ECE3270 Digital Computer Design: Lab 6

OPENCL IMAGE DILATION DESIGN

April 27, 2019

Aayahna Herbert
Clemson University
Department of Electrical and Computer Engineering
aayahnh@g.clemson.edu

Abstract

The goal of this lab was to further our understanding of FPGA computing and to introduce OpenCL. The lab required C code and an OpenCL kernel to be written. This design lab was split into two parts: (1) threshold and dilating a PPM image in C code and timing the dilation algorithm and (2) modifying the C code to create an OpenCL kernel and testing it on a DE1-SoC board. In this lab, part 1 of the lab was successfully written and tested using five PPM images with different sizes each; part 2 was incomplete due to a segmentation fault in the kernel that could not be debugged in the allotted time for this lab assignment .

1 Introduction

According to Netpbm [1], a portable pixel map (PPM) image is a lowest common denominator color image file format.A PPM file consists of a "magic number" that identifies the file type, white space, a width and height that are formatted as ASCII characters in decimal, sometimes comments, and binary data holding the color of each pixel in the form of red-green-blue (rgb). For this lab, a PPM image was read in, converted to black and white if any of its rgb values exceeded 185 (threshold), and then dilated and written out to another PPM image. The next sections will go into details about how each part of the lab was constructed and the reason it was done the way it was.

2 C Code

Because the main focus of this lab was creating the dilation algorithm, an external library was used to create functions that can read and write PPM images with a magic number of P6 - the external library came from a past assignment in CPSC1010 course in Spring 2015 that worked with reading and writing PPM images. To threshold the image, a nested for-loop was used to run through the rows and columns of the image, check to see if any of the rgb values were above 185. If they were above 185 all of the pixels were given the value 255 to make the pixel white, signifying it was on; on the other hand, if they were not, all of the values were set to 0 to make the pixel black, signifying it was off. After this was tested to make sure the PPM image was successfully edited to black and white, the dilation algorithm was made. The dilation algorithm goes through every row and column searching for pixels that are on. If the filter is on, then a nested for loop goes through and makes the pixels surrounding the white pixel white as well, making it look like a 5x5 white square was centered on the white pixel detected by the algorithm. After it was shown that the algorithm worked, a clock was used to stamp the start and end time for the algorithm and the run time was calculated.

2.1 Testing and Simulation

When it came to testing the C code with PPM images, five PPM images all with different dimensions were used to see how dilation was done and to see the runtime for the code; Figure 5.1 shows the runtime of each file tested while the rest of the figures show the original pictures and their dilated images as a result of the C code's dilation algorithm.

2.2 OpenCL Kernel

2.2.1 Design Method

In order to save time and have an outline for the second part of this labe, the OpenCL folder from Lab 2 was used to make the kernel (.cl) and main (.cpp) files. For the OpenCL kernel, the dilation algorithm was slightly modified by discarding the first two nested

for loops in order to make the function parallel; other than that, it was kept the same and copy and pasted over. Once that was completed, the main.cpp file was modified - along with Makefiles and run shell files - to complement the kernel dilation algorithm and calculated the dilation algorithm's runtime for the kernel and CPU.

2.2.2 Testing and Simulation

To save time working with the long compiling time of OpenCL, emulation was used to test if the algorithm was completed and the runtimes were calculated and displayed correctly. Unfortunately, a segmentation fault was constantly detected within the kernel section; because there were no faults in the C code, it was difficult finding a way to modify the kernel code to debug and get rid of the fault found. It is suspected that the cause of the fault might have to do with reading in the data to the pixel's rgb values and possibly not having enough memory while doing this action.

3 Results

While the C code was fairly easy to complete, the kernel code was much more difficult in comparison and was unable to be finished in time. A segmentation fault was found within the kernel code (.cl file) and it was very hard trying to determine what code modifications could be used to fix this error. Because it could not be found, it could never be implemented on the board and the speedup time could not be calculated since the kernel time could not be computed due to the error. Nonetheless, if were to have worked, it would have been expected that the kernel runtime would be significantly faster than the C runtime because of parallelism in the kernel, causing a lack of dependency compared to the C code that was dependent on different conditions and variables from for-loops and conditional statements.

4 Conclusion

This lab has given me the chance to test and apply my knowledge about OpenCL and kernel coding that was learned from class lectures while also getting acclimated to PPM images and how to handle them. Although I was unable to fully complete it, by doing this lab, I have a clearer understanding on the intricacies of kernels and parallelism in OpenCL. If I could go back and change how things were done throughout the lab, I would spend more time finding the cause of the segmentation fault within the kernel code so I could then do an in-depth analysis of the speedup on the hardware.

References

[1] S. Forge. ppm. Website, USA, 2016.

5 Appendix

```
aayahnh@ullab02:~/lab6$ ./a.out tillman.ppm out1.ppm


C Time spent = 0.005458 s
aayahnh@ullab02:~/lab6$ ./a.out program.ppm out2.ppm


C Time spent = 0.038937 s
aayahnh@ullab02:~/lab6$ ./a.out sign_1.ppm out3.ppm


C Time spent = 0.002668 s
aayahnh@ullab02:~/lab6$ ./a.out west_1.ppm out4.ppm


C Time spent = 0.005037 s
aayahnh@ullab02:~/lab6$ ./a.out paw.ppm out5.ppm


C Time spent = 0.086516 s
```

Figure 5.1: Runtime for Each PPM Image in C

Figure 5.2: Original PPM Image of Tillman

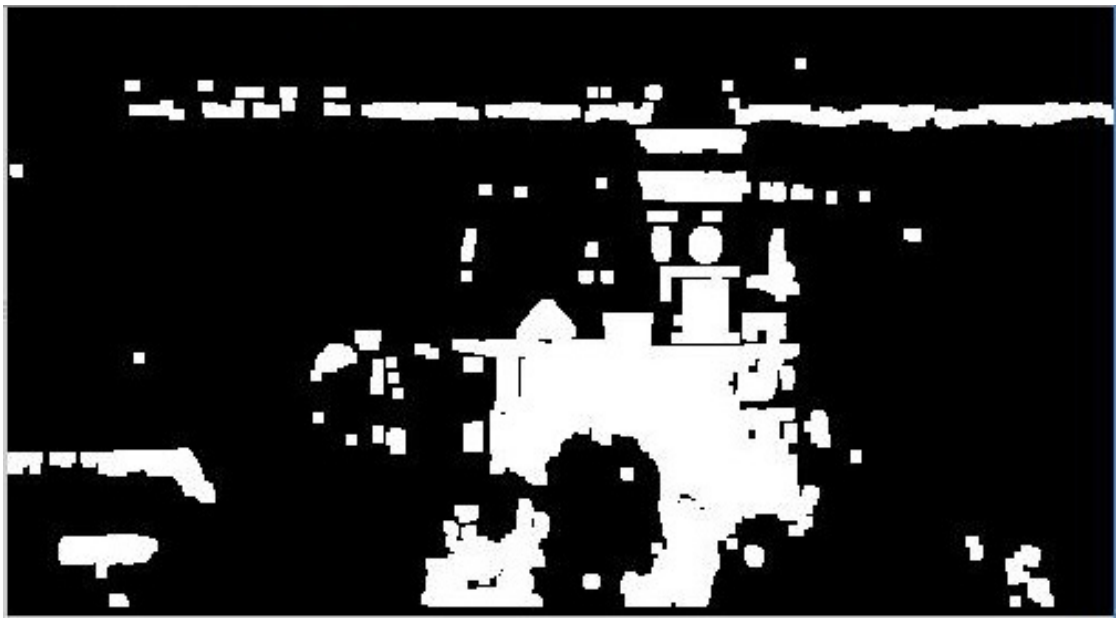

Figure 5.3: Dilated PPM Image of Tillman

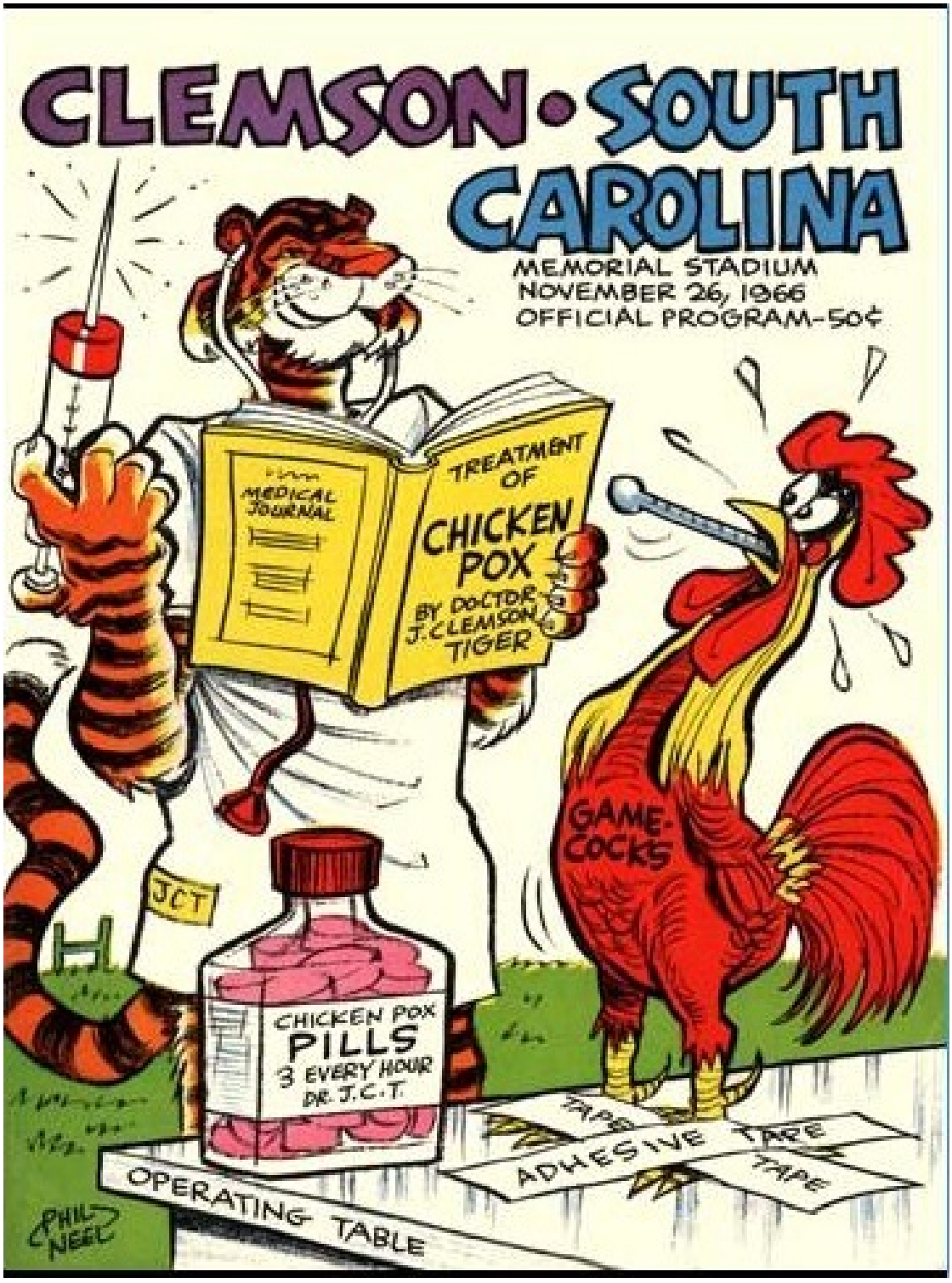

Figure 5.4: Original PPM Image of Program

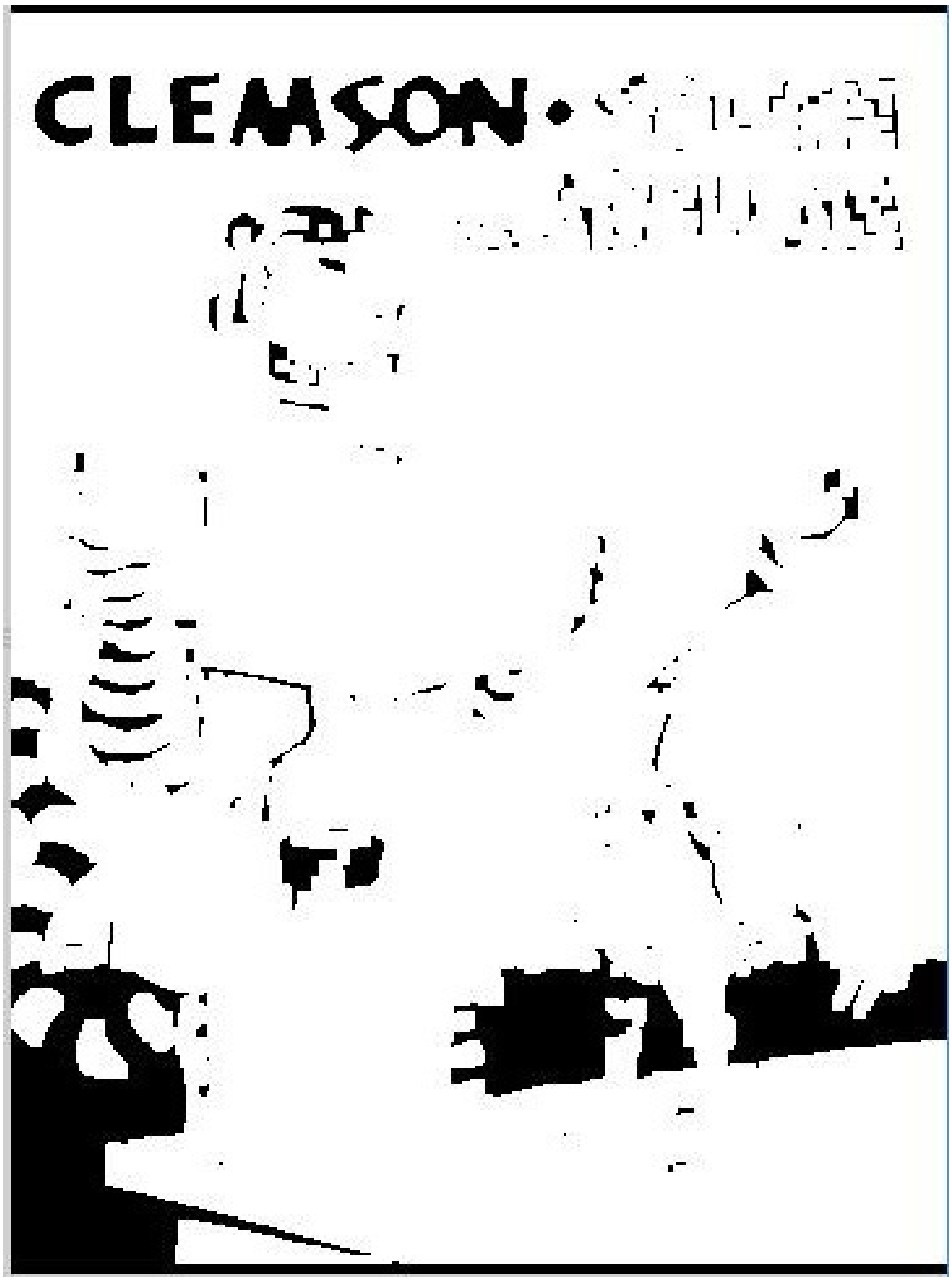

Figure 5.5: Dilated PPM Image of Program

Figure 5.6: Original PPM Image of Sign1

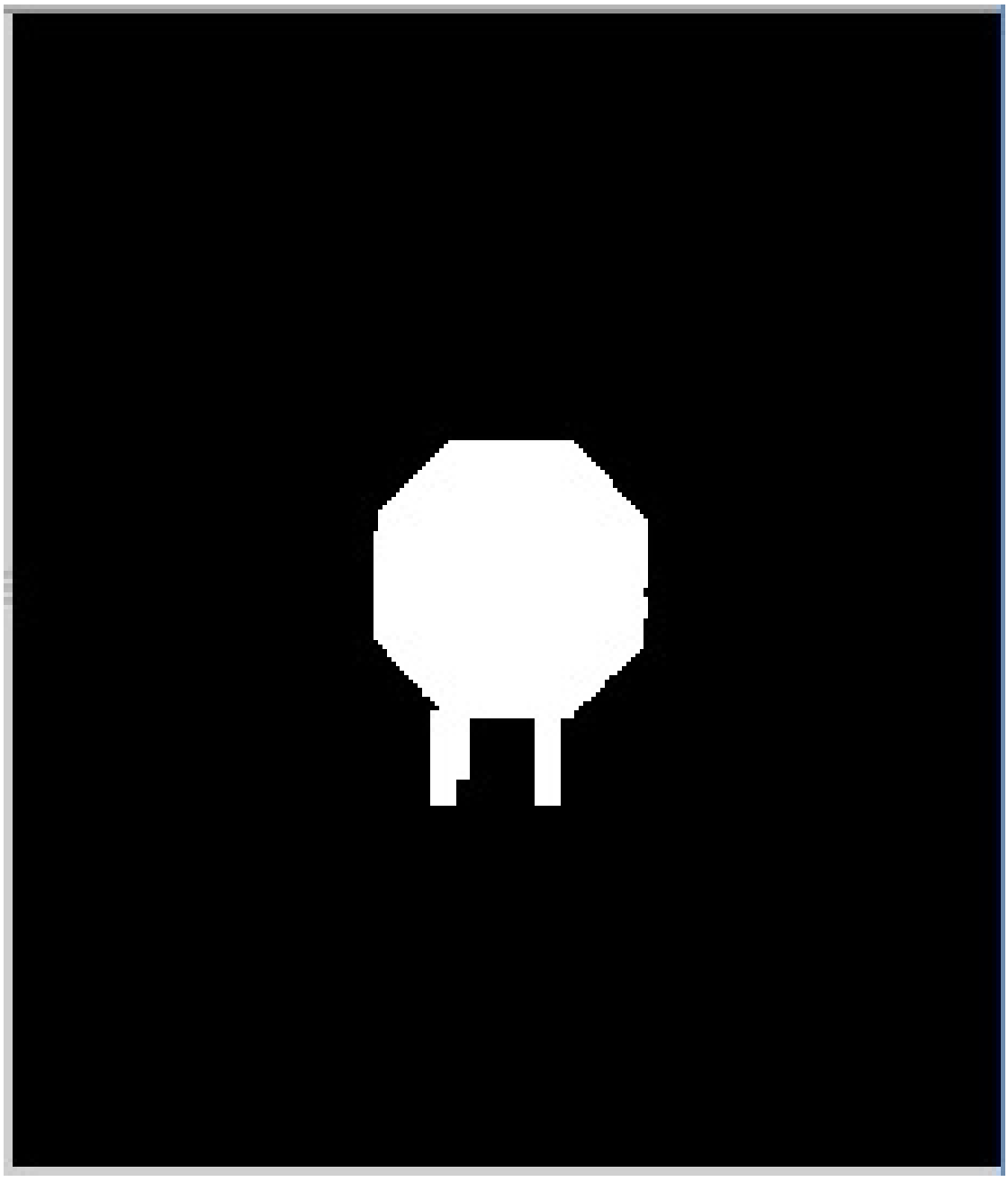

Figure 5.7: Dilated PPM Image of Sign1

Figure 5.8: Normal PPM Image of West

Figure 5.9: Dilated PPM Image of West

Figure 5.10: Normal PPM Image of Paw

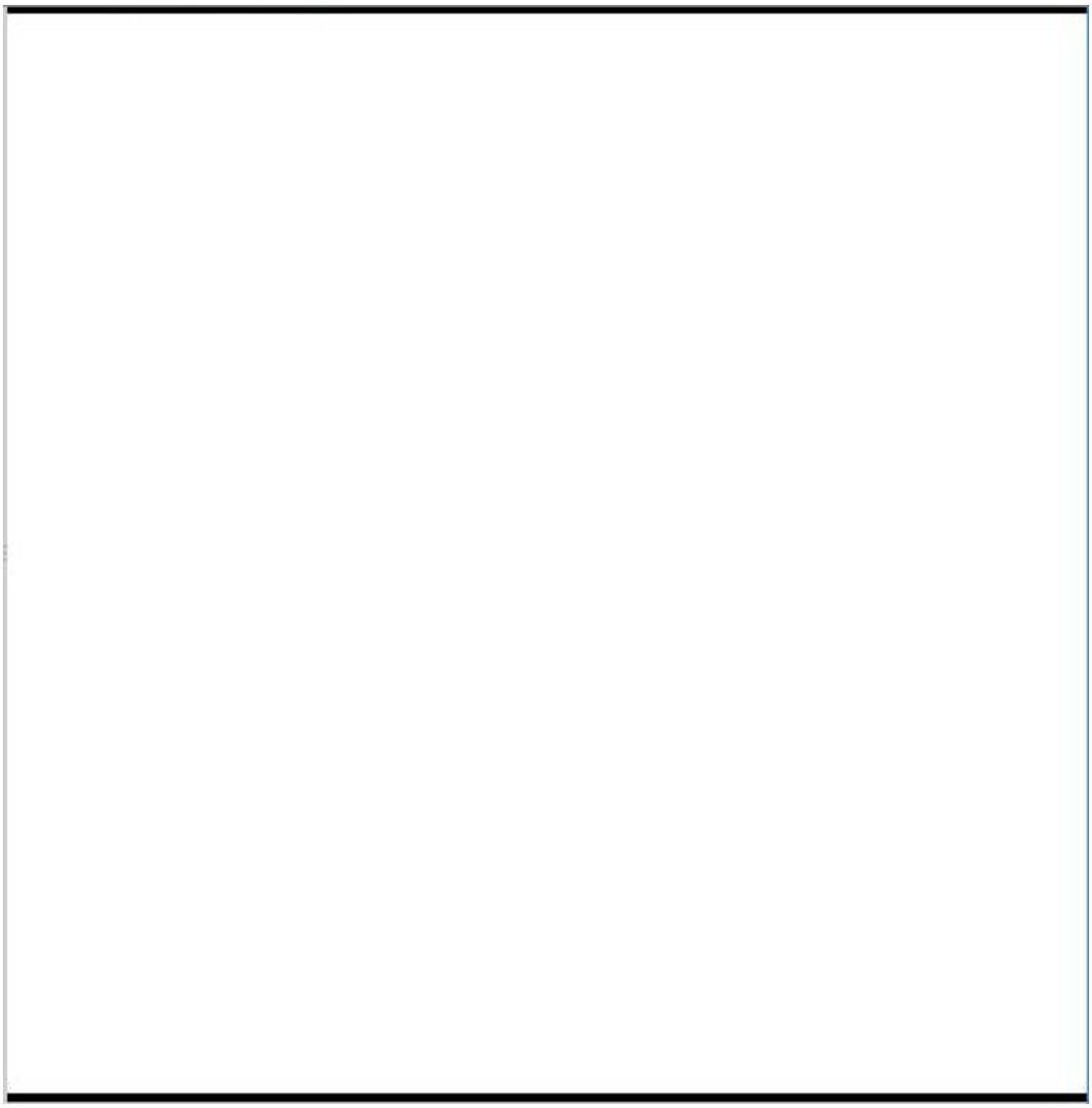

Figure 5.11: Dilated PPM Image of Paw