Parallelize Painterly Rendering with Multiple Stroke Size

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

Abstract

Since image processing utilizes lots of filter, we try to find the optimized parallelized implementation in this project. The paper we use is "Painterly Rendering with Curved Brush Strokes of Multiple Sizes" and we accelerate each image processing part with OpenMP. Finally, we compare each part with the traditional sequential CPU-based version and determine the part need to be parallelized.

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, he provided 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. He also proposed a method that the user can define different parameters to get different painting styles such as Impressionist, Expressionist, Colorist Wash, Pointillist. In his algorithm shown in the figure 2, 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.

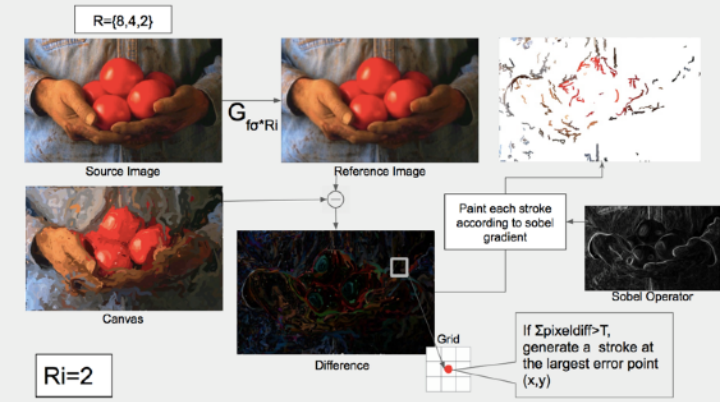

Fig. 2 The whole process of painterly rendering algorithm proposed in the paper

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 not well implemented in following reasons. First, the technique it used 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 inefficient comparing with multicore CPU. Because the communication overhead of CPU-to-GPU is higher than CPU-to-CPU, the image size and computation is not so huge that can compensate the communication overhead. Last, the stroke generation and drawing process cannot be parallelize on GPU, but can be parallelized on CPU. Therefore, we think it is necessary to propose our new implementation using multicore CPU using OpenMP rather than GPU.

3 Proposed Idea

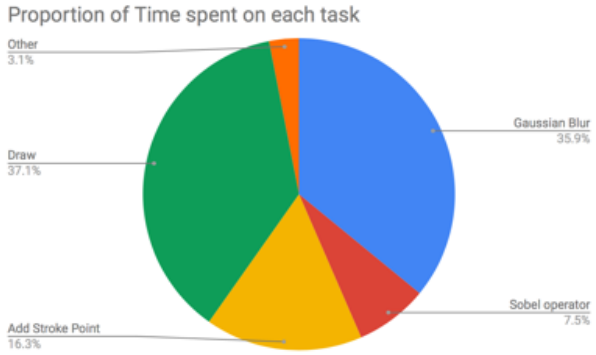

Fig 3 Proportion of time spent on each task with number of thread=1

We run a benchmark of each task to help us decide the part to parallelize.

In the figure 3, we can see that Gaussian blur, Sobel Operator, Add Strokepoint and Draw takes majority of computation. Therefore, we select them as our parallel target and propose method as bellowed.

```
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_(for R_i)
      // paint a layer
      paintLayer(canvas, referenceImage, R_i)
   }

   return canvas
}
```

Fig. 4 Paint Algorithm

As the proposed algorithm in the figure 4, for Gaussian blur, since each brush size uses different blur factor to get the reference image for each layer, the process do convolution with special gaussian kernel(mask), and do double precision multiplication work on each pixel independent. Thus, it is good place to do parallelization.

And the Sobel operator that calculate the gradient at each pixel to generate the stroke curve's direction can be parallelized in the same way, because it also do convolution and is independent from each pixel.

```
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 := Σ_{i,j∈M} D_{i,j} / grid^2
        if (areaError > T) then
        {
           // find the largest error point
           (x_1,y_1) := arg max_{i,j∈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. 5 paintLayer Algorithm

For add stroke point process in the figure 5, we find that the two for loops to sum the error can be parallelized, because there is no dependency on each pixel.

Last, the draw that paint all strokes in S on the canvas can be executed simultaneously, since we don't have to draw in certain order, and each stroke is independent from other strokes.

4 Experimental Setup

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 implementation, we use OpenCV to deal with image read, write, draw and pixel manipulation, and use OpenMP to do shared memory multiprocessing in C++.
For time measurement, we use both std::chrono::high_resolution_clock and linux time command. We use chromo::high_resolution_clock for

measuring the time of each task, and use linux 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. 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.

5 Experiments & Discussion

We will do three experiments: (1) Task benchmark with different number of threads using Impressionist style parameters (2) Task benchmark with different number of threads using Pointillist style parameters (3) speedup with different number of threads on different image size in this chapter.

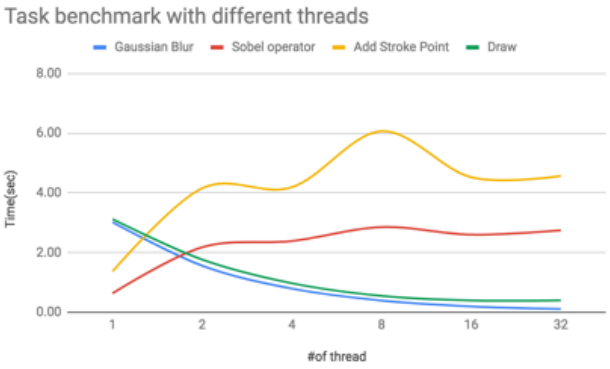

Fig 6 Task benchmark with different number of threads(Impressionist)

At first, we check our parallel result in 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 6. 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. This shows that not all tasks are deserved 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 on the single thread version.

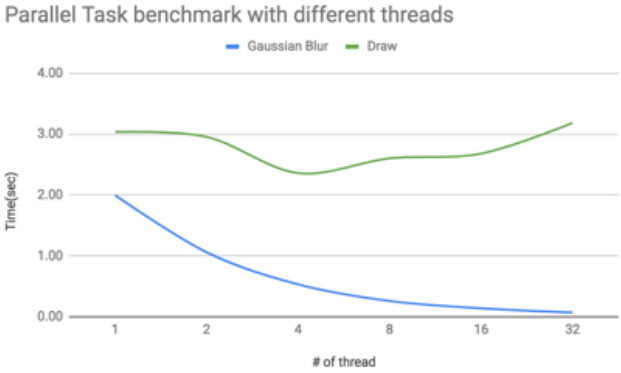

Fig 7 Task benchmark with different number of threads(Pointillist)

We verify this assumption in experiment (2) with pointillist style parameters. We use the revised version code which is the same as experiment (1) 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.

According to experiment (1) & (2), in order to get a better performance, we finalize our code to not parallelize sobel operator task and adding stroke points. Then, we do the

experiment (3) and record the time of each task with different number of threads(# of thread = 1, 2, 4, 8, 16, 32, 64) and use cap18-512x270, cap18-1024x540, cap18-2048x1080 as input images and set the impressionist parameters; then, we can get the table 1 which is shown in seconds. And from the table 1, we can draw the figure 8 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
64	0.17	0.53	1.94

From the figure 8, 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

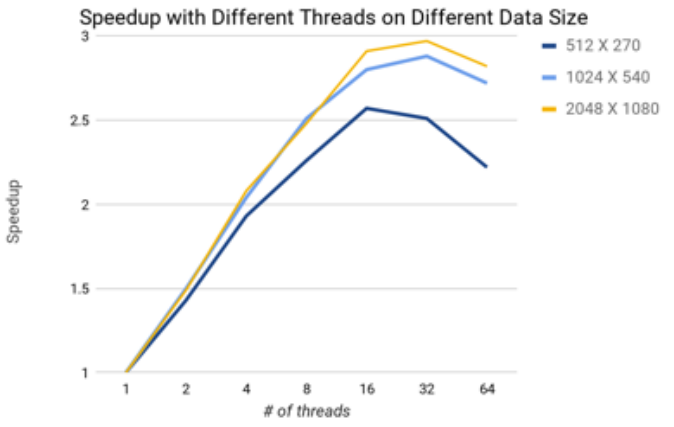

Fig 8 Speedup over single thread with different image sizes

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 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 when the number of threads is 32. But, we still meet a drop with larger number of threads(number of threads = 64) when the image size is 1024x540 or 2048x1080, because the overhead of thread creation and management is over the benefit of parallelization.

6 Conclusions

In our implementation, according to the above experiments, we can conclude that:

- The task property would affect the parallel performance. According to the experiment (1), we can find that even the task takes large portion of time, it doesn't mean it can gain benefit from parallelization. Combined with the result of experiment (2), we know that we could not gain benefit from parallelization 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.
- The problem size is a crucial factor to determine the speedup. According to the experiment (3), 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

We put different style images in figure 8, 9, 10 and 11.

References

[1] Aaron Hertzmann. "Painterly rendering with curved brush strokes of multiple sizes." Proc. SIGGRAPH 1998.
[2] "GPU-Accelerated Painterly Rendering", http://msmolens.github.io/painterly-rendering/

Fig 9 Impressionist

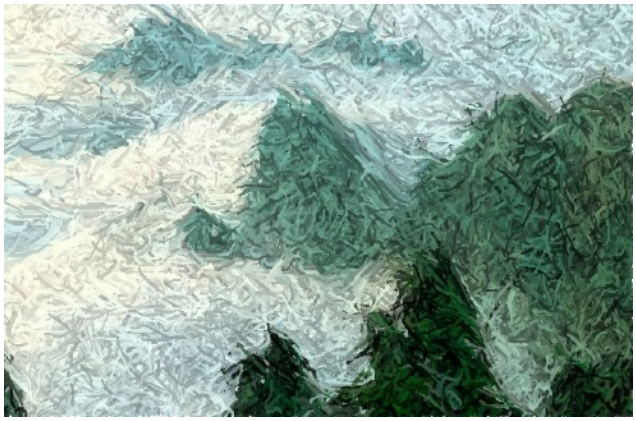

Fig 10 Expressionist

Fig 11 Colorist Wash

Fig 12 Pointillist