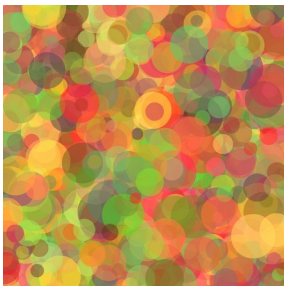
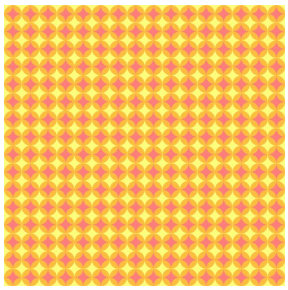
Part3

A Simple CUDA Renderer

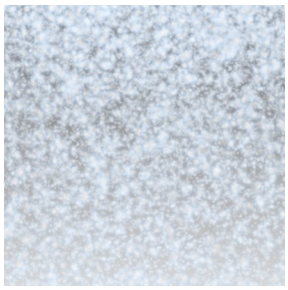
Random10K



Random 100K



Pattern



Snow

PLEASE NOTE: you can not view the images generated by your code visually on HPCLAB machine. However you can generate pictures in PPM format and download (scp / sftp) them to your local machine to visually inspect them.

Overview

In this assignment you will write a parallel renderer in CUDA that draws colored circles. While this renderer is very simple, parallelizing the renderer will require you to design and implement data structures that can be efficiently constructed and manipulated in parallel. This is a **challenging assignment** so you are advised to start early. Seriously, you are advised to start early. Good luck!

A Simple Circle Renderer (85pts)

Now for the real show!

The directory render of the assignment starter code contains an implementation of a renderer that draws colored circles. Build the code, and run the renderer with the command line "./render rgb." You will see an image of three circles appear on screen ('q' closes the window). Now run the renderer with the command line "./render snow." You should see an animation of falling snow. The assignment starter code contains two versions of the renderer: a sequential, single-threaded C++ reference implementation, implemented in **refRenderer.cpp**, and an incorrect parallel CUDA implementation in **cudaRenderer.cu**

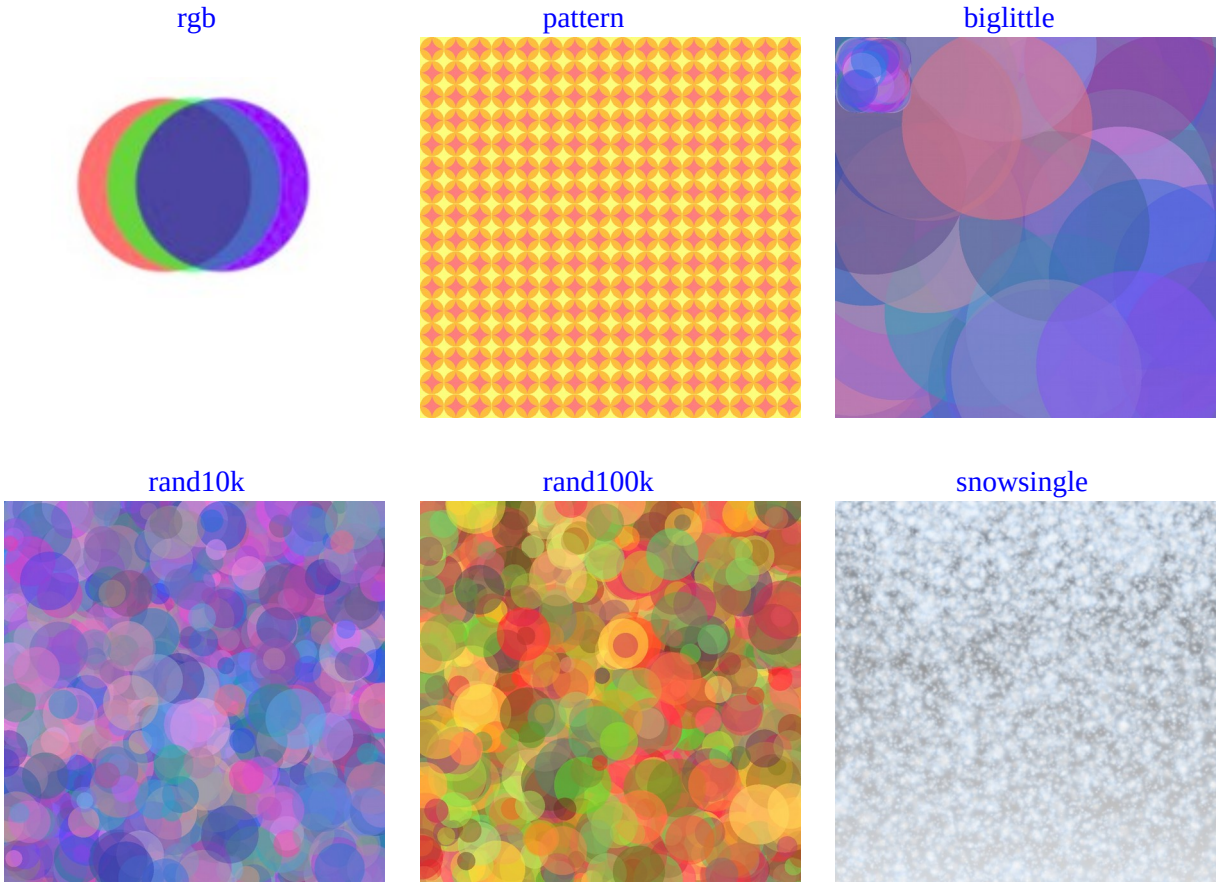


Figure 4:Sample images generated by renderer

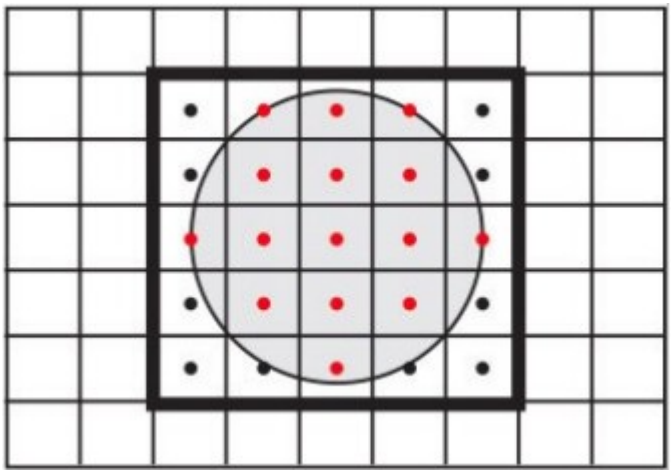


Figure 5:Computing the contribution of a circle to the output image

Renderer Overview

We encourage you to familiarize yourself with the structure of the renderer code base by inspecting the reference implementation in refRenderer.cpp. The method setup() is called prior to rendering the first frame. In your CUDA-accelerated renderer, this method will likely contain all your renderer initialization code (allocating buffers, etc). The render() method is called for each frame and is responsible for drawing all circles in the output image. The other main function of the renderer, advanceAnimation(), is also invoked once per frame. It updates circle positions and velocities. You will not need to modify advanceAnimation() in this assignment. The renderer accepts an array of circles (3D position, velocity, radius, color) as input. The basic sequential algorithm for rendering each frame is:

```
Clear image
For each circle:
    Update position and velocity
For each circle:
    Compute screen bounding box
    For all pixels in bounding box:
        Compute pixel center point
        If center point is within the circle:
            Compute color of circle at point
            Blend contribution of circle into image for this pixel
```

Figure 5 illustrates the basic algorithm for computing circle-pixel coverage using point-in-circle tests. All pixels within the circle's bounding box are tested for coverage. For each pixel in the bounding box, the pixel is considered to be covered by the circle if its center point(shown with a dot) is contained within the circle.

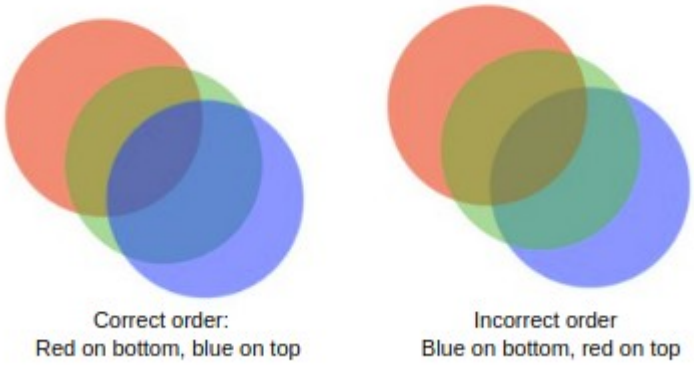


Figure 6: Ordering dependency of rendering

Pixel centers that are inside the circle are colored red, while those that are outside the circle are colored black. The circle's contribution to the image will be computed only for covered pixels. An important detail of the renderer is that it renders **semi-transparent** circles. Therefore, the color of any one pixel is not the color of a single circle, but the result of blending the contributions of all the semi-transparent circles overlapping the pixel (note the "blend contribution" part of the pseudocode above). The renderer represents the color of a circle via a 4-tuple of red (R), green (G), blue (B), and opacity (alpha, written "α") values (RGBA). Alpha value α = 1.0 corresponds to a fully opaque circle. Alpha value α = 0.0 corresponds to a fully transparent circle. To draw a semi-transparent circle with color (C_r, C_g, C_b, α) on top of a pixel with color P_r, P_g, P_b, the renderer performs the following computation:

$$R_r = \alpha \cdot C_r + (1.0 - \alpha) \cdot P_r$$

$$R_g = \alpha \cdot C_g + (1.0 - \alpha) \cdot P_g$$

$$R_b = \alpha \cdot C_b + (1.0 - \alpha) \cdot P_b$$

Notice that composition is not commutative (object *X* over *Y* does not look the same as object *Y* over *X*), so it's important to render circles in a manner that follows the order they are provided by the application.(You can assume the application provides the circles in depth order.)For example, consider the two images shown in Figure6,where the circles are ordered as red, green, and blue. The image on the left shows them rendered in the correct order, while the image on the right has them reversed.

CUDA Renderer

After familiarizing yourself with the circle rendering algorithm as implemented in the reference code, now study the CUDA implementation of the renderer provided in ***cudaRenderer.cu***. You can run the CUDA implementation of the renderer using the command-line option "***-r* cuda**."

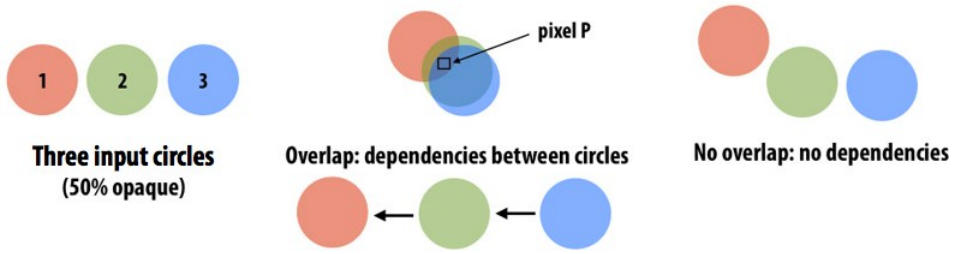


Figure 7:Illustration of ordering dependencies for rendering

The provided CUDA implementation parallelizes computation across all input circles, assigning one circle to each CUDA thread. While this CUDA implementation is a complete implementation of the mathematics of a circle renderer, it contains several major errors that you will fix in this assignment. Specifically, the current implementation does not ensure that image update is an atomic operation, and it does not preserve the required order of image updates, both of which are described below

Renderer Requirements

Your parallel CUDA renderer implementation must maintain two invariants that are preserved trivially in the sequential implementation:

1. **Atomicity**: All image update operations must be atomic. The critical region includes reading the four32-bit floating-point values (the pixel's rgba color), blending the contribution of the current circle with the current image value, and then writing the pixel's color back to memory.

2. **Order**: Your renderer must perform updates to an image pixel in circle input order. That is, if circle 1 and circle 2 both contribute to pixel P, any image updates to P due to circle 1 must be applied to the image before updates to P due to circle 2. As discussed above, preserving the ordering requirement allows for a correct rendering of transparent circles. **A key observation is that the definition of order only specifies the order of updates to an individual pixel**. Thus, as shown in *Figure 7*, there is no ordering requirement between circles that do not contribute to the same pixel. These circles can be processed independently.

Since the provided CUDA implementation does not satisfy either of these requirements, the result of not correctly respecting order or atomicity can be seen by running the CUDA renderer implementation on the different scenes. You may see horizontal streaks through the resulting images, as shown in *Figure 8* for the rgb scene. These streaks will change with each frame. If not streaks, you will see cases where the program renders the circles in the wrong order.

What You Need To Do

Your job is to write the fastest, correct CUDA renderer implementation you can. You may take any

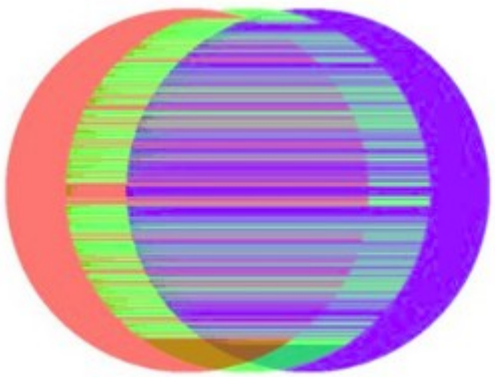


Figure 8:Incorrect rendering by provided CUDA implementation

approach you see fit, but your renderer must adhere to the atomicity and order requirements specified above. We have already given you such a solution!

Please Note : There is one optimization you may not perform : you should render each frame independently, even if one frame contains the same circles as the previous one.

A good place to start would be to read through ***cudaRenderer.cu*** and convince yourself that it doesnot meet the correctness requirement. (Look specifically at the CUDA kernel ***kernelRenderCircle***, and the inline function ***shadePixel***.) To visually see the effect of violation of above two requirements,compile the program with **make**. Then run:

./render -r cuda rand10k

Compare this image with the one generated by sequential code by running

./render -r ref rand10k

(This image is shown in the lower-left corner of *Figure4*.)

You can get a listing of the options to the render program by running

./render --help

Checker code: To detect correctness of the program, ***render*** has a convenient "***--check***" option. This option runs the sequential version of the reference CPU renderer along with your CUDA renderer and then compares the resulting images to ensure correctness. The time taken by your CUDA renderer implementation is also printed.

There are a total of six circle data sets on which you will be graded for performance. However, in order to receive full credit, your code must pass the correctness test for all of our test data sets. There are some hidden tests for preventing you from tailoring the program to specific image size or opacity. The hidden tests have varied image size (w.l.o.g you could assume the size of image is multiple of 64), alpha value, number of circles, etc. To check the correctness and performance score of your code, run "**make check**" (or directly call the program **checker.pl**.) If you run it on the starter code, the program will print the results for the entire test set, plus a table like the following:

```
------------
Score table:
------------
-------------------------------------------------------------------------
| Scene Name      | Target Time     | Your Time       | Score           |
-------------------------------------------------------------------------
| rgb             | 1.2117          | 120.0979 (F)    | 0               |
| rand10k         | 5.8035          | 25.2887 (F)     | 0               |
| rand100k        | 59.6547         | 528.2250 (F)    | 0               |
| pattern         | 1.2956          | 3.6817 (F)      | 0               |
| snowsingle      | 30.9009         | 9.6775 (F)      | 0               |
| biglittle       | 36.4317         | 307.6269 (F)    | 0               |
-------------------------------------------------------------------------
|                                   | Total score:    | 0/72            |
-------------------------------------------------------------------------
```

Note: on some runs, you may receive credit for some of these scenes, since the provided renderer's runtime is non-deterministic. This doesn't change the fact that the current CUDA renderer is incorrect.

"Target time" is the performance of a good solution on your current machine (in the provided **render_soln** executable.) "Your time" is the performance of your CUDA renderer. Your grade will depend on the performance of your implementation compared to that of the provided implementation (see Grading Guidelines.)

Along with your code, we would like you to hand in a clear, high-level description of how your implementation works as well as a brief description of how you arrived at this solution. Specifically address approaches you tried along the way, and how you went about determining how to optimize your code (For example, what measurements did you perform to guide your optimization efforts?).

Aspects of your work that you should mention in the write-up include:

1. Include both partner's names and your team-Id the top of your write up.
2. Replicate the score table generated for your solution and specify which machine you ran your code on.
3. Describe how you decomposed the problem and how you assigned work to CUDA thread blocks and threads (and may be even warps.)
4. Describe where synchronization occurs in your solution.
5. What, if any, steps did you take to reduce communication requirements (e.g., synchronization or main memory band width requirements)?
6. Briefly describe how you arrived at your final solution. What other approaches did you try along the way. What was wrong with them?

Grading Guidelines

- The write-up for the assignment is worth 13points.
- Your implementation is worth 72 points. These are equally divided into 12 points per scene as follows:
  - 2 correctness points per scene.
  - 10 performance points per scene (only obtainable if the solution is correct). Your performance will be graded with respect to the target performance of a provided renderer, T_s.
  - No performance points will be given for solutions having time $T > 10T_s$.
  - Full performance points will be given for solutions within 20% of the optimized solution ($T \leq 1.2T_s$.)
  - For other values of T, your performance score on a scale 1 to 9 will be calculated as a linear interpolation based on the value of T_s / T.
  - If your code
    1) fails to pass the correctness test for provided test data sets other than above six
    2) fails to pass the correctness test for hidden cases
    3) has unreasonable performance on hidden cases (e.g. T_s / T halved with small change on image size alpha) 10 points per scene will be deducted as a penalty. Up to 20 points in total.
- Up to **20 points extra credit** (instructor discretion) for solutions that achieve significantly greater performance than required. Your write up must clearly explain your approach thoroughly.
- Up to **20 points extra credit** (instructor discretion) for a high-quality parallel CPU-only renderer implementation that achieves good utilization of all cores and SIMD vector units of the cores with ISPC or any other programming frameworks. To receive credit you should analyze the performance of your GPU and CPU-based solutions and discuss the reasons for differences in implementation choices made.

Assignment Tips and Hints

Below are a set of tips and hints compiled. Note that there are various ways to implement your renderer and not all hints may apply to your approach.

- To facilitate remote development and benchmarking, we have created a "**--bench**" option to the render program. This mode does not open a display, and instead runs the renderer for the specified number of frames. (You'll see that the checking code runs with the command-line option "**--bench 0:4.**")
- When in benchmark mode, the "**--file Name**" command-line option sets the base file name for PPM images created at each frame. Created files have names of the form "**Name_xxxx.ppm**," where **xxxx** is a 4-digit frame number. No PPM files are created if the **--file** option is not used.
- There are two potential axes of parallelism in this assignment. One axis is parallelism across pixels another is parallelism across circles (provided the ordering requirement is respected for overlapping.
- You are allowed to use the **Thrust** library (http://thrust.github.io/) in your implementation if you so choose. **Thrust** is not necessary to achieve the performance of the optimized CUDA reference implementations circles.)
- Is there data reuse in the renderer? What can be done to exploit this reuse?
- The circle-intersects-box tests provided to you in **circleBoxTest.cu_inl** are your friend.
- How will you ensure atomicity of image update since there is no CUDA language primitive that performs the logic of the image update operation atomically? Constructing a lock out of global memory atomic operations is one solution, but keep in mind that even if your image update is atomic, the updates must be performed in the required order. **We suggest that you think about ensuring order in your parallel solution first, and only then consider the atomicity problem (if it still exists at all) in your solution**.
- If you are having difficulty debugging your CUDA code, you can use printf directly from device code if you use a sufficiently new GPU and CUDA library: see this brief guide (http://15418.courses.cs.cmu.edu/fall2017/article/5) on how to print from CUDA.

Catching CUDA Errors

(Credit: The following was adapted from this Stack Overflow post)

By default, if you access an array out of bounds, allocate too much memory, or otherwise cause an error, CUDA won't normally inform you; instead it will just fail silently and return an error code. You can use the following macro (feel free to modify it) to wrap CUDA calls:

```
#define DEBUG #ifdef

DEBUG
#define cudaCheckError(ans) cudaAssert((ans), __FILE__, __LINE__);
inline void cudaAssert(cudaError_t code, const char *file, int line, bool abort=true)
{
    if (code != cudaSuccess)
    {
        fprintf(stderr, "CUDA Error: %s at %s:%d\n", cudaGetErrorString(code), file,
            line);
        if (abort) exit(code);
    }
}
#else
#define cudaCheckError(ans) ans #endif
```

Note that you can undefine **DEBUG** to disable error checking once your code is correct for improved performance.

You can then wrap CUDA API calls to process their returned errors as such:

cudaCheckError(cudaMalloc(&a, size*sizeof(int)));

Note that you can't wrap kernel launches directly. Instead, their errors will be caught on the next CUDA call you wrap:

kernel<<<1,1>>>(a); // suppose kernel causes an error!
cudaCheckError(cudaDeviceSynchronize()); // error is printed on this line

All CUDA API functions, ***cudaDeviceSynchronize(), cudaMemcpy(), cudaMemset()***, and so on can be wrapped.

IMPORTANT:if a CUDA function caused an error previously, but it wasn't caught, that error will show up in the next error check, even if the check wraps a different function. For example:

...
line 742: ***cudaMalloc(&a, -1);*** // executes, then continues
line 743: ***cudaCheckError(cudaMemcpy(a,b));*** // Prints error for line 743
...

Therefore, while debugging,it's recommended that you wrap **all** CUDA API calls (at least in code that you wrote).

Hand-in Instructions

You will submit via Blackboard. For the code, you will be submitting your entire directory tree.

1. Your code
   (a) One submission per group is sufficient.
   (b) Make sure all of your code is compilable and runnable. We should be able to simply run make, then execute your programs in render without manual intervention.
   (c) Remove all nonessential files, especially output images, from your directories.
   (d) Run the command "make handin.tar." This will run "make clean" and then create an archive of your entire directory tree.
   (e) Submit the filehandin.tar.