Lab #1 for CPE 458 & CPE 473

Ray-tracing with CUDA

Due Thursday, April 28th, 2011

1. Orientation

This is a team based lab. Please plan to work in the teams specified by your instructors. Each team should include a student from CPE 473 and students from CPE 458 and you are expected to help one another with understanding ray-tracing and CUDA. You will be required to hand in your lab and demonstrate specific profiling reports to be specified later. Your second joint lab will build on this code, so please plan accordingly.

1. Introduction

The following project will help acquaint 473 students to the power of today's GPUs and the versatility of the CUDA framework. 458 students should view this as an exercise in applying CUDA in a real-world application. Ray-tracing, at least at a high level, is embarrassingly parallel and can easily be modified to run on dozens, hundreds, or thousands of cores at once. We hope that through this project you will gain a firm grasp of the power of CUDA and the interesting problem sets that it can be applied to.

2. Profiling your ray-tracer

You may discuss using gprof in class. There are also many very useful resources online for getting started (we suggest checking out some of the first results of "using gprof" on Google.) You will be given some large scenes to run against, and you will find that even a couple thousand spheres will bring your current implementation to a crawl.

Please Note:
If your gprof results aren't pointing towards 50-70% of your program run time in intersections and/or notice that your data accessors are a huge amount of your time, you should inline all of your vector accessor and operations. You will notice a big difference in your profiling results.

3. Preparing your code for the GPU

1. Split planes into separate array (this will be useful later both in the lab and in future work)
2. Separate intersection testing for both arrays
3. Note that you will be moving all intersection tests for spheres onto the GPU, so you should strive to have one function that JUST does those intersections if you do not already have that

4. Outsourcing intersection tests to the GPU

Structures Required
You are going to need to create the following C-style structs in order to facilitate the movement of the spheres, rays and intersections onto the GPU:

- `cuda_sphere_t` -- includes the location and radius of the sphere in the scene. No finish or pigment information is necessary as shading will be done on the host machine.
- `cuda_ray_t` -- contains the start and direction vectors for a given ray
- `cuda_intersection_t` -- contains the index of the object it hit and the *t* value of the given object

Important -- In order to avoid indexing issues later on, please be aware of the following paradigms:

- Keep the spheres described in the `cuda_sphere_t` array *in the same order* as the spheres on the host machine. This will allow the GPU to easily identify what sphere a given ray intersected with by returning it's index in the array. If these differ, then the intersection information you return will be incorrect.
- Similar to sphere indexing, please keep the array of `cuda_ray_t`'s in the same order as you generate them on the host. When you return the list of `cuda_intersection_t`'s, you will be able to map them index-by-index to the pixels that they relate to (remember that each `cuda_ray_t` sent to the GPU must generate a `cuda_intersection_t` that corresponds with what it hit.)

4.1 Steps

These steps are broken down into four spots in your ray-tracing process: one after the scene has been constructed, one after all rays have been constructed, one during the intersection testing phase and one after all intersections have been computed on the GPU

4.1.1 After Scene Construction

1. Generate from your scene an array of `cuda_sphere_t`'s on the host machine.
2. Allocate enough space for every `cuda_sphere_t` on the GPU using `cudaMalloc`
3. Use `cudaMemCpy` to copy the `cuda_sphere_t` memory onto the GPU

4.1.2 Before Intersection Tests

1. Before the rays are cast off, allocate enough space for `cuda_ray_t` and `cuda_intersection_t arrays` (WIDTH x HEIGHT)
2. Iterate through every pixel, creating a `cuda_ray_t` array on the host machine which maps a ray for every pixel
3. Use `cudaMemCpy` to copy the rays onto the GPU. The intersections will be constructed on the GPU itself

4.1.3 During Intersection Tests

1. Move all sphere intersection tests into a cuda-callable function utilizing the `cuda_sphere_t` and `cuda_ray_t` array pointers. Each intersection test should generate a `cuda_intersect_t` result which is placed into the resulting array.

4.1.4 After Intersection Tests

1. Use `cudaMemCpy` (but indicate device to host) to copy all intersections back onto the host machine. At this point you a list of intersections indexed by each pixel of the image.
2. Loop through each pixel and compare the intersection *t* values with those gathered from the plane intersections that were computed on the host machine itself.

4.2 Suggestions for First Steps

This is not an all-or-nothing process. These steps can be done iteratively, and this is how we suggest you approach this problem. An example of how to cut this workload into manageable chunks would be to do the following:

1. **Separate the sphere and plane arrays on the host machine.** This will be very useful for later projects (such as BVH construction) as well as the current CUDA project.
2. **Create all cuda-related structures.** These include the `cuda_sphere_t`, `cuda_ray_t`, and `cuda_intersection_t` structures.
3. **Separate sphere and plane intersection testing into two loops.** Keep in mind the sphere intersection loop will be *completely* moved onto the GPU, but we do not suggest entirely removing it from your host machine code (simply comment it out or don't call the function.) For reasons explained later, many 473 students will not continue on their project using the CUDA project code.

4. **Compile and test CUDA capabilities before diving in.** Utilize `cudaMalloc`

and `cudaMemCpy` and send back and forth test data to the GPU to make sure the communication back and forth is correctly implemented. Make sure this is done in your ray-tracing code, so that you are sure the project compiles, builds, and operates as expected.

5. Compiling the results

For those of you who wish to avoid strapping on additional (and often confusing) cuda compilation arguments and commands into your makefile, you are encouraged to take advantage of an awesome CUDA build program developed by Bob, one of the TAs, called cubuild. The program can be downloaded from github here: https://github.com/bobsomers/cubuild

If you have any questions concerning this build program, don't hesitate to ask (or check the included documentation.)

Please Note:
Cubuild requires git to be installed in order to compile. If you wish to download the application straight, click the "Download" button on the github project website and download the given cubuild-0.1.tar.gz file (do not download the source directly using the Download link.) You may also clone the repository if you are git-savvy.

6. Measuring your speedup

We suggest that you re-build your program using gprof and look at where the program is now spending most of its time. For small scenes including only a few spheres you may not notice a speedup (in fact it may be slower,) but you will be given much larger scenes to test against. This will lead to better profiling results.