15-418: Assignment 2:
A Simple CUDA Renderer

Due: Tuesday Feb 21, 11:59PM

1 Overview

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

2 Environment Setup

You are strongly encouraged to develop and test your code on the machines in GHC 5205 for this assignment. Each of these machines contains a NVIDIA GTX 480 GPU (CUDA compute capability 2.0). In CUDA speak, these GPUs contain 15 multi-threaded "SMs" (think of an SM as a core). The capabilities of this chip can be found in Appendix F.4 of the CUDA C Programming Guide. Table F.1 in the Programmers Guide is a very handy reference for the maximum number of CUDA threads per thread block, size of thread block shared memory, and additional details about this chip.

The CUDA C programmer's guide (version 4.0) can be found at the URL below. This is an excellent reference for learning how to program in CUDA.

`http://developer.download.nvidia.com/compute/cuda/4_0/toolkit/docs/CUDA_C_Programming_Guide.pdf`

You can also find a large number of examples in `/usr/local/cuda/C/src`. In addition, there are a wealth of CUDA tutorials and SDK examples on the web:

`http://developer.nvidia.com/cuda-libraries-sdk-code-samples`

To get started:

1. The NVIDIA CUDA C/C++ Compiler (NVCC) will be used to compile CUDA code. It is located at `/usr/local/cuda/bin/`, which you need to add to your PATH.
2. The CUDA shared library will be loaded at runtime. It is located at `/usr/local/cuda/lib64/`, which you need to add to your LD_LIBRARY_PATH.
3. Download the Assignment 2 starter code from the course directory:
   `/afs/cs.cmu.edu/academic/class/15418-s12/assignments/asst2.tgz`

3 Assignment

3.1 Part 1: CUDA Warm-up (5 pts)

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. Conversely, 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.

As part of your implementation, add timers around the CUDA kernel invocation in `saxpyCuda` . Your additional timing measurement *should not* include the time to transfer data to and from device memory (just the time to execute the computation). Note that the call to `cudaThreadSynchronize` following the kernel call waits for completion of all CUDA work since the kernel launch is asynchronous with the main application thread.

Question 1. What performance do you observe compared to the sequential CPU-based implementation of SAXPY (recall program 3 from Assignment 1)? Compare and explain the difference between the results provided by two sets of timers (the timer you added and the timer that was already in the provided starter code). Are the bandwidth values observed consistent with the peak bandwidths available to the different components of the machine?

3.2 Part 2: A Simple Circle Renderer (45 pts)

The directory `/render` of the assignment tarball contains an implementation of renderer that draws colored circles. Build the code, and run the render with the following 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 a parallel CUDA implementation in `cudaRenderer.cpp`.

3.2.1 Renderer Overview

We encourage you to familiarize yourself with the renderer 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 responsible for drawing all circles into the output image. The other main function of the renderer, `advanceParticles`, is also invoked once per frame. It updates circle positions and velocities. You will not need to modify `advanceParticles` in this assignment.

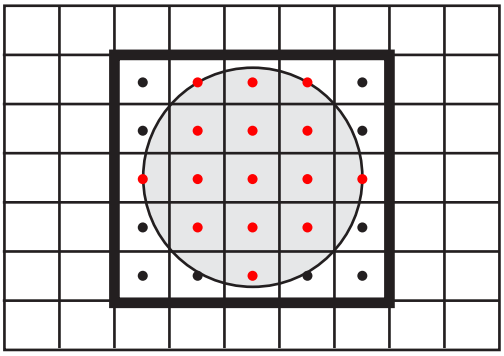

Figure 1: All pixels within the circle's conservative bounding box are tested for coverage. For each pixel in the bounding box, the pixel is classified as covered by the circle if its center point (black dots) is contained within the circle. Pixels centers that are inside the circle are colored red. The circle's contribution to the image will be computed at these pixels.

The renderer accepts an array of circles (3D position, velocity, radius, color) as input. The basic sequential algorithm for rendering circles 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 1 illustrates how the renderer computes circle-pixel coverage using point-in-circle tests.

3.2.2 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` 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 (see below).

3.2.3 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 consists of reading of four 32-bit floating-point values (the pixel's 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 input triangle order. That is, if triangle 1 and triangle 2 both contribute to pixel P, the update due to triangle 1 must be applied to the image before the update due to triangle 2. As discussed in class, preserving the ordering requirement allows for correct rendering of transparent surfaces. (It has a number of other benefits for graphics systems. If curious, please talk to Kayvon.) Note there are no ordering requirements between circles that do not contribute to the same pixel. These circles can be processed independently.

The result of not correctly respecting order can be seen by running the CUDA renderer implementation on the `rgb` and `circles` scenes. You will see horizontal streaks through the resulting images. These streaks will change with each frame.

3.2.4 What You Need To Do

Your job is to write the fastest 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 in Section 3.2.3. *A solution that does not meet both requirements will be given no more than 10 points on part 2 of the assignment.* We have already given you such a solution!

The following table summarizes the performance of one possible solution running on the machines in GHC 5205. Your code must produce correct output, should achieve performance that is within 20% of the speed of this implementation on all scenes to receive full credit for the programming part of the assignment. By correct output, we require you to respect the two renderer requirements given in Section 3.2.3. Due no worry if your images differ from those of the reference implementation by a few pixels or so (such differences are likely due to floating point error in pixel-circle overlap tests).

The times reported below are per-frame timings for just the call to `render()`. The timing was performed using the `--bench 0:4` flag. All values are in milliseconds.

	image size: 512x512		image size: 1024x1024	
	ref	cuda (speedup)	ref	cuda (speedup)
rgb	1.94	0.13 (14.9x)	8.02	0.49 (16.4x)
rgby	1.05	0.12 (8.8x)	4.31	0.46 (9.4x)
pattern	4.32	0.49 (8.8x)	18.86	1.76 (10.7x)
rand10k	208.40	5.86 (35.6x)	882.75	21.26 (41.5x)
rand100k	2084.03	60.47 (41.3x)	8860.17	217.72 (40.7x)
snowsingle	255.55	29.72 (8.6x)	1006.35	113.96 (8.8x)

Along with your code, we would like you to hand in a clear description of your implementation as well as a brief description of how you arrived at this solution (what other approaches did you try along the way?). Aspects of your work that you should mention in the write-up include:

1. Include both partners names and andrew id's at the top of your write-up.
2. Replicate the above table for your solution.

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?

3.2.5 Grading Guidelines

- The write-up for part 2 is worth 10 points
- Your implementation is worth 35 points:
  - 10 points for a correct solution (meets functional requirements)
  - 35 points for a solution that meets the performance requirements
  - Up to 5 points extra credit (instructor discretion) for solutions that achieve *significantly* greater performance than required. Your write-up must explain your approach thoroughly.

3.3 Tips and Hints

- To facilitate remote development and benchmarking, we've created a `--benchmark` option to the render program. This mode does not open a display, and instead runs the renderer for the specified number of frames.
- When in benchmark mode, the `--file` option sets the base file name for PPM images created at each frame. Created files are basename_xxxx.ppm. No PPM files are created if the `--file` option is not used.
- There are two major axes of parallelism in this assignment. There is parallelism across pixels (all pixels can be processed in parallel) and there is parallelism across circles (providing the ordering requirement is respected for overlapping circles).
- The prefix-sum operation may be valuable to you on this assignment. See the simple description of a prefix-sum here: `http://code.google.com/p/thrust/wiki/QuickStartGuide#Prefix-Sums`. We have provided an implementation of an exlcusive prefix-sum on an array in shared memory.
- Is there significant data reuse in the renderer? What can be done to exploit this reuse?
- How are you going to maintain atomicity of image update since there is no language primitive that performs the logic of the image update operation atomically on the GTX 480? Creating a lock out of the available atomics is one solution. Are there lockless solutions for image update.
- If you find yourself with free time, have fun making your own scenes. A fireworks demo is just asking to be made!

3.4 Hand-in Instructions

Our grading scripts will use the benchmark mode of the program to time your performance, and the `--file` options to dump frames for correctness testing. (Please to not change the contents of `benchmark.cpp`). Grading runs will be performed at 512x512 and/or 1024x1024 resolution.

Hand-in directories have been created at:
`/afs/cs.cmu.edu/academic/class/15418-s12/handin/asst2/<ANDREW ID>/`.

Copy your `asst2/` directory into your hand-in directory. (To keep sizes small, please do a `make clean` prior to creating the archive). The written answers should be in `asst2/writeup.pdf`. **When handed in, all code must be compilable and runnable!** We should be able to `make` and execute your programs without manual intervention.