Nathan Bain
neb553
December 12, 2015
Sreepathi Pai

Haralick Edge Detection

The Haralick edge detection algorithm is a method for detecting edges in images. It is based on the second derivative of the image, which allows it to determine more of the characteristics of the edge and to reduce unnecessary noise in the image. The algorithm is effective at finding true edges in images, as well as for highlighting clear edges better.

The algorithm takes an image, approximates it through a bi-cubic polynomial fitting, and then creates an expression to find the zeroes of the second derivative of the polynomial. A pixel is a zero if any two opposite pixels in the surrounding neighborhood have differing signs, and their difference is greater than an input threshold. Initially, the algorithm runs every value at in the surrounding five by five matrix around x and y through the bi-cubic polynomial

$$f(x,y) = k_1 + k_2 x + k_3 y + k_4 x^2 + k_5 xy + k_6 y^2 + k_7 x^3 + k_8 x^2 y + k_9 xy^2 + k_{10} y^3$$

If we set

$$\cos(\theta) = -\frac{k_2}{\sqrt{k_1^2 + k_2^2}}, \sin(\theta) = -\frac{k_3}{\sqrt{k_1^2 + k_2^2}}$$

$$x = \rho\cos(\theta),\ y = \rho\sin(\theta)$$

Then this formula can be changed to

$$f_\theta(\rho) = C_0 + C_1\rho + C_2\rho^2 + C_3\rho^3$$
$$C_0 = k_1$$
$$C_1 = k_2\cos(\theta) + k_3\sin(\theta)$$
$$C_2 = k_4\cos(\theta)^2 + k_5\cos(\theta)\sin(\theta) + k_6\sin(\theta)^2$$
$$C_3 = k_7\cos(\theta)^3 + k_8\cos(\theta)^2\sin(\theta) + k_9\cos(\theta)\sin(\theta)^2 + k_{10}\sin(\theta)^3$$
$$f''_\theta(\rho) = 2C_2 + 6C_3\rho$$
$$f'''_\theta(\rho) = 6C_3$$

Once this is calculated, a point can be determined to lie on an edge second derivative is zero within a threshold, and the third derivative is negative. The algorithm will detect an edge if and only if

$$\left|\frac{C_2}{C_3}\right| < \rho_0,\ C_3 < 0$$

The higher the threshold ρ_0, the more edges are detected. The best results are obtained with a threshold less than 1.5, and are usually better around .5.

The baseline used for this project was obtained from a github repository to accompany paper on edge detections. The paper and source code was written by Haldo Spontón and Juan Cardelino at the University of the Republic in Uruguay. The source was written only to demonstrate the algorithms, and includes very few optimizations. All of the code is written in C, and works with images in PNG, JPG, and TIFF. The program first converts the image to an array, and then converts it to grayscale, as only the intensity of the pixel is relevant. It takes four inputs, the input image, the threshold for an edge, whether or not to add blank spaces around the array to determine edges, and the output image. The baseline code takes each pixel, computes the local neighborhood, and multiplies each pixel in the neighborhood by ten different values in ten constant masks. It then computes the sum of the neighborhoods to create $k_1 \ldots k_{10}$, and then computes C_2 and C_3 to determine if it is an edge.

The bottleneck in the code was in the edge detection algorithm ran on each pixel, as this required 250 multiplications for each pixel. The optimization in this report will focus on just that loop. Since the program is just shifting the neighborhood over each time, performance might be increased by not computing a new neighborhood each time and instead shifting the old one. The inner most loop computer multiplies a point in the surrounding matrix by a mask array, and adds that to an accumulator. This can be improved by using vectors, either by using multiple points in the surrounding matrix and computing a horizontal add at the end, or my using multiple points in the image, determining the edge detection four pixels at a time. Multiple threads can be implemented so that the work can be split up to different cores on the machine. Since the focus of this project is not the on loading the images, only the performance of the middle loop will be measured. Vectorization should be able to improve the performance by at least a factor of two or three, and by splitting the image up into threads, I hope to increase that by at least a factor of six. After all optimizations, the program should be at least fifteen times faster.

The program will be run on the Stampede computers, which include 16 cores and run Centos. The performance will be measured using monotomic time taken before and after the main loop runs. The input image is a 2448 by 3264 pixel 7.9 Mb image of lizard, which is the standard Iphone 5 photo size. The program will be run with a threshold of 1, which determines realistic results, and bordering will be set, so that a border is set around the array, which allows for edges up to the border of the image to be found without determining the entire border of the image to be an edge. Once the program is run, it will be verified against the baseline output by checking the number of edges counted and using the *diff* command to check against the output image. When ran with the inputs specified above, the baseline loop takes an average of 8.3 seconds.

Initially, the algorithm could be improved upon to increase performance. Since sine and cosine share the same denominator and the degree of the each term in the numerator and denominator are the same as the other terms in their respective polynomial, the denominator can be removed in the sine and cosine equations and the end result can be changed to

$$\left|\sqrt{k_1^{\,2} + k_2^{\,2}}\,\frac{C_2}{C_3}\right| < \rho_0,\ C_3 < 0$$

This removed two divisions per pixel and adds one multiplication per pixel. This increased decreased the runtime to 8.1s, almost two percent better than the baseline. This may seem like a small amount that could be due to many factors, however, since the 99% confidence intervals are nowhere near each other, the results are significant, and the increase in performance can be reasonably attributed to this algorithm change.

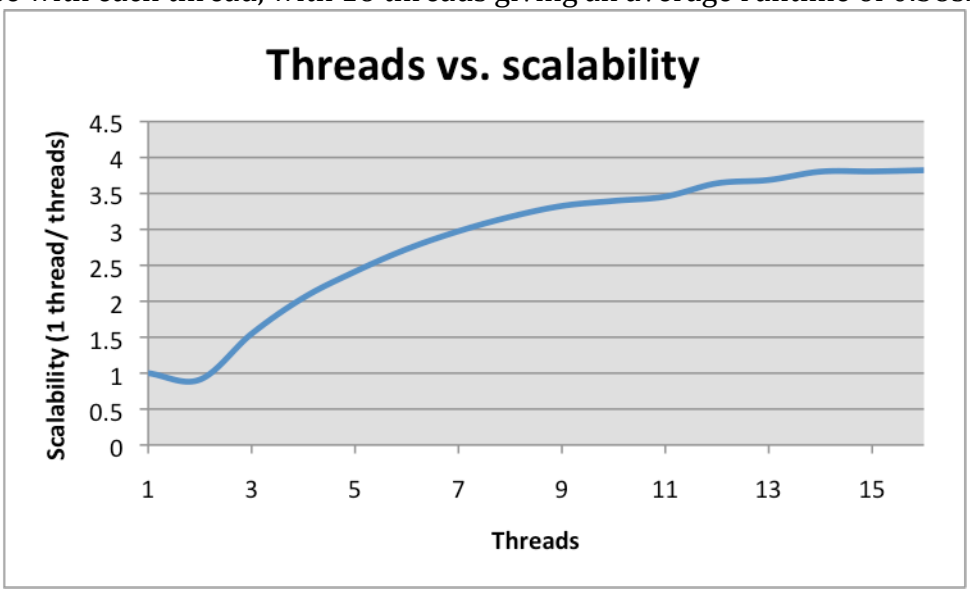
	Start Interval	Average	End Interval
Baseline	8.261s	8.287s	8.313s
New Algorithm	8.100	8.133s	8.165

The next optimization that was added was vectorization. At first, I vectorized only the innermost loop using vector intrinsics with sse3 and avx vectors on the machine, multiplying four neighbors at a time, and summing with a horizontal add. Since there were still more calculations in the rest of the loop, and since this added two horizontal adds for each of the ten masks, performance actually suffered by a small amount. program in the vectors my self using vector intrinsics. I then changed the code so that it computed all of the same calculations for four different pixels at a time. This resulted in average runtime of 4.7, about 1.7 times better than the baseline. However, by using the previous code and setting the compiler to use autovectorization with the flags“-O3 -ftree-vectorize -ftree-vectorizer-verbose=1”, the program resulted in a runtime of 2.6, almost half the runtime of my vectorization, and 3.2 times better than the basecode. Autovectorization was used for the rest of the project.

In the original algorithm, k_1 was necessary to compute the second derivative, but not necessary to compute final result. However, the original code still computed that value by multiplying the 25 pixels in the neighborhood by another mask. Since the value was never used again, it was removed. The original loop multiplying the

masks by the neighborhood computed a neighborhood and then multiplied each point by a corresponding point in a mask, accumulated it, set the *k* value, and continued for another mask. By changing the program to load a point in a neighborhood to a variable and multiply it by nine *k* values, it loads the neighborhood eight fewer times. This had an average runtime of 1.4s, and resulted in a speedup of 5.9 over the base code and 1.9 over the previous improvements.

Finally, threads were implemented using pthreads to spread out the work of the loop. The main thread set up the values for the other threads to use, created them, ran its own loop on the remaining pixels, and joined all of the threads together. The threads all wrote to the same array representing the image since no two threads would be writing to the same point. Each thread returned the count of edges as a return value, which was summed up when joined by the main thread. By checking what core each thread was using, I found that each thread did not have its own core, and instead they only used two to six cores each run. By setting the thread to a core with pthread_attr_setaffinity_np, the program gained a slight performance improvement, though it still leveled out around a factor of four, though it did increase with each thread, with 16 threads giving an average runtime of 0.38s.



The final improvements are

	Runtime (s)	Current/previous	Current/baseline
Baseline	8.287		
Algorithm Change	8.133	1.019	1.019
Vectorization	4.751	1.712	1.744
Autovectorization	2.605	1.824	3.182
Optimizations	1.407	1.852	5.891
Threads	.379	3.708	21.845

Sources

“A Review of Classic Edge Detectors”; Haldo Spont´on, Juan Cardelino; Instituto de Ingenier´ıa El´ectrica, Universidad de la Repu´blica, Uruguay

github.com/haldos/edges