Parallelize Painterly Rendering with Multiple Stroke Size

Shih-Ting(Emily) Huang, Yu-Lin(Connie) Chou

Abstract

The feature of image processing makes it suitable to parallelize. In this project, we will implement the algorithm that is proposed in the paper "Painterly Rendering with Curved Brush Strokes of Multiple Sizes" and accelerate it with OpenMP in a parallel fashion. Finally, I will compare it with the traditional sequential CPU-based version and know whether this part should be parallelized from the experiment.

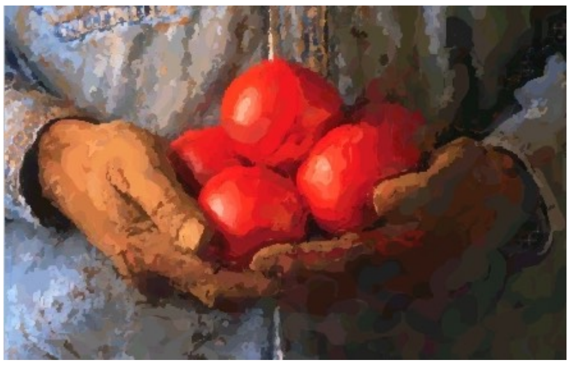


Fig. 1 The result of Painterly Rendering with Curved Brush Strokes of Multiple Sizes

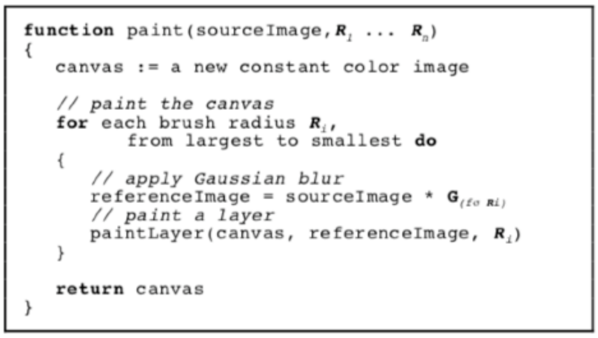
1 Introduction

Painterly rendering is a technique that automatically creates images that have handed-painted appearance from a photograph. Most current computer painterly rendering algorithm use very simple brush strokes that are all of equal size and shape; thus, the resulting images tends to be mechanical in comparison to hand-made work. Therefore, Hertzmann proposed a new algorithm in Siggraph 98 paper "Painterly Rendering With Curved Brush Strokes of Multiple Sizes". In this paper, they present techniques for painting an image with multiple brush sized, and for painting with long, curved brush strokes in order to get a more natural resulting image, and he also proposed a method that the user can define different parameters to get different painting styles such as Impressionist, Expressionist, Colorist Wash, Pointillist. We find that in their algorithm, there are many techniques that can be parallelized such as Gaussian Blur, Sobel Operator, adding stroke points part, and drawing part. Thus, we think this can be a very suitable and interesting application to parallelize.

2 Literature Survey

We can only find one parallelized version of this implementation using GPU in 2004: GPU-Accelerated Painterly Rendering, but we find that it is implemented many years ago, and he used many techniques that is deprecated. For example, he used Cg programming language which is now deprecated by Nvidia since 2012, and he used Qt 3.1.2, which is now unsupported by many operating systems. We tried to run his code using new Qt version, but there are many libraries and usages are deprecated in the new version. Furthermore, we believe that this implementation is more suitable to do in multicore CPU rather than GPU, because the communication overhead of CPU-to-GPU is higher than CPU-to-CPU. Therefore, we think it is necessary to propose our new implementation using multicore CPU using OpenMP rather than GPU.

3 Proposed Idea



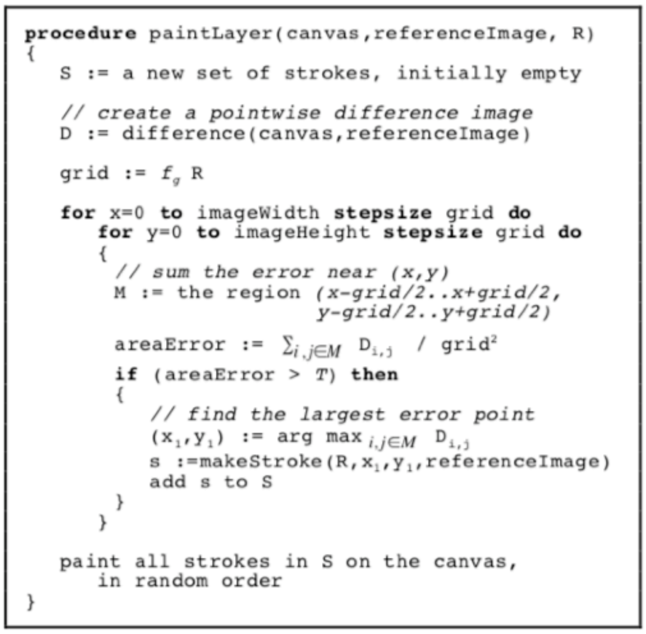
```
function paint(sourceImage,R_1 ... R_n)
{
   canvas := a new constant color image

   // paint the canvas
   for each brush radius R_i,
         from largest to smallest do
   {
      // apply Gaussian blur
      referenceImage = sourceImage * G_(fσ Ri)
      // paint a layer
      paintLayer(canvas, referenceImage, R_i)
   }

   return canvas
}
```

Fig. 2 Paint Algorithm

In this chapter, I will explain what sections can be parallelized and why. From the algorithm in Fig. 2 that is proposed in the paper, we can find that we have to do



```
procedure paintLayer(canvas,referenceImage, R)
{
   S := a new set of strokes, initially empty

   // create a pointwise difference image
   D := difference(canvas,referenceImage)

   grid := $f_g$ R

   for x=0 to imageWidth stepsize grid do
      for y=0 to imageHeight stepsize grid do
      {
        // sum the error near (x,y)
        M := the region (x-grid/2..x+grid/2,
                         y-grid/2..y+grid/2)

        areaError := $\sum_{i,j \in M} D_{i,j}$ / $grid^2$
        if (areaError > T) then
        {
           // find the largest error point
           ($x_1$,$y_1$) := arg max$_{i,j \in M}$ $D_{i,j}$
           s :=makeStroke(R,$x_1$,$y_1$,referenceImage)
           add s to S
        }
      }

   paint all strokes in S on the canvas,
      in random order
}
```

Fig. 3 paintLayer Algorithm

Gaussian blur for each brush size with different blur factor to get the reference image for each layer. Gaussian blur is the process to do convolution with special gaussian kernel(mask), and the process we have to do repetitive work on each pixel and each pixel is independent. Thus, it is good chance to do parallelization.

The main purpose of the algorithm in Fig. 3 that is proposed in the paper is to decide which point should generate a stroke and we find that the two for loops to some the error also can be parallelized, because there is no dependency on each pixel. In addition, the section that paint all strokes in S on the canvas can also be parallelized, because we don't have to draw in certain order, and each stroke is independent from other strokes.

Moreover, in this paper, they used Sobel operator to get the gradient at each pixel and according to the gradient, we can decide the stroke curve's direction. Sobel operator also can be parallelized, because it is also independent from each pixel.

For implementation, we use OpenCV to deal with image read, write, draw and pixel manipulation, and use OpenMP to do shared memory multiprocessing in C++.

4 Experimental Setup

In this chapter, I will clearly illustrate my experimental configurations. First of all, I want to clarify that in these experiments, besides showing the speedup over traditional CPU sequential version, we'd like to do some experiments to prove that not all the tasks are deserved to do parallelization. We do all the experiments on CIMS crunchy6, which has four AMD Opteron 6272 (2.1 GHz) (64 cores) and has 256 GB memory, and the operating system is CentOS 7. For time measurement, we use both std::chrono::high_resolution_clock and time command. We use chromo::high_resolution_clock for measuring the time of each task, and use time command for measuring the total time taken of this application.

In these experiments, we use cap-18 images which is used in the original paper, and we generate the images of different sizes that are put in "Test Image" folder. In order to show the importance of the problem size, we do the experiment with 4 different sizes: 512x270 pixels, 1024x540 pixels, 2048x1080 pixels, and 2250x1510 pixels. And we divide our implementation into four tasks: Gaussian Blur, Sobel operator, adding stroke points, and draw. According to the above configuration, we will do three experiments: (1) Task benchmark with different number of threads using Impressionist style parameters (2) speedup with different number of threads on different image size (3) Task benchmark with different number of threads using Pointillist style parameters.

5 Experiments & Discussion

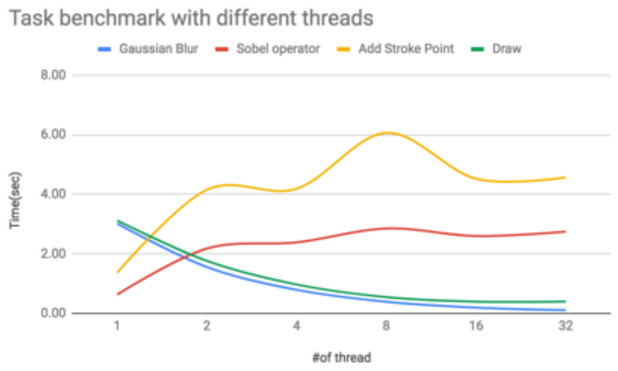


Fig 4 Task benchmark with different number of threads(Impressionist)

In this chapter, we will show our experiment results and explain it in detail. For the experiment (1), we parallelize all possible parts (gaussian blur, sobel operator, adding stroke points, and draw), and record the time of each task with different number of threads(# of thread = 1, 2, 4, 8, 16, 32) and use cap18-2250x1510 as input image and set the impressionist parameters. Then, we can get the figure 4. From this figure, we can find that when the number of thread increases, the time taken on the gaussian blur and the draw tasks reduce, but the sobel operator and the adding stroke point task doesn't. From the experiment, we can also draw the figure 5 with number of thread equals to one and from the figure, we can clearly see that most of the time is taken on Gaussian blur task and draw task. These two figures show that not all tasks are deserve to parallelize. If we parallelize those tasks that are too simple and do not take too much time in the sequential version, we can not get speedup and can even get slowdown over the single thread version.

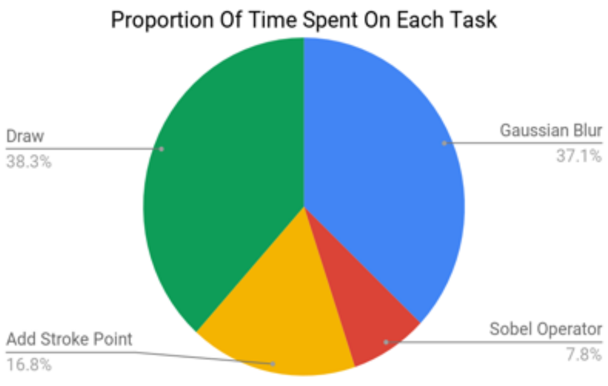


Fig 5 Proportion of time spent on each task with number of thread=1

According to experiment (1), in order to get a better performance, we revise our code to not parallelize sobel operator task and adding stroke points. Then, we do the experiment (2) with the revised version code, and record the time of each task with different number of threads(# of thread = 1, 2, 4, 8, 16, 32) and use cap18-512x270, cap18-1024x540, cap18-2048x1080 as input images and set the impressionist parameters; then, we can get the table 1 shown in seconds. And from the table 1, we can draw the figure 6 showing the speedup over single thread.

Table 1 Time taken with different image sizes and different

Number of threads	Size: 512X270	Size: 1024x540	Size: 2048x1080
1	0.39	1.43	5.46
2	0.27	0.95	3.67
4	0.2	0.7	2.63
8	0.17	0.57	2.2
16	0.15	0.51	1.88
32	0.15	0.5	1.84

From the figure 6, we can find that the speedup is more significant on large image size especially when the number of threads is large. For example, when the number of threads is 16 and 32, we can obviously

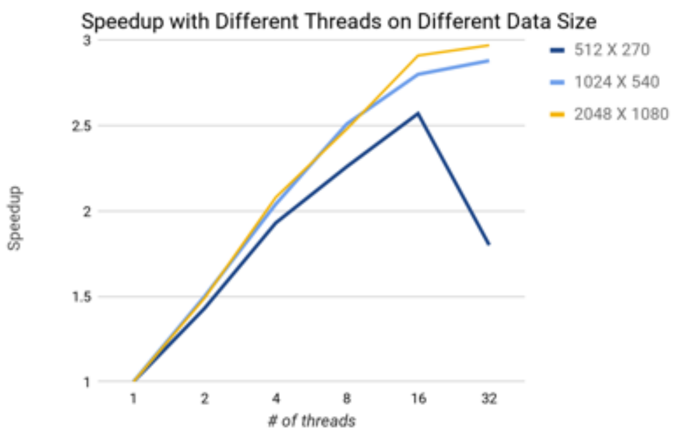


Fig 6 Speedup over single thread with different image sizes

observe that the speedup on 2048x1080 is larger than 1024x540, and 512x270 is the smallest. Furthermore, there is something interesting happened when the number of threads equals to 32. There is an obvious performance drop when the image size is 512x270. This is because when the problem size is not big enough, it can not overcome the overhead to create and manage multiple threads. Thus, this situation does not happen when the image size is 1024x540 or 2048x1080. However, in theory, there should be a drop with larger number of threads when the image size is 1024x540 or 2048x1080.

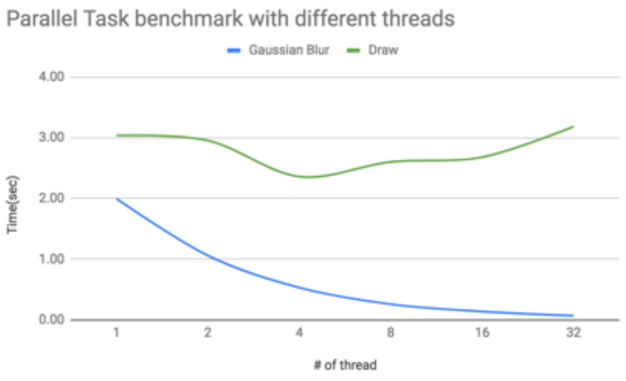


Fig 7 Task benchmark with different number of threads(Pointillist)

For the experiment (3), we use the revised version code which is the same as experiment (2) and record the time of gaussian blur task and draw task with different number of threads(# of thread = 1, 2, 4, 8, 16, 32) and use cap18-2250x1510 as input image and set the pointillist parameters. Then, we can get the figure 7. The reason we choose pointillist style to do this experiment is that instead of drawing a stroke, it draws a point each time. This makes the draw task become very simple. Thus, from the figure 7, we can find that compared to the impressionist style, we can not get too much speedup on draw task.

6 Conclusions

In our implementation, according to the above experiments, we discover the following findings:

- According to the experiment (1) and (3), we can not get speedup if the task is too simple. The work done by each thread is too little, so it can not overcome the overhead to create and manage multiple threads.
- According to the experiment (1), we can conclude that it is very important to make a good decision on which part should be parallelized, and before doing the parallelization, we should do some analysis first.
- According to the experiment (2), we can conclude that problem size is a very crucial factor to determine the speedup. If we do not have enough problem size, we can't get high utilization of CPU so that it can not overcome the overhead of thread creation and management.

7 Results

In this chapter, we will show some results in the figure 8, 9, 10 and 11 with different

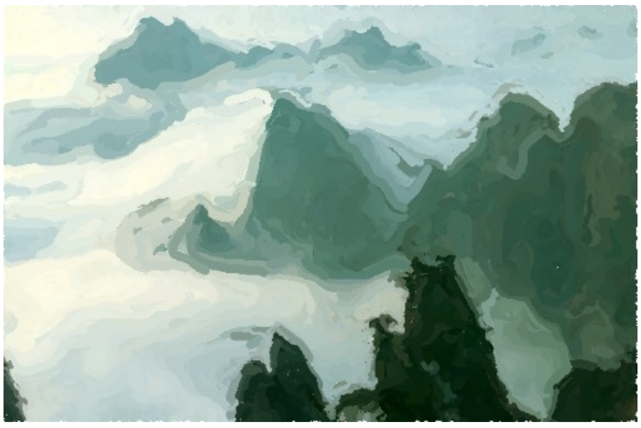


Fig 8 Impressionist

style and different image.

References

All the referenced links are hyperlinks that you can directly click the blue and underlined text to redirect to the referenced website.

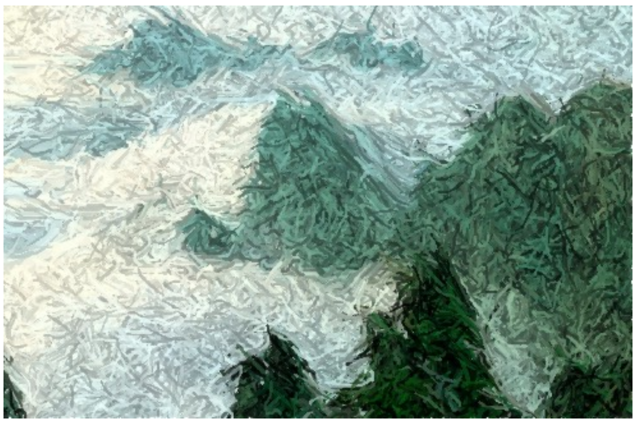


Fig 9 Expressionist

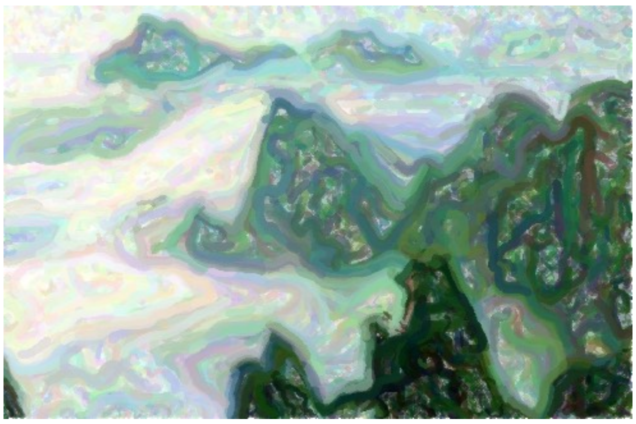


Fig 10 Colorist Wash

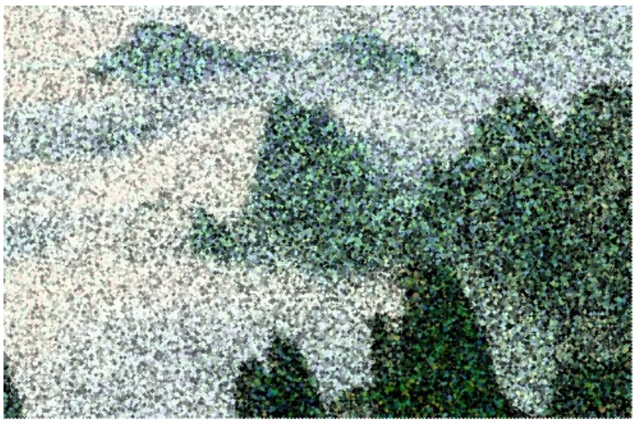


Fig 11 Pointillist