CUDA Circle Renderer

Parallelized rendering of colored circles on screen using NVIDIA CUDA

Claudia Raffaelli

A.A. 2019/2020

Introduction

What is the purpose of this project ?

The aim of this presentation is to show two possible approaches on the task of drawing on screen colored circles, following a specific pattern.

- The first version implements a **sequential computing** solution to the problem, with a single thread CPU.
- The second approach makes use of **multi-threaded programming** to GPU, for a **parallel** alternative.

The focus of this work is in particular on the GPU side. The behavior of the two versions is then compared and validated with measurement data.

Introduction

The task to achieve

The task to achieve concerns the ability to draw circles on screen following a specific pattern.

Whatever it is the pattern to follow, the renderer accepts an array of circles characterized by:

- A position, or more precisely, by the **3D position** of its center,
- A **radius**,
- An RGBA **color**

An important detail of the application is that it renders **semi-transparent** circles. 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.

Rendering Challenges

The task to achieve

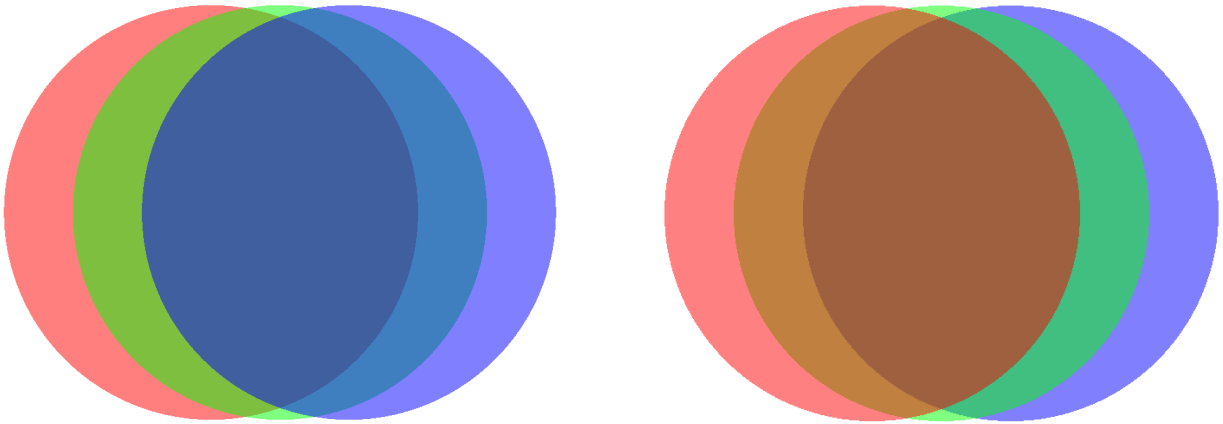
The two main aspects upon which is important to pay particular attention to are:

- **Atomicity:** 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,
  - **writing** the pixel's **color** back to memory.

- **Order:** Even the order must be taken into account. 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. 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.

Rendering Challenges

Possible wrong output



Comparison between a **correct implementation vs incorrect rendering**. The input ordering of the circles was: Red, Green, Blue.

Because of the transparency, the resulting output is different if the right ordering is not followed.

Single thread CPU Renderer

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
```

Because of the sequential nature of the CPU, the coloring order of each pixel is easily observed and so is the atomicity invariant.

Multi-threaded GPU CUDA Renderer

The same cannot be said in the **parallel version** of the renderer.

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
```

Multi-threaded GPU CUDA Renderer

Phase 1

As a first step:

1. The whole **image** is **divided** up into smaller squared portions.

2. 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.

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
```

Multi-threaded GPU CUDA Renderer

Phase 1

3. The next step 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.

4. At the end of computation all **circles** are **checked** to see **if** they are **inside** a block. This process happens for every block.

5. All the circles found within a block are added to a **restricted array of circles**, henceforth called **block's circle list**.

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
```

Multi-threaded GPU CUDA Renderer

Phase 2

The second step consists in:

1. **Assigning** each **thread** in a block, **to a pixel** in that same block.

2. 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.

3. If that is the case the **color pixel is updated**

The obtained image is the same as the one we would have got using a single threaded CPU.

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
```

Multi-threaded GPU CUDA Renderer

Conclusion

Following this algorithm then:

- 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.

- 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. There's an array that contains for each thread the number of circles found inside that block. Using a given function each thread gets the correct starting index from which copy back to block's circle lists, the circles. Because of the way the **circles** are **accessed sequentially** for each pixel, the invariant is preserved.

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
```

Experiments

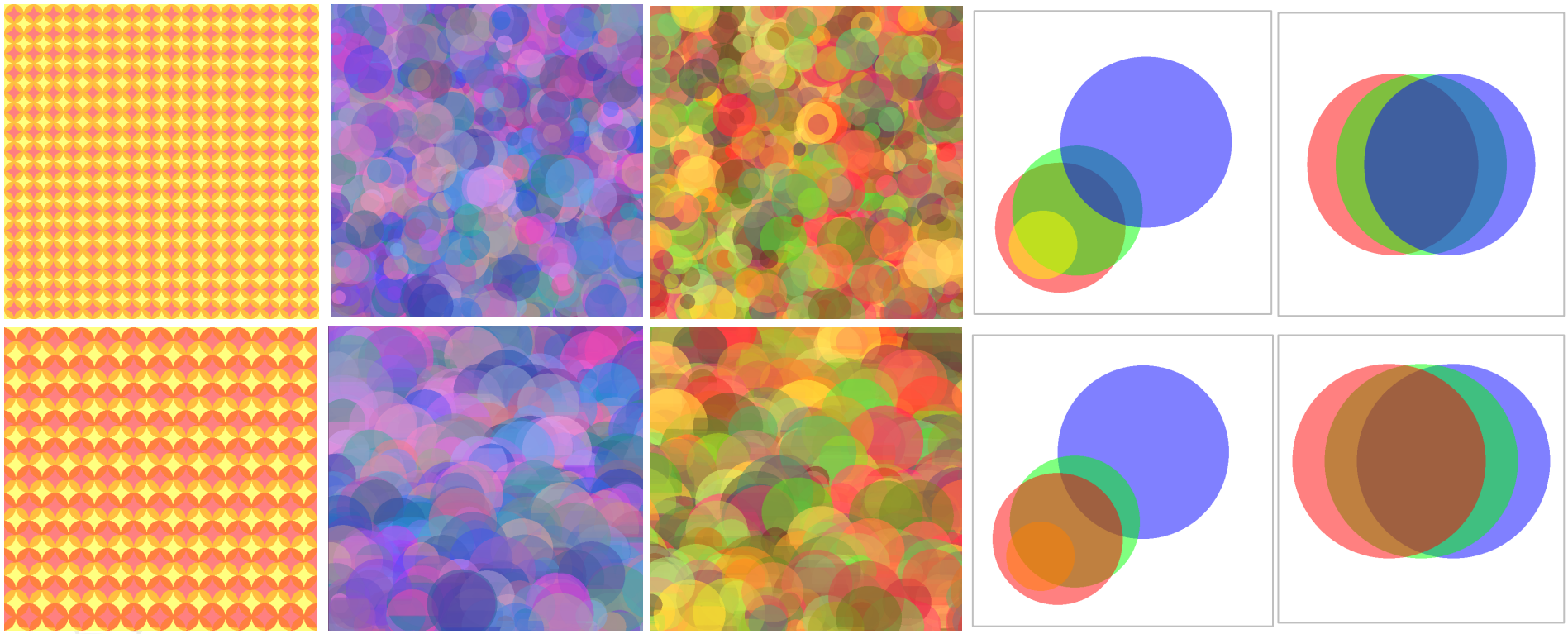
Architecture specifics

To conduct the experiments were used two architectures:

- The system, henceforth called S1 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 system, S2:
  - **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

Experiments

Examples of right and wrong CUDA implementation



a) Pattern *b) Rand10K* *c) Rand100K* *d) RGBY* *e) RGB*

Experiments

Description of the experiments

The main experiments conducted about the five patterns involved:

- Studying the behavior of the two systems S1 and S2,
- Adjusting the size of threads per block between the values 16 × 16 and 32 × 32,
- Computing the speedup between CPU and GPU to identify improved performances

The output obtained are a result of **ten frames** computation of the same pattern. Then is computed the average time of execution (in milliseconds) and used the latter as resulting time.

Experiments

Results

	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

Looking at the tables we find out that:

- Blocks of 16 × 16 are more convenient in the images with few circles, and 32 × 32 for the images with many circles. This can be explained by the smaller number of threads required to perform the task in the first case.
- Considering the results produced by the CPU and CUDA, the **GPU** performs generally **better**.
- The speedup obtained in S1 is less pronounced than it is in S2: as expected.

	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

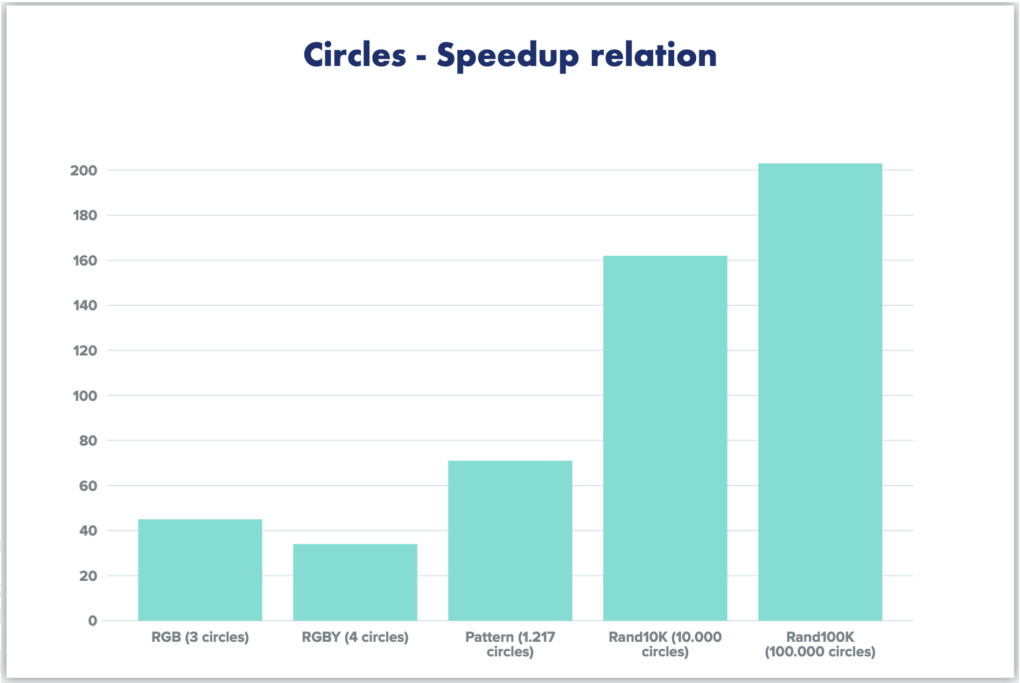
Table 5: RGBY

Experiments

Conclusions

In conclusion:

- 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 **overhead** for generating all the threads undermines all possible benefits from parallelizing the task when the circles are **few**. So: the more the circles, the more the speedup.



To conclude the analysis performed, the parallelization of circles appears to be really **useful** when the **circles** to render are **many**.

On the other end the **overhead** resulting from parallelizing **few** circles is too high in order to be really beneficial for the rendering task.