Pass-by-Function Image Filtering

Kirtan Patel & Tim Swyzen

May 13, 2019

Why and How

Passing a function as a value is a newer development that has a lot of uses, especially in making certain types of programs more organized and structured. We see it as a core feature of functional programming, where functions are just lists that can be passed back and forth with ease. Python 3 has this feature built-in, and we used it in a very clear way - image filtering.

Each filter is a function defined in filters.py, which takes in a numpy image array and modifies it in some way. The functions are put into a dictionary which corresponds to the selection made by the user. The uploaded image is then put through the selected function by passing that function as a value, and it returns the filtered image.

Since we tried to emulate functional-style programming in Python, there are some similarities and differences. On a surface level, the image functions don't change any greater state - they take an image as input and output another image - nothing else. However, in order to get UI working properly, extra-functional variables were used, which would absolutely not be possible in a pure functional language. Additionally, loops were used to modify each pixel, although it could easily have been modified to use recursive calls instead.

In addition to passing functions as values, we used a few more advanced Python features, including NumPy, OpenCV2, skimage, scipy, and matplotlib. Many of these were used in creating specific filters for images. Additionally, list comprehension, which was discussed in the Python section of the class, was used in the parallel processing experiment (although it did not make it to the final version).

One big change in design we should do is abstract out the image functions. They all have the same structure, and rewriting them is tedious. Making a system with easier function creation would not be too difficult with more time.

Problems

Runtime

Compared to applications like Gimp and Paint.net, ours runs extremely slow. For example, a black and white filter on an image roughly the size of the chimpanzee image we tested with took roughly half a second on Paint.net, while it took our application 1.6 seconds more or less. There is a lot of optimization to be done, although research suggested that parallel processing is the only real optimization available for image filtering.

UI

The main decision to make was whether we wanted our application to run in browser using Flask or to run as its own standalone application using tkinter. Long processing times on a web app are an issue due to high amounts of possible interference, as well as requiring a server that can handle the heavy lifting.

A standalone tkinter app had issues of portability and relatively ugly UI. By running our app on a browser, we are able to use the application on any platform. In addition, we don't have to worry about the user having the right libraries and using the right version of Python. However, as we stated before, we are giving up performance for compatibility and nicer UI.

Website

In order to run our Python application we needed to find a framework that will allow us to create a website that can run our code, so I picked Flask. Flask is a popular Python micro web framework and we just need to be able to process images, so the framework is very useful. In order for the Python application to run and communicate with a server we need to pick a Web Server Gateway Interface, so we picked Gunicorn as it is fast and can handle multiple users on one website. After we created the website and it was time to process images. The image processing was very slow compared to our stand alone python application. If a person chooses to upload a large image to the website, it will take a large amount of time to process the image. We couldn't find the reason why the website was slower than the standalone app, but we think it has to do with how Flask is not good at handling resource intensive Python applications, as the framework is not made to do that. In order to get around this problem, we allowed the user the ability to compress their image, which reduces it to 50% of the original size. This greatly speeds up the process.

Hosting

To make sure that our application is reachable, we needed to find a platform that will allow us to host our python application. At first, we hosted our application on Heroku as it allowed us to host it for free. However, some issues started to appear when we were testing out the application. When a user tried to access the app, it would take about 5-10 seconds for the app to load up. Heroku would hibernate the app after a specific time of inactivity, which resulted in timeouts for larger images, rather than long processing time. Due to these problems, we decided to look for a different hosting platform. Besides, we wanted a better performing server as it might help increase the performance of our application, so we looked at AWS and Microsoft Azure. We decided to go with Azure because Microsoft gives $100 in credit if you are a student, while AWS doesn't allow python applications to be created if you have a student account. Deploying our app was hassle free as we just had to clone our project from Github and let Azure's built-in command setup it up. In addition, Azure automatically configures Gunicorn, which was great as we didn't need to mess around with the configuration. Azure was a great choice as we hadn't had any problems with our website that was hosting related.

Parallel Processing

We attempted to implement parallel processing in the forms of multithreading and multiprocessing. However, they do not seem to be a good solution for this sort of project. All of these were tested with the simple black and white filter on three images: a 228x168 image of Blue from Blue's Clues; a 702x378 image of chimpanzees; and a 1500x1500 image of Ryu from Street Fighter (less tested).

0.1 Base case (no parallel processing)

By default, we loop through the 3-dimensional NumPy matrix and modify each pixel, as follows.

Listing 1: A basic, pixel-by-pixel black and white filter

```
def bnw( img ):
    for i in range( len( img ) ):
        for j in range( len( img[i] ) ):
            ave = img[i][j][0] * 0.299 + img[i][j][1]
                * 0.587 + img[i][j][2] * 0.114
            img[i][j][0], img[i][j][1], img[i][j][2] =
                ave, ave, ave
    return img
```

Table 1: Runtimes of default black-and-white filter given various sized images.

	Blue (228x168px)	Chimps (702x378px)	Ryu (1500x1500px)
Trial 1	0.23783s	1.68409s	14.40296s
Trial 2	0.24235s	1.69750s	-
Trial 3	0.24528s	1.68266s	-

0.2 Multithreading

To implement multithreading, we created a thread for each row of pixels, which would modify the original image.

Listing 2: Multithreaded black-and-white filter

```
def bnwMT( img ):
    start = time.time()
    def pixBnw( rownum ):
        for i in range( rownum ):
            ave = img[rownum][i][0] * 0.299 +
                img[rownum][i][1] * 0.587 +
                img[rownum][i][2] * 0.114
            img[rownum][i][0], img[rownum][i][1],
                img[rownum][i][2] = ave, ave, ave

    ts = []
    for i in range( len( img ) ):
        ts.append(threading.Thread(target=pixBnw,
            args=(i,)))
    [t.start() for t in ts]
    [t.join() for t in ts]

    print( time.time() - start )
    return img
```

Table 2: Runtimes of multithreaded black-and-white filter given various sized images.

	Blue (228x168px)	Chimps (702x378px)	Ryu (1500x1500px)
Trial 1	0.26818s	1.72368s	14.212799s
Trial 2	0.27477s	1.71300s	-
Trial 3	0.27708s	1.71720s	-

0.3 Multiprocessing

To implement multiprocessing, we split the image vertically into three and created a process to handle each third.

Listing 3: Multithreaded black-and-white filter

```
def pixBnw( imgsec, procnum, return_dict ):
        for i in range( len(imgsec) ):
            for j in range( imgsec[i] ):
                ave = imgsec[i][j][0] * 0.299 +
                        imgsec[i][j][1] * 0.587 +
                        imgsec[i][j][2] * 0.114
                imgsec[i][j][0], imgsec[i][j][1],
                        imgsec[i][j][2] = ave, ave, ave
        return_dict[procnum] = imgsec

def bnwMP( img ):
    start = time.time()
    manager = multiprocessing.Manager()
    return_dict = manager.dict()

    procSize = math.ceil( len(img) / 3 )
    procs = []
    procs.append( multiprocessing.Process(target = pixBnw,
        args = (img[0:procSize],0,return_dict) ) )
    procs.append( multiprocessing.Process(target = pixBnw,
        args = (img[procSize+1:2*procSize],1,return_dict)))
    procs.append( multiprocessing.Process(target = pixBnw,
        args = (img[2*procSize+1:],2,return_dict) ) )

    [p.start() for p in procs]
    [p.join() for p in procs]
    img = return_dict[0] + return_dict[1] + return_dict[2]

    print( time.time() - start )
    return img
```

Table 3: Runtimes of multithreaded black-and-white filter given various sized images.

	Blue (228x168px)	Chimps (702x378px)	Ryu (1500x1500px)
Trial 1	5.69958s	6.450966s	-
Trial 2	-	-	-
Trial 3	-	-	-

0.4 Conclusions

Perhaps the implementation was poor, but it seems that the startup times for Python make it a poor candidate as a tool for image processing on such a small scale. Multiprocessing had very heavy investment to start the processes, but may have scaled better. Multithreading had the same issue on a smaller level, and may have broken even with larger images, given more testing.