CUDA Circle Renderer

Parallel Computing 2019/2020

Claudia Raffaelli

Department of Information Engineering
University of Florence
Via di Santa Marta 3, Florence, Italy
claudia.raffaelli96@gmail.com

Abstract

This paper presents two separate approaches for the drawing of colored circles. The first version implements a sequential computing solution to the problem, with a single thread CPU. The second approach makes use of multithreaded programming to GPU, for a parallel alternative. The focus of this work is in particular on the GPU side, for which is provided an inside view of the techniques used to achieve the goal. The behavior of the two versions is then compared and validated with measurement data.

1. Introduction

The aim of this project is to create a renderer that draws colored circles on the screen, following a specific pattern. The renderer accepts an array of circles. Each of this circles is denoted by its position, or more precisely, by the position of its center (in terms of horizontal position, vertical position and depth). Also, every circle is characterized by a radius and a color.

An important detail of the application is that it renders semi-transparent circles. Therefore, the color of any one pixel of the image, is not the color of a single circle, but the result of blending the contributions of all the semi-transparent circles overlapping the pixel.

As a matter of fact the renderer represents the color of each circle via the three-channel RGB color model, supplemented with a 4^{th} alpha channel that indicates the opacity of each pixel. Because of that, composition is not commutative. It is important that the renderer draws the circles keeping the depth order provided by the application. While this task does not require to pay too much attention in the sequential approach, it can be challenging to carry on in the parallel version.

2. Overview of the two approaches

In this section is described the approach carried out in the single thread CPU computation.

Later on are presented the principal issues related to the parallelization of this particular application.

Keeping this problems in focus is then proposed the examination of the multithreaded computation version.

2.1. Single thread CPU approach

In the Algorithm 1 below, is well explained the proposed CPU algorithm, formulated using as cornerstone the work from CMU [1].

Algorithm 1 CPU Algorithm

```
1:  procedure RENDER
2:      Clear image
3:      for each circle do
4:          compute bounding box of the circle
5:          for each pixels in bounding box do
6:              compute pixel center point
7:              if center point is within the circle then
8:                  compute color of circle at point
9:                  blend circle's color into pixel's color
10:     return Image
```

2.2. Multithreading challenges

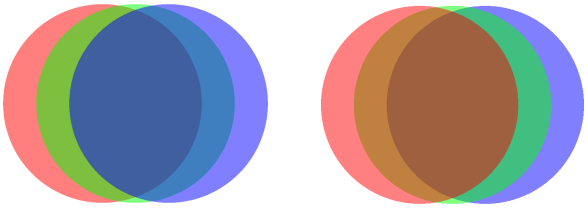
Because of the sequential nature of the CPU, the coloring order of each pixel is easily observed. On the other hand, in order to parallelize the task of rendering circles, the same ordering constrains needs to be complied.

The two main aspects upon which is important to pay particular attention in the rendering through parallel implementation are atomicity and order.

The atomicity invariant expect all image update operations to be atomic. The critical region that must be atomic includes reading the pixel's RGBA color, blending the contribution of the circle under consideration with the current image color value, and then writing the pixel's color back to memory.

Even the order must be taken into account. The renderer must perform updates to an image pixel in circle input order. Preserving the ordering requirement allows the correct rendering of semi-transparent circles. However there are no ordering requirements between circles that do not contribute to the same pixels.

In the image 1a below is reported an example of CPU rendering or correct GPU rendering, while on figure 1b an example of wrong GPU rendering, realized without observing the two invariants described above.



(a) Correct (b) Incorrect

Figure 1: Comparison between two GPU versions

2.3. Multithreaded GPU approach

In the course of this work has been used NVIDIA CUDA architecture to perform the requested task. The approach proposed in this paper as solution to parallel rendering, relies on a parallelization upon pixels.

As a first step the whole image is divided up into smaller squared portions. Each one of this fragment is assigned to a different thread block. Therefore every thread block takes care of a different image section. This results in a separated management of each image area that allows to perform a targeted parallelism. How the thread block sizes are chosen is a very crucial point to consider in order to guarantee a good computation outcome. Later on, in the Experiment section will be discussed different tests involving this particular aspect of computation.

The technique adopted subsequently consists in finding which circles are present within a certain image area. In order to do so, each thread in a block handles a different portion of the original array containing all circles present inside the image.

Every thread checks if the circles of its competence are located inside the current block, even for one pixel. At the end of computation all circles are checked to see if they are inside a block. This process happens for every block. All the circles found within a block are added to a restricted array of circles, henceforth called block's circle list. This concludes Phase 1 of the rendering algorithm.

The second step consists in assigning each thread in a block, to a pixel in that same block. For every pixel is then applied a sequential checking. In other words, making use of the block's circle list previously made, for each circle in that list, is checked if the current pixel is part of the circle at issue.

By doing so the atomicity invariant is guaranteed. Many circles can be taken care of, at the same time, but each pixel is handled separately. The critical region of each pixel is then entered only by one thread throughout computation.

Moreover, also the ordering is protected. In Phase 1, while generating the block's circle lists, the circles were added observing the depth ordering. The new list produced is only smaller compared with the original one, but with the same ordering. Because of the way the circles are accessed sequentially for each pixel, the invariant is preserved.

This can be considered the end of Phase 2. The obtained image is the same as the one we would have got using a single threaded CPU.

The process as described can be summarized in the pseudocode Algorithm 2 below.

Algorithm 2 GPU CUDA Algorithm

```
1:  procedure RENDER
2:                                                  ▷ Phase 1:
3:      division of empty image in smaller blocks
4:      each thread in block gets a small portion of circles
5:      for each thread in a block do
6:          for each circle in small portion of circles do
7:              if circle belongs to current block then
8:                  add circle to block's circle list
9:                                                  ▷ Phase 2:
10:     assign to each thread in a block a pixel
11:     for each pixel/thread in a block do
12:         for each circle in block's circle list do
13:             if pixel belongs to circle then
14:                 compute color of circle at point
15:                 blend circle's color into pixel's color
16:     return Image
```

3. Experiments

In this section will be presented a series of different experiments conducted on the code discussed, changing some parameters and using two distinct architectures.

3.1. Architecture overview

In order to perform the analysis of performances, has been adopted for this task two architectures. Below are are submitted their specifics:

The first system, henceforth called S1 is equipped with:
CPU: Intel Core i7-3630QM (2.40GHz)
GPU: NVIDIA GeForce GT 630M
2 Streaming multiprocessors, compute capability 2.1,
96 CUDA cores, global memory 1985 MB
CUDA libraries: 8
Operating System: Ubuntu 16.4

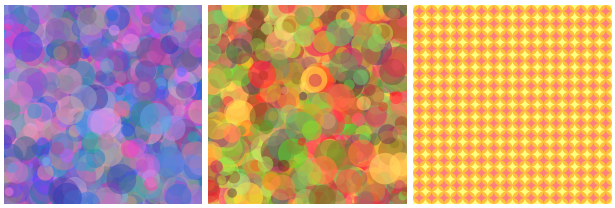
The second system, called from now on S2, on the other hand, comprises of:
CPU: Intel Core i7-860 (2.80GHz)
GPU: NVIDIA GeForce GTX 980
16 Streaming multiprocessors, compute capability 5.2,
2048 CUDA cores, global memory 4036 MB
CUDA libraries: 10
Operating System: Ubuntu MATE

3.2. Description of the experiments

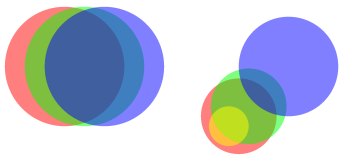
Taking into account the specifics of both systems is undoubtedly expected a different behavior of the program while running on S1 or S2.

Besides comparing the results obtained from the two machines, the size of the blocks is adjusted between the values 16×16 and 32×32 threads per block.

Also the experiments are conducted upon five different patterns visible in the Figure below, namely: Rand10K 2a, Rand100K 2b, Pattern 2c, RGB 2d and RGBY 2e .



(a) Rand10K (b) Rand100K (c) Pattern



(d) RGB (e) RGBY

To evaluate the performances, the program is let run on 10 frames of a same pattern. Then is computed the average time of execution and used the latter as resulting time.

3.3. Evaluation of performances

In this section, are shown the experiments conducted for each pattern, in which the two architectures are compared. Are also timed and reported the differences between the two block dimensions (16×16 and 32×32) for some of the patterns. Note that the times are expressed in milliseconds.

	CPU	GPU	Speedup
S1 16×16 blocks	626.57	354.60	1.77
S1 32×32 blocks	628.38	177.14	3.55
S2 16×16 blocks	1297.49	12.29	105.59
S2 32×32 blocks	1232.64	7.57	162.75

Table 1: Rand10K

	CPU	GPU	Speedup
S1 32×32 blocks	6257.69	2332.27	2.68
S2 32×32 blocks	13227.0	64.88	203.87

Table 2: Rand100K

	CPU	GPU	Speedup
S1 32×32 blocks	12.37	6.72	1.84
S2 32×32 blocks	29.36	0.41	71.24

Table 3: Pattern

	CPU	GPU	Speedup
S1 32×32 blocks	5.80	4.02	1.44
S2 32×32 blocks	11.58	0.26	45.17

Table 4: RGB

	CPU	GPU	Speedup
S1 16×16 blocks	3.16	2.29	1.38
S1 32×32 blocks	3.21	3.10	1.04
S2 16×16 blocks	6.97	0.17	40.92
S2 32×32 blocks	6.48	0.18	34.95

Table 5: RGBY

As first thing are analyzed the differences among having the blocks with dimension 16×16 and 32×32. Looking at Rand10K table, and paying attention at the speedup column, is clearly visible an improvement, switching by the smaller block dimension to the bigger. The same results can be obtained with other patterns that require many circles to render on screen.

On the other hand, by looking at RGBY table, the smaller block dimension results more adequate. The same thing applies to patterns with few circles. This can be explained by the smaller number of threads required to perform the task on the latter case.

Considering the results produced by the CPU and CUDA, the GPU performs generally better. A speedup is always present. Looking first at the architecture S1, the speedup obtained is less pronounced than it is in S2.

Of course this was the expected behavior of the two systems. Anyhow, for both systems, the speedup outcome can be linked to the number of circles involved. The GPU turns out to be way faster than the CPU when the renderer has to draw a lot of circles on screen. The maximum speedup obtained is visible in table 2, at a high 203.87x for S2. The lowest speedup, still looking at S2, can be found in table 5, with a low 34.95x. This outcome was easily foreseeable. The number of circles in RGBY pattern, is merely four. The overhead for generating all the threads undermines all possible benefits from parallelizing the task.

The analysis produced, then, indicates that the more the circles, the more the speedup of the GPU from the CPU.

Finally the conducted tests are summarized in the following chart.

Figure 3 is obtained by using all speedup data of the second architecture, with block dimension 32×32. Each pattern adds a contribution to the chart by making explicit the relation between the the number of circles rendered, and the speedup obtained.

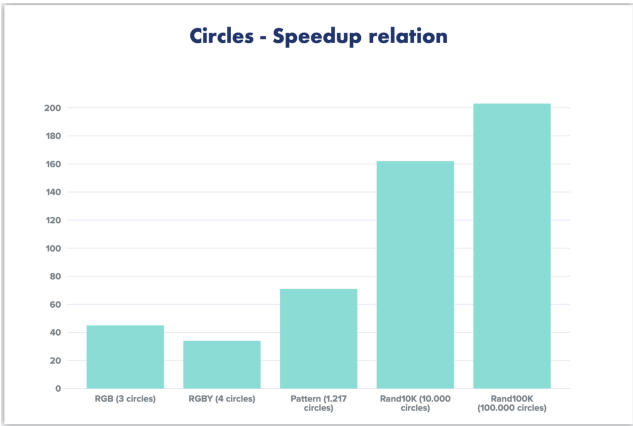


Figure 3: Speedup relation with the number of circles

To conclude the analysis performed, the parallelization of circles appears to be really useful when the circles to render are many. On the other hand the overhead resulting from parallelizing few circles is too high in order to be really beneficial for the rendering task.

References

[1] Carnegie Mellon University. *A Simple CUDA Renderer*. http://15418.courses.cs.cmu.edu/fall2016/article/4