TÉCNICO LISBOA

Assignment 2

High Performance Computing Systems

Sistemas de Computação de Elevado Desempenho (SCED)

2020/2021

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!

Deadlines and Submission

You will submit the work developed for Lab2 via Fenix (until 8th of November at 23h59, i.e., Sunday).

The submission should be made in a single `zip` file with the following content:

- Your report (writeup) in a file called `report.pdf` (it must be pdf)
- Your implementation (codes) for all programs (to keep submission small, please do a `make clean` in program directories prior to creating the archive, and remove any residual output images, etc.)

When handed in, all codes must be compilable and runnable out of the box on the `adriana` machine!

Your report should have at most 10 pages and should be well structured. For each part (problem) you should: elaborate your solution, explain your rationale (how did you arrive there, why it makes sense, describe all (if any) specific measurements/experiments that were conducted by you to prove your hypothesis etc). You should also discuss and analyze the obtained results!

Environment Setup

For this assignment, you will need to run (the final version of) your code on `adriana` machine (hostname: `adriana.inesc-id.pt`). This machine contains NVIDIA K40c GPU with the following characteristics:

```
Device 0: "Tesla K40c"
  CUDA Driver Version / Runtime Version          10.1 / 10.1
  CUDA Capability Major/Minor version number:    3.5
  Total amount of global memory:                 11441 MBytes (11996954624 bytes)
  (15) Multiprocessors, (192) CUDA Cores/MP:     2880 CUDA Cores
  GPU Max Clock rate:                            745 MHz (0.75 GHz)
  Memory Clock rate:                             3004 Mhz
  Memory Bus Width:                              384-bit
  L2 Cache Size:                                 1572864 bytes
  Total amount of shared memory per block:       49152 bytes
  Total number of registers available per block: 65536
  Warp size:                                     32
  Maximum number of threads per multiprocessor:  2048
  Maximum number of threads per block:           1024
  Max dimension size of a thread block (x,y,z): (1024, 1024, 64)
  Max dimension size of a grid size    (x,y,z): (2147483647, 65535, 65535)
```

The CUDA C programmer's guide PDF version or web version is an excellent reference for learning how to program in CUDA (we can also provide you some CUDA SDK samples on `adriana`, if you think that you need more codes to practice). In addition, there are a wealth of CUDA tutorials and SDK examples on the web (just Google!) and on the NVIDIA developer site.

For C++ questions (like what does the *virtual* 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++!

The access to `adriana` machine is performed through `ssh`, by using the command:

`ssh HPCS_g<group_number>@adriana.inesc-id.pt`

where you should substitute `<group_number>` with your group number (e.g., the username for the group number 3 is `HPCS_g3`). The passwords for each group have the flowing format `g<group_number>hpcs` (again, the password for the group number 3 is `g3hpcs`).

IMPORTANT: Since the machine is a shared resource, please verify if nobody else is performing experiments before running your program, in order not to affect the results of other groups (or even other researchers from INESC-ID). To check the users logged in the machine, you can use the **`who`** command. It is highly recommended to schedule the utilization of `adriana` among yourselves to avoid the simultaneous use of the machine.

Note: For grading purposes, we expect you to report on the performance of code run on the `adriana` machine. However, for development, you may also want to run the programs in this assignment on your own machine.

To get started:

1. The assignment starter code is available on the course webpage (section: Labs). `wget` it, `unzip` it and start having fun with it.

Part 1: CUDA Warm-Up 1: SAXPY

To gain a bit of practice writing CUDA programs your warm-up task is to re-implement the SAXPY function from Assignment 1 in CUDA. Starter code for this part of the assignment is located in the `/saxpy` directory of the assignment tarball.

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 Programmer's Guide 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 to the CPU.)

When adding your timing code in the latter case, you'll need to be careful: By default, 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.

```
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.

Questions:

1. What performance do you observe compared to the sequential CPU-based implementation of SAXPY (recall your results from `saxpy` on Program 5 from Assignment 1)?
2. 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? (Hint: You should use the web to track down the memory bandwidth of an NVIDIA K40 GPU, and the maximum transfer speed of the computer's PCIe-x16 bus. It's PCIe 3.0, and a 16-lane bus connecting the CPU with the GPU.)

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 `i` 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}`.

Exclusive Prefix Sum

We want you to implement `find_repeats` by first implementing parallel exclusive prefix-sum operation. Exclusive 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 code is a recursive C-code implementation of a work-efficient, parallel implementation of scan. **Note: Some of you may wish to skip the following recursive implementation and jump to the iterative version below.**

```
void exclusive_scan_recursive(int* start, int* end, int* output, int* scratch) {

    int N = end - start;

    if (N == 0)
        return;
    else if (N == 1) {
        output[0] = 0;
        return;
    }

    // sum pairs in parallel.
    for (int i = 0; i < N/2; i++)
       output[i] = start[2*i] + start[2*i+1];

    // prefix sum on the compacted array.
    exclusive_scan_recursive(output, output + N/2, scratch, scratch + (N/2));

    // finally, update the odd values in parallel.
    for (int i = 0; i < N; i++) {
        output[i] = scratch[i/2];
        if (i % 2)
            output[i] += start[i-1];
    }
}
```

While the above code expresses our intent well and recursion is in fact supported on modern GPUs, its use can lead to fairly low performance. Instead, we can express the algorithm in an iterative manner. The following "C-like" code is an iterative version of scan. In the pseudocode before, we use `parallel_for` to indicate potentially parallel loops.

```
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) and you are not allowed to merge `upsweep` and `downsweep` phases.

Note: In the starter code, the `cudaScan` function does not assume that the input array's length (`N`) is a power of 2. 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`. Command line 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
```

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 indices 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.

Grading: We will test your code for correctness and performance on random input arrays. Consider that your implementation must correctly process big arrays, e.g., array with 1M, 10M, 20M and 40M elements! Please consult `Ref_Timings.txt` file for reference timings (you are qualified for a maximum number of points if your execution time is around 20% of the reference values).

Important note: 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 naïve 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.

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.

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`.

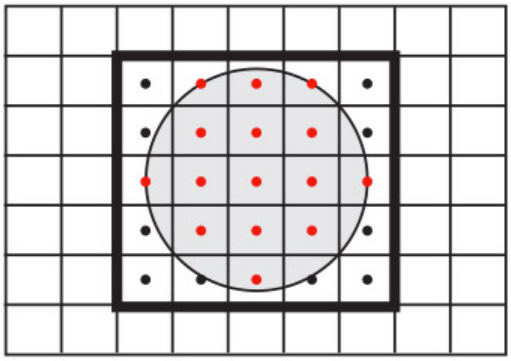
Renderer Overview

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
```

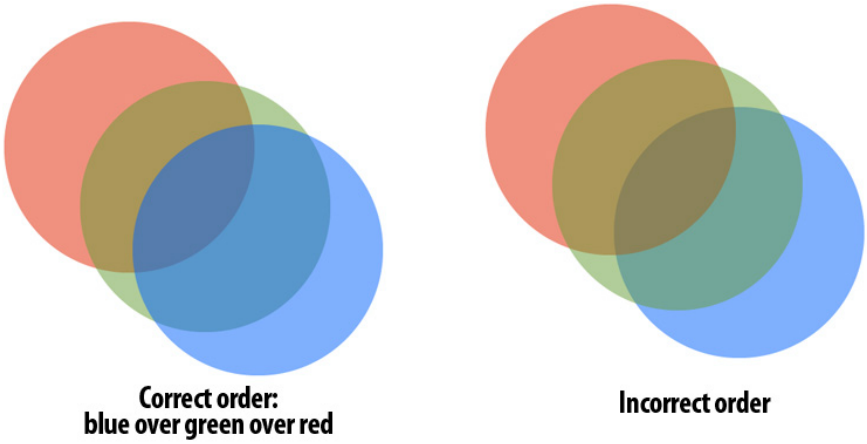
The figure below illustrates the basic algorithm for computing circle-pixel coverage using point-in-circle tests. Notice that a circle contributes color to an output pixel only if the pixel's center lies within the circle.



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 is 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.



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.

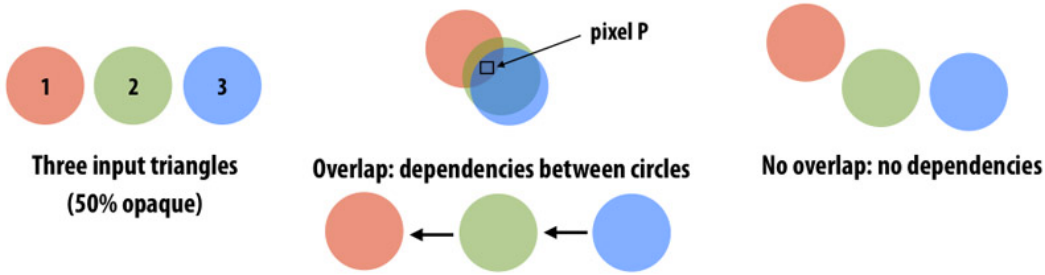
The provided CUDA implementation parallelizes computation across all input circles, assigning one circle to each CUDA thread. While this CUDA implementation is a complete implementation of the mathematics of a circle renderer, it contains several major errors that you will fix in this assignment. Specifically: 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).

Renderer Requirements

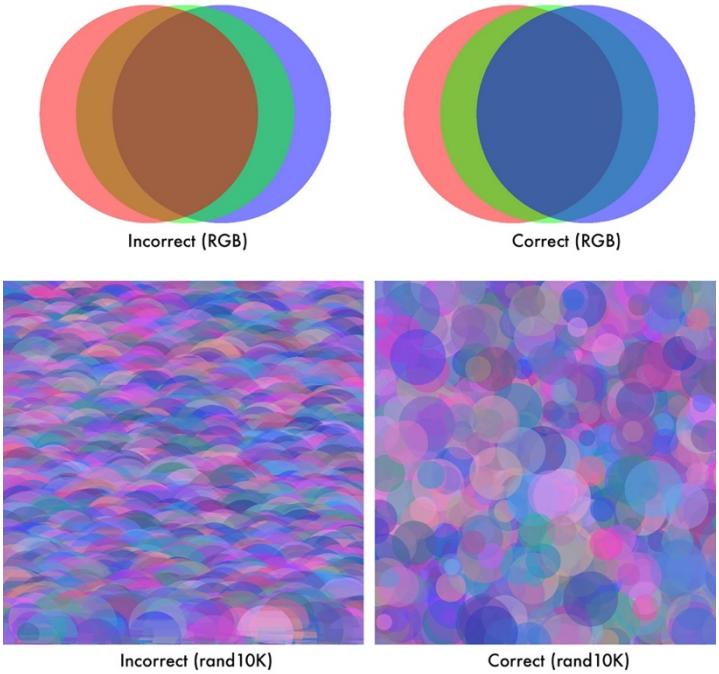
Your parallel CUDA renderer implementation must maintain two invariants that are preserved trivially in the sequential implementation.

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. **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.



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 less than 10% of the points for this part 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 command line options to `./render`:

```
Usage: ./render [options] scenename
Valid scenenames are: rgb, rgby, rand10k, rand100k, biglittle, littlebig, pattern,
                      bouncingballs, fireworks, hypnosis, snow, snowsingle
Program Options:
  -r  --renderer <cpuref/cuda> Select renderer: ref or cuda (default=cuda)
  -s  --size  <INT>            Make rendered image <INT>x<INT> pixels (default=1024)
  -b  --bench <START:END>      Run for frames [START,END)   (default [0,1))
  -f  --file  <FILENAME>       Output file name (FILENAME_xxxx.ppm)
  -c  --check                 Check correctness of CUDA output against CPU reference
  -i  --interactive            Render output to interactive display
  -?  --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 five circle datasets you will be graded on (`rgb`, `rand10k`, `rand100k`, `pattern`, `snowsingle` and `biglittle`). However, in order to receive full credit, your code must pass all of our correctness-tests, i.e., scenes: `rgb`, `rgby`, `rand10k`, `rand100k`, `pattern`, `snowsingle`, `snow`, `biglittle`, `littlebig`, `bouncingballs`, `hypnosis` and `fireworks`. Please consult `Ref_Timings.txt` file for reference timings (you are qualified for a maximum number of points if your execution time is around 20% of the reference values).

Your grade will depend on the performance of your implementation. 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 report include:

1. Describe how you decomposed the problem and how you assigned work to CUDA thread blocks and threads (and maybe even warps).
2. Describe where synchronization occurs in your solution.
3. What, if any, steps did you take to reduce communication requirements (e.g., synchronization or main memory bandwidth requirements)?
4. Briefly describe how you arrived at your final solution. What other approaches did you try along the way. What was wrong with them?

Assignment Tips and Hints

Below are a set of tips and hints. 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. The provided implementation of an exclusive prefix-sum works only 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.** You are also 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.
- 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.**

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])