 1], [1, 1, 1, 1, 1]])
        self.k5 = torch.FloatTensor([[-1, -1, 0, 1, 1], [-1, -1, 0, 1, 1],[-1, -1, 0, 1, 1],[-1, -1, 0, 1, 1],[-1, -1, 0, 1, 1]])

    def forward(self, img):
        # img is a 3D FloatTensor
        # output is a tuple of (# of operations, 3D FloatTensor impage)
        # the 3D FloatTensor image is a greyscale image with 0s on the third domain.

        channels, height, width = img.size()
        #print("img height:", height, "img width", width, "img stride", self.stride)
```

```python
        out_height = int((( height - self.kernel_size ) / self.stride + 1 ))
        out_width = int((( width - self.kernel_size ) / self.stride + 1 ))
        out_img = torch.zeros(self.o_channel, out_height, out_width)
        #print("out_height:", out_height, "out_width", out_width)
        mul_cnt = 0
        add_cnt = 0
        if self.mode == 'known':
            # task 1
            if self.o_channel == 1:
                kernel = torch.stack([self.k1 for i in range(self.in_channel)])
                kernels = [kernel]
            elif self.o_channel == 2:
                # task 2
                kernel1 = torch.stack([self.k4 for i in range(self.in_channel)])
                kernel2 = torch.stack([self.k5 for i in range(self.in_channel)])
                kernels = [kernel1, kernel2]

            elif self.o_channel == 3:
                # task 3
                kernel1 = torch.stack([self.k1 for i in range(self.in_channel)])
                kernel2 = torch.stack([self.k2 for i in range(self.in_channel)])
                kernel3 = torch.stack([self.k3 for i in range(self.in_channel)])
                kernels = [kernel1, kernel2, kernel3]

            for ind in range(self.o_channel):
                for row in range(out_height):
                    for col in range(out_width):
                        temp_tensor = torch.mul(kernels[ind], img[:, row * self.stride : row * self.stride + self.kernel_size, col * self.stride : col * self.stride + self.kernel_size])
                        out_img[ind, row, col] = temp_tensor.sum()
                        mul_cnt += 1
                        add_cnt += 1
                # change the output image to 3D float tensor
                ops = mul_cnt * self.kernel_size **2 + add_cnt * ( self.kernel_size **2 - 1)
            return ops, out_img
```

```
        elif self.mode == 'rand':
            # for part B
            kernels = []
            for i in range(self.o_channel):
                k = torch.stack([torch.rand(self.kernel_size, self.kernel_size) for i in range(self.in_channel)])
                kernels.append(k)

            for ind in range(self.o_channel):
                for row in range(out_height):
                    for col in range(out_width):
                        temp_tensor = torch.mul(kernels[ind], img[:, row : row + self.kernel_size, col : col + self.kernel_size])
                        out_img[ind, row, col] = temp_tensor.sum()
                        mul_cnt += 1
                        add_cnt += 1
                # change the output image to 3D float tensor
                ops = mul_cnt * self.kernel_size **2 + add_cnt * ( self.kernel_size **2 - 1)
            return ops, out_img

        else:
            print("unknown mode: " + self.mode)
            exit(1)
```

main.py

```
from conv import Conv2D
from PIL import Image
from torchvision.transforms import Compose, ToTensor
from torchvision.utils import save_image
import torch
from sys import argv
from time import time
import csv
import numpy as np


transform = Compose([ToTensor()])
# transform a image to tensor (channel, height, width)
img1 = Image.open("img1.jpg")
img0 = Image.open('img0.jpg')
img_tensors = [transform(img0), transform(img1)]
if len(argv) != 2:
    print("Usage: python main.py part[A, B, C]")
```

```
    exit(1)

tasks = [[3,1,3,1], [3,2,5,1], [3,3,3,2]]
if argv[1] == 'partA':
    # Part A
    for tsk_id in range(len(tasks)):
        task = tasks[tsk_id]
        print('Part A Task ' + str(tsk_id + 1))

        for img_id in range(len(img_tensors)):

            print("Image ", img_id, "size: ", img_tensors[img_id].size())
            conv = Conv2D(task[0],task[1],task[2],task[3],)
            ops, output_img = conv.forward(img_tensors[img_id])
            print('Total operation', ops, ', output tensor size:', output_img.size())
            num_channels = output_img.size()[0]
            if num_channels == 1:
                # task 1
                file_name = "image" + str(img_id) + "/plt_" + str(img_id) + "_partA_task" + str(tsk_id + 1) + "_k1.jpg"
                print("Save to", file_name, "\n")
                save_image(output_img, file_name, normalize=True)

            elif num_channels == 2:
                # task 2
                for i in range(num_channels):
                    file_name = "image" + str(img_id) + "/plt_" + str(img_id) + "_partA_task" + str(tsk_id + 1) + "_k" + str(i + 4) +".jpg"
                    print("Save to", file_name, "\n")
                    save_image(output_img[i, :, :], file_name, normalize=True)
            else:
                # task 3 with 3 o channels
                for i in range(num_channels):
                    file_name = "image" + str(img_id) + "/plt_" + str(img_id) + "_partA_task" + str(tsk_id + 1) + "_k" + str(i + 1) +".jpg"
                    print("Save to", file_name, "\n")
                    save_image(output_img[i, :, :], file_name, normalize=True)
elif argv[1] == 'partB':
    print("Part B")
    task = tasks[0]

```

```
    with open('partB_result.csv', 'w') as out:
        csv_out = csv.writer(out)
        csv_out.writerow(['image', 'i', 'computing time'])
        for img_id in range(len(img_tensors)):
            print("Image " + str(img_id))
            print("Image, i, Time")
            for i in range(11):
                o_c = 2**i
                s = time()
                conv = Conv2D(task[0], o_c, task[2], task[3], 'rand')
                ops, output_img = conv.forward(img_tensors[img_id])
                e = time()
                row = (img_id, i, e-s)
                print(row)
                csv_out.writerow(row)

elif argv[1] == 'partC':
    print("part C")
    task = tasks[1]

    with open('partC_result.csv', 'w') as out:
        csv_out = csv.writer(out)
        csv_out.writerow(['image', 'kernel size', 'operations'])
        for img_id in range(len(img_tensors)):
            print("Image " + str(img_id))
            print("Image, Kernel Size, Operations")
            for ker_size in range(3, 12, 2):
                conv = Conv2D(task[0], task[1], ker_size, task[3], 'rand')
                ops, output_img = conv.forward(img_tensors[img_id])
                row = (img_id, ker_size, ops)
                print(row)
                csv_out.writerow(row)

else:
    print("Wrong argument", argv[1])
    print("Abort!")
```

main.c

```
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#include <math.h>
```

```

void c_conv(int in_channel, int o_channel, int kernel_size, int stride, double*** img, int img_height, int img_width);

void init_plot(double*** img, int height, int width, int in_channel);


int main(int argc, char** argv){
    int in_channel = 3;
    int kernel_size = 3;
    int stride = 1;

// Create Image matrix
    double *** img0 = (double ***) malloc(in_channel * sizeof(double**));
    double *** img1 = (double ***) malloc(in_channel * sizeof(double**));
    init_plot(img0, 720, 1280, 3);
    init_plot(img1, 1080, 1920, 3);
    char* filename = "partD_result.csv";

    FILE *fp;
    fp = fopen(filename, "w+");
    fprintf(fp, "Image, i, time\n");

    for (int i = 0; i < 12; i++){
        int o_c = (int)pow(2, i);
        clock_t s = clock();
        c_conv(in_channel, o_c, kernel_size, stride, img0, 720, 1280);
        clock_t e = clock();
        double elapse =  (double) (e - s) / CLOCKS_PER_SEC;
        fprintf(fp, "0,%d,%f\n", i, elapse);
        printf("img0: 0,%d,%f\n", i, elapse);

        // next for image 1
        s = clock();
        c_conv(in_channel, o_c, kernel_size, stride, img1, 1080, 1920);
        e = clock();
        elapse = (double) (e - s) / CLOCKS_PER_SEC;
        fprintf(fp, "1,%d,%f\n", i, elapse);
        printf("img1: 0,%d,%f\n", i, elapse);
    }
    fclose(fp);
    return 0;
}

```

```
void c_conv(int in_channel, int o_channel, int kernel_size, int stride, double*** img, int img_height, int img_width){
    //printf("morning\n");
    double ***kernel = (double***) malloc(in_channel * sizeof(double**));
    for (int i = 0; i < in_channel; i++)
        kernel[i] = (double**) malloc(kernel_size * sizeof(double*));
    for (int i = 0; i < kernel_size; i++){
        for (int j = 0; j < kernel_size; j ++)
            kernel[i][j] = (double*) malloc(kernel_size* sizeof(double));
    }
    for (int i = 0; i < in_channel; i ++){
        for (int j = 0; j < kernel_size; j++){
            for (int k = 0; k < kernel_size; k++)
                kernel[i][j][k] = (double) (rand() % 10);
        }
    }
    int out_height = (img_height - kernel_size) /stride + 1;
    int out_width = (img_width - kernel_size) / stride + 1;
//    printf("in_channel = %d, o_channel = %d, kernel_size = %d,  stride = %d\n", in_channel, o_channel, kernel_size, stride);
    //printf("out_height = %d, out_width = %d\n", out_height, out_width);
    for (int ker = 0; ker < o_channel; ker ++){
        for (int i = 0; i < in_channel; i++){
            for (int j = 0; j < out_height; j++){
                for (int k = 0; k < out_width; k++){
                    double temp = 0;
                    for (int p = 0; p < in_channel; p++){
                        for (int m = 0; m < kernel_size; m++){
                            for (int n = 0; n < kernel_size; n++){
                                temp += kernel[p][m][n] * img[p][j + m][k + n];
                                //if (img_height == 1080 && j >= out_height - 360)
                                //printf("i = %d, j = %d, k = %d, p = %d, m = %d, n = %d\n", i, j, k, p, m, n);
                            }
                        }
                    }

                }
            }
        }
    }
};
```

```

void init_plot(double ***img, int height, int width, int in_channel){
    for (int i = 0; i < in_channel; i ++)
    {
        img[i] = (double**) malloc(height * sizeof(double*));
    }

    for (int i = 0; i < in_channel; i ++){
        for (int j = 0; j < height; j ++)
            img[i][j] = (double*) malloc(width * sizeof(double));
    }

    for (int i = 0; i < in_channel; i++){
        for (int j = 0; j < height; j ++){
            for (int k = 0; k < width; k++)
                img[i][j][k] = (i + k + j) % 255;
        }

    }

};
```