Project 3-A: GPU Render

Last Updated: October 29, 2025

In this project, 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!*

Contents

1 Due dates

- ☐ *Final submission & write-up:* 10:00 P.M. on Monday, December 8, 2025

2 Getting started

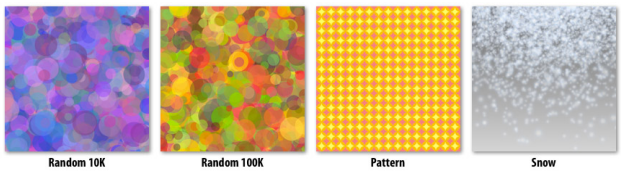
Download the starter code from Github:

`https://github.com/spe-fall25/project3-A_<your netid>.git}`

The CUDA C programmer's guide [PDF version] or [web version] is an excellent reference for learning how to program in CUDA. There are a wealth of CUDA tutorials and SDK examples on the web (just Google!) and on the [NVIDIA developer site]. In particular, you may enjoy the free Udacity course [Introduction to Parallel Programming in CUDA].

Table 21 in the [CUDA C Programming Guide] is a handy reference for the maximum number of CUDA threads per thread block, size of thread block, shared memory, etc for the NVIDIA RTX 2000 Ada Generation GPU you will used in this assignment. NVIDIA RTX 2000 Ada Generation support CUDA compute capability 8.9.

For C++ questions (like what does the keyword mean), the [C++ Super-FAQ] is a great resource that explains things in a way that's detailed yet easy to understand (unlike a lot of C++ resources), and was co-written by Bjarne Stroustrup, the creator of C++!



3 Part 1: CUDA Warm-Up 1: SAXPY

To gain a bit of practice writing CUDA programs, your warm-up task is to implement the SAXPY function in CUDA. Starter code for this part of the assignment is located in the `/saxpy` directory of the assignment repository. You can build and run the saxpy CUDA program by calling `make` and `./cudaSaxpy` in the `/saxpy` directory.

Please finish off the implementation of SAXPY in the function `saxpyCuda` in `saxpy.cu`. You will need to allocate device global memory arrays and copy the contents of the host input arrays `X`, `Y`, and `result` into CUDA device memory prior to performing the computation. After the CUDA computation is complete, the result must be copied back into host memory. Please see the definition of `cudaMemcpy` function in Section 3.2.2 of the Programmer's Guide (web version), or take a look at the helpful tutorial pointed to in the assignment starter code.

As part of your implementation, add timers around the CUDA kernel invocation in `saxpyCuda`. After your additions, your program should time two executions:

- The provided starter code contains timers that measure **the entire process** of copying data to the GPU, running the kernel, and copying data back to the CPU.
- You should also insert timers the measure *only the time taken to run the kernel*. (They should not include the time of CPU-to-GPU data transfer or transfer of results from the GPU back to the CPU.)

When adding your timing code in the latter case, you'll need to be careful: By defult a CUDA kernel's execution on the GPU is *asynchronous* with the main application thread running on the CPU. For example, if you write code that looks like this:

```
double startTime = CycleTimer::currentSeconds();
saxpy_kernel<<<blocks, threadsPerBlock>>>(N, alpha, device_x, device_y, device_result);
double endTime = CycleTimer::currentSeconds();
```

You'll measure a kernel execution time that seems amazingly fast! (Because you are only timing the cost of the API call itself, not the cost of actually executing the resulting computation on the GPU.

Therefore, you will want to place a call to `cudaDeviceSynchronize()` following the kernel call to wait for completion of all CUDA work on the GPU. This call to `cudaDeviceSynchronize()` returns when all prior CUDA work on the GPU has completed. Note that `cudaDeviceSynchronize()` is not necessary after the `cudaMemcpy()` to ensure the memory transfer to the GPU is complete, since `cudaMempy()` is synchronous under the conditions we are using it. (For those that wish to know more, see [this documentation].)

```
double startTime = CycleTimer::currentSeconds();
saxpy_kernel<<<blocks, threadsPerBlock>>>(N, alpha, device_x, device_y, device_result);
cudaDeviceSynchronize();
double endTime = CycleTimer::currentSeconds();
```

Note that in your measurements that include the time to transfer to and from the CPU, a call to `cudaDeviceSynchronize()` **is not** necessary before the final timer (after your call to `cudaMemcopy()` that returns data to the CPU) because `cudaMemcpy()` will not return to the calling thread until after the copy is complete.

Question 1. Compare and explain the difference between the results provided by two sets of timers (timing only the kernel execution vs. timing the entire process of moving data to the GPU and back in addition to the kernel execution). Are the bandwidth values observed *roughly* consistent with the reported bandwidths available to the different components of the machine? (You should use the web to track down the memory bandwidth of an NVIDIA RTX 2000 Ada Generation GPU. Here is a hint. Several factors prevent peak bandwidth, including CPU motherboard chipset performance and whether or not the host CPU memory used as the source of the transfer is "pinned" — the latter allows the GPU to directly access memory without going through virtual memory address translation. If you are interested, you can find more info here)

4 Part 2: CUDA Warm-Up 2: Parallel Prefix-Sum

Now that you're familiar with the basic structure and layout of CUDA programs, as a second exercise you are asked to come up with parallel implementation of the function `find_repeats`, which, given a list of integers `A`, returns a list of all indices `A` for which `A[i] == A[i+1]`.

For example, given the array `{1,2,2,1,1,1,3,5,3,3}`, your program should output the array `{1,3,4,8}`.

4.1 Exclusive Prefix Sum

We want you to implement `find_repeats` by first implementing parallel exclusive prefix-sum operation.

Exlusive prefix sum takes an array `A` and produces a new array `output` that has, at each index `i`, the sum of all elements up to but not including `A[i]`. For example, given the array `A={1,4,6,8,2}`, the output of exclusive prefix sum `output={0,1,5,11,19}`.

The following "C-like" code is an iterative version of scan. In the pseudocode before, we use `parallel_for` to indicate potentially parallel loops. Here is another resource about this algorithm: http://cs149.stanford.edu/fall24/lecture/dataparallel/slide_17.

```
void exclusive_scan_iterative(int* start, int* end, int* output) {

    int N = end - start;
    memmove(output, start, N*sizeof(int));

    // upsweep phase
    for (int two_d = 1; two_d <= N/2; two_d*=2) {
        int two_dplus1 = 2*two_d;
        parallel_for (int i = 0; i < N; i += two_dplus1) {
            output[i+two_dplus1-1] += output[i+two_d-1];
        }
    }

    output[N-1] = 0;

    // downsweep phase
    for (int two_d = N/2; two_d >= 1; two_d /= 2) {
        int two_dplus1 = 2*two_d;
        parallel_for (int i = 0; i < N; i += two_dplus1) {
            int t = output[i+two_d-1];
            output[i+two_d-1] = output[i+two_dplus1-1];
            output[i+two_dplus1-1] += t;
        }
    }
}
```

We would like you to use this algorithm to implement a version of parallel prefix sum in CUDA. You must implement `exclusive_scan` function in `scan/scan.cu`. Your implementation will consist

of both host and device code. The implementation will require multiple CUDA kernel launches (one for each parallel_for loop in the pseudocode above).

Note: In the starter code, the reference solution scan implementation above assumes that the input array's length (`N`) is a power of 2. In the `cudaScan` function, we solve this problem by rounding the input array length to the next power of 2 when allocating the corresponding buffers on the GPU. However, the code only copies back `N` elements from the GPU buffer back to the CPU buffer. This fact should simplify your CUDA implementation.

Compilation produces the binary `cudaScan`. Commandline usage is as follows:

```
Usage: ./cudaScan [options]

Program Options:
  -m  --test <TYPE>      Run specified function on input.  Valid tests are: scan,
  find_repeats (default: scan)
  -i  --input <NAME>     Run test on given input type. Valid inputs are: ones, random
  (default: random)
  -n  --arraysize <INT>  Number of elements in arrays
  -t  --thrust           Use Thrust library implementation
  -?  --help             This message
```

4.2 Implementing "Find Repeats" Using Prefix Sum

Once you have written `exclusive_scan`, implement the function `find_repeats` in `scan/scan.cu`. This will involve writing more device code, in addition to one or more calls to `exclusive_scan()`. Your code should write the list of repeated elements into the provided output pointer (in device memory), and then return the size of the output list.

When calling your `exclusive_scan` implementation, remember that the contents of the `start` array are copied over to the `output` array. Also, the arrays passed to `exclusive_scan` are assumed to be in `device` memory.

4.3 Grading and Test Harness

Grading: We will test your code for correctness and performance on random input arrays.

For reference, a scan score table is provided below, showing the performance of a simple CUDA implementation on a K80 GPU. To check the correctness and performance score of your `scan` and `find_repeats` implementation, run `./checker.py scan` and `./checker.py find_repeats` respectively. Doing so will produce a reference table as shown below; your score is based solely on the performance of your code. In order to get full credit, your code must perform within 20% of the provided reference solution.

```
-------------------------
```

```
Scan Score Table:
-------------------------
-------------------------------------------------------------------------
| Element Count   | Ref Time        | Student Time    | Score           |
-------------------------------------------------------------------------
| 1000000         | 0.766           | 0.143 (F)       | 0               |
| 10000000        | 8.876           | 0.165 (F)       | 0               |
| 20000000        | 17.537          | 0.157 (F)       | 0               |
| 40000000        | 34.754          | 0.139 (F)       | 0               |
-------------------------------------------------------------------------
|                                 | Total score:    | 0/5             |
-------------------------------------------------------------------------
```

This part of the assignment is largely about getting more practice with writing CUDA and thinking in a data parallel manner, and not about performance tuning code. Getting full performance points on this part of the assignment should not require much (or really any) performance tuning, just a direct port of the algorithm pseudocode to CUDA. However, there's one trick: a naive implementation of scan might launch N CUDA threads for each iteration of the parallel loops in the pseudocode, and using conditional execution in the kernel to determine which threads actually need to do work. Such a solution will not be performant! (Consider the last outmost loop iteration of the upsweep phase, where only two threads would do work!). A full credit solution will only launch one CUDA thread for each iteration of the innermost parallel loops.

Test Harness: By default, the test harness runs on a pseudo-randomly generated array that is the same every time the program is run, in order to aid in debugging. You can pass the argument `-i random` to run on a random array - we will do this when grading. We encourage you to come up with alternate inputs to your program to help you evaluate it. You can also use the `-n <size>` option to change the length of the input array.

The argument `-thrust` will use the [Thrust Library's] implementation of [exclusive scan]. **Up to two points of extra credit for anyone that can create an implementation is competitive with Thrust.**

5 Part 3: A Simple Circle Renderer

Now for the real show!

The directory `/render` of the assignment starter code contains an implementation of renderer that draws colored circles. Build the code, and run the render with the following command line: `./render -r cpuref rgb`. The program will output an image `output_0000.ppm` containing three circles. Now run the renderer with the command line `./render -r cpuref snow`. Now the output image will be falling snow. PPM images can be viewed directly on OSX via preview. For windows you might need to download a viewer.

Note: you can also use the `-i` option to send renderer output to the display instead of a file.

(In the case of snow, you'll see an animation of falling snow.) However, to use interactive mode you'll need to be able to setup X-windows forwarding to your local machine. ([This reference] or [this reference] may help.)

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`.

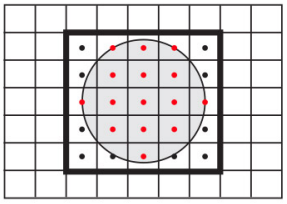
5.1 Renderer Overview

We encourage you to familiarize yourself with the structure of the renderer codebase 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). `render` is called each frame and is responsible for drawing all circles into 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
```

The figure below illustrates the basic algorithm for computing circle-pixel coverage using point-in-circle tests. Notice that a circle contributes color to an output pixel only if the pixel's center lies within the circle.

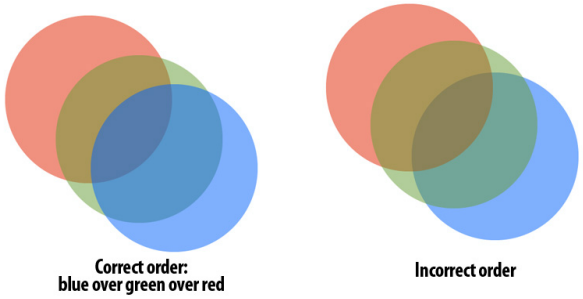


An important detail of the renderer is that it renders **semi-transparent** circles. Therefore, the color of any one pixel is not the color of a single circle, but the result of blending the contributions

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) values (RGBA). Alpha = 1 corresponds to a fully opaque circle. Alpha = 0 corresponds to a fully transparent circle. To draw a semi-transparent circle with color `(C_r, C_g, C_b, C_alpha)` on top of a pixel with color `(P_r, P_g, P_b)`, the renderer uses the following math:

```
result_r = C_alpha * C_r + (1.0 - C_alpha) * P_r
result_g = C_alpha * C_g + (1.0 - C_alpha) * P_g
result_b = C_alpha * C_b + (1.0 - C_alpha) * 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 that the render draw 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 below where a blue circle is drawn OVER a green circle which is drawn OVER a red circle. In the image on the left, the circles are drawn into the output image in the correct order. In the image on the right, the circles are drawn in a different order, and the output image does not look correct.



5.2 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 `-renderer cuda (or -r cuda)` cuda program option.

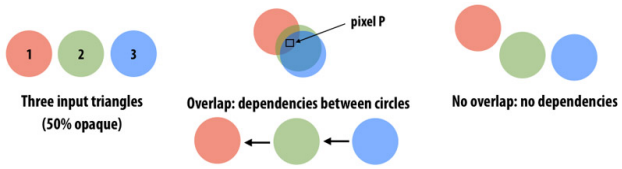
The provided CUDA implementation parallelizes computation across all input circles, assigning one circle to each CUDA thread. While this CUDA implementation is a complete implementation of the mathematics of a circle renderer, it contains several major errors that you will fix in this assignment. Specifically: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

5.3 Renderer Requirements

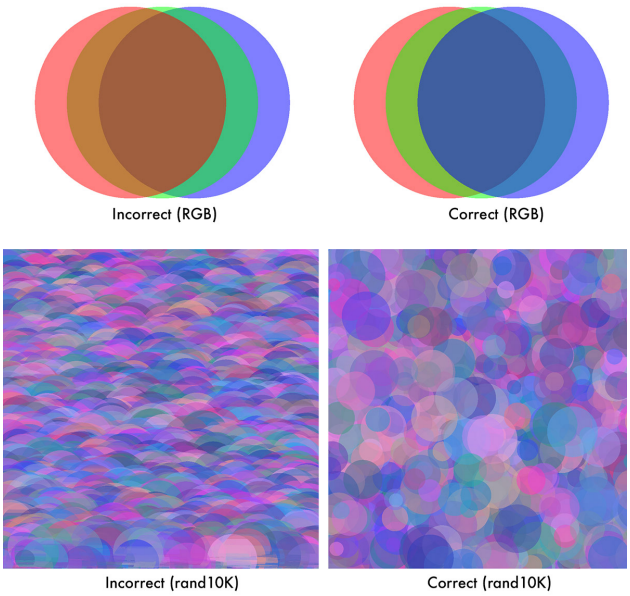
Your parallel CUDA renderer implementation must maintain two invariants that are preserved trivially in the sequential implementation.

1. **Atomicity:** All image update operations must be atomic. The critical region includes reading the four 32-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 correct rendering of transparent circles. (It has a number of other benefits for graphics systems. If curious, talk to Kayvon.) **A key observation is that the definition of order only specifies the order of updates to the same pixel.** Thus, as shown below, there are no ordering requirements 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 rgb and circles scenes. You will see horizontal streaks through the resulting images, as shown below. These streaks will change with each frame.



5.4 What You Need To Do

Your job is to write the fastest, correct CUDA renderer implementation you can. You may take any approach you see fit, but your renderer must adhere to the atomicity and order requirements specified above. A solution that does not meet both requirements will be given no more than 12 points on part 3 of the assignment. We have already given you such a solution!

A good place to start would be to read through `cudaRenderer.cu` and convince yourself that it *does not* meet the correctness requirement. In particular, look at how `CudaRenderer:render` launches the CUDA kernel `kernelRenderCircles`. (`kernelRenderCircles` is where all the work happens.) To visually see the effect of violation of above two requirements, compile the program with `make`. Then run `./render -r cuda rand10k` which should display the image with 10K circles, shown in the bottom row of the image above. Compare this (incorrect) image with the image generated by sequential code by running `./render -r cpuref rand10k`.

We recommend that you:

1. First rewrite the CUDA starter code implementation so that it is logically correct when running in parallel (we recommend an approach that does not require locks or synchronization)
2. Then determine what performance problem is with your solution.
3. At this point the real thinking on the assignment begins... (Hint: the circle-intersects-box tests provided to you in `circleBoxTest.cu_inl` are your friend. You are encouraged to use these subroutines.)

Following are commandline options to `./render`:

```
Usage: ./render [options] scenename
Valid scenenames are: rgb, rgby, rand10k, rand100k, rand1M, biglittle, littlebig,
pattern, micro2M, bouncingballs, fireworks, hypnosis, snow, snowsingle
Program Options:
  -r  --renderer <cpuref/cuda>  Select renderer: ref or cuda (default=cuda)
  -s  --size  <INT>             Rendered image size: <INT>x<INT> pixels (default=1024)
  -b  --bench <START:END>       Run for frames [START,END) (default=[0,1))
  -c  --check                   Check correctness of CUDA output against CPU reference
  -i  --interactive             Render output to interactive display
  -f  --file  <FILENAME>        Output file name (FILENAME_xxxx.ppm) (default=output)
  -?  --help                    This message
```

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.

We provide a total of eight circle datasets you will be graded on. However, in order to receive full credit, your code must pass all of our correctness-tests. To check the correctness and performance score of your code, run `./checker.py` (notice the .py extension) in the `/render` directory. If you run it on the starter code, the program will print a table like the following, along with the results of our entire test set:

```
Score table:
------------
-------------------------------------------------------------------------
| Scene Name      | Ref Time (T_ref) | Your Time (T)   | Score           |
-------------------------------------------------------------------------
| rgb             | 0.2698           | (F)             | 0               |
| rand10k         | 2.7341           | (F)             | 0               |
| rand100k        | 26.1481          | (F)             | 0               |
| pattern         | 0.3591           | (F)             | 0               |
| snowsingle      | 16.1636          | (F)             | 0               |
| biglittle       | 14.9861          | (F)             | 0               |
| rand1M          | 188.0086         | (F)             | 0               |
| micro2M         | 355.9104         | (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 sometimes it might be correct. This doesn't change the fact that the current CUDA renderer is in general incorrect.

"Ref time" is the performance of our reference solution on your current machine (in the provided

`render_ref` executable). "Your time" is the performance of your current CUDA renderer solution, where an `(F)` indicates an incorrect solution. Your grade will depend on the performance of your implementation compared to these reference implementations (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 your name and Netid at 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 maybe 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 bandwidth 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?

5.5 Grading Guidelines

The write-up for the assignment is worth 7 points.

- Your implementation is worth 72 points. These are equally divided into 9 points per scene as follows:
  - 2 correctness points per scene.
  - 7 performance points per scene (only obtainable if the solution is correct). Your performance will be graded with respect to the performance of a provided benchmark reference renderer, T_{ref}:
  - No performance points will be given for solutions having time (T) 10 times the magnitude of T_{ref}.
  - Full performance points will be given for solutions within 20% of the optimized solution (T <= 1.20T_{ref})
  - For other values of T (for 1.20 T_{ref} < T < 10 _ T_{ref}), your performance score on a scale 1 to 7 will be calculated as: '7 _ T_ref / T'.

- Up to 5 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 5 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. Feel free to use any tools at your disposal (e.g., SIMD intrinsics, ISPC, pthreads). 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.

So the total points for this project is as follows:

- part 1 (5 points)
- part 2 (10 points)
- part 3 write up (13 points)
- part 3 implementation (72 points)
- potential **extra** credit (up to 10 points)

6 Tips and Hints

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

- 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 circles). Solutions will need to exploit both types of parallelism, potentially at different parts of the computation.

- The circle-intersects-box tests provided to you in `circleBoxTest.cu_inl` are your friend. You are encouraged to use these subroutines.

- The shared-memory prefix-sum operation provided in `exclusiveScan.cu_inl` may be valuable to you on this assignment (not all solutions may choose to use it). See the simple description of a prefix-sum [here]. We have provided an implementation of an exclusive prefix-sum on a **power-of-two-sized** arrays in shared memory. **The provided code does not work on non-power-of-two inputs and IT ALSO REQUIRES THAT THE NUMBER OF THREADS IN THE THREAD BLOCK BE THE SIZE OF THE ARRAY. PLEASE READ THE COMMENTS IN THE CODE.**

- Take a look at the `shadePixel` method that is being called. Notice how it is doing many global memory operations to update the color of a pixel. It might be wise to instead use a local accumulator in your `kernelRenderCircles` method. You can then perform the accumulation of a pixel value in a register, and once the final pixel value is accumulated you can then just perform a single write to global memory.

- You are allowed to use the [Thrust library] in your implementation if you so choose. Thrust is not necessary to achieve the performance of the optimized CUDA reference implementations. There is one popular way of solving the problem that uses the shared memory prefix-sum implementation that we give you. There another popular way that uses the prefix-sum routines in the Thrust library. Both are valid solution strategies.

- Is there data reuse in the renderer? What can be done to exploit this reuse?

- 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.**

- For the tests which contain a larger number of circles - `rand1M` and `micro2M` - you should be careful about allocating temporary structures in global memory. If you allocate too much global memory, you will have used up all the memory on the device. If you are not checking the `cudaError_t` value that is returned from a call to `cudaMalloc`, then the program will still execute but you will not know that you ran out of device memory. Instead, you will fail the correctness check because you were not able to make your temporary structures. This is why we suggest you to use the CUDA API call wrapper below so you can wrap your `cudaMalloc` calls and produce an error when you run out of device memory.

- If you find yourself with free time, have fun making your own scenes!

6.1 Catching CUDA Errors

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

```
...
line 742: cudaMalloc(&a, -1); // executes, then continues
line 743: cudaCheckError(cudaMemcpy(a,b)); // prints "CUDA Error: out of memory at
cudaRenderer.cu:743"
...
```

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

(Credit: adapted from [this Stack Overflow post]

7 Hand-in Instructions

Please submit your work using Gradescope.

1. **Please submit your writeup as the file `writeup.pdf`.**
2. **Please submit run `sh create_submission.sh` to generate a zip to submit to gradescope**. Note that this will run make clean in your code directories so you will have to run make again to run your code. If the script errors saying `Permission denied`, you should run `chmod +x create_submission.sh` and then try rerunning the script.

Our grading scripts will rerun the checker code allowing us to verify your score matches what you submitted in the `writeup.pdf`. We might also try to run your code on other datasets to further examine its correctness.