Programming Assignment 3

Brian Jackson (bjack205)
CS 149 Fall 2019

Part 1: Saxpy

Question 1

On the Myth machine I got a timing result of 12 ms for saxpy implemented with ISPC
The CUDA version takes 7 ms for the kernel, but a total of 170 ms for entire program, including the data movement from the CPU to GPU.

Question 2

In this case, it is obvious that the cost of the data movement is definitely too large for the amount of computation being done on the GPU, so this calculation doesn't benefit much from parallelizing on a GPU.

The NVIDIA K80 has a bandwidth of 480 GB/s. PCIe 3.0 delivers about 985 MB/s per lane, so with 16 lanes gives about 15 GB/s. The test was calculating about 6.5 GB/s.

Part 2: Find Repeats

My implementation of scan works correctly for the size checked by the default binary. However, it fails correctness on the sizes used by checker. After talking to the TA, I'm still not sure why this is happening, and is very hard to debug, so I left it as-is.

My implementation of find-repeats only returns the total number repeats. I didn't take the time to implement the functionality to return the list.

Part 3: Renderer

My final implementation of the renderer was very competitive with the reference solution. All my solutions returned in less than 0.2 seconds, and were all correct (see table below).

```
------------
Score table:
------------
-------------------------------------------------------------------------
| Scene Name      | Ref Time (T_ref) | Your Time (T)   | Score           |
-------------------------------------------------------------------------
| rgb             | 0.5208           | 0.1861          | 12              |
| rand10k         | 11.1218          | 0.1706          | 12              |
| rand100k        | 110.6148         | 0.1702          | 12              |
| pattern         | 1.1900           | 0.1691          | 12              |
| snowsingle      | 68.7228          | 0.1690          | 12              |
| biglittle       | 74.9662          | 0.1668          | 12              |
-------------------------------------------------------------------------
|                                  | Total score:    | 72/72           |
-------------------------------------------------------------------------
```

My implementation was actually surprisingly simple. I didn't change anything to the "advanceAnimation" function, it worked correctly and I didn't see anything in particular that would gain significant performance gains.

To shade the image, I ran a single kernel that parallelized the computation over all the pixels in the image. This seemed to be the most natural way to parallelize the task since each pixel is independent from the others and there are a lot of them. To ensure correctness, I had each pixel loop through all of the circles and immediately check if it lies within the circle. This ends up being a pretty cheap check, only a handful of floating point operations per circle. If the pixel was inside of the circle, I computed the color of the pixel and passed it off to a simpler version of the "shadePixel" device function.

This implementation ensures correct computation since it serially processes each circle, and doesn't require any synchronization, since each computation is independent of the other pixels and only runs a single kernel for the render. It also avoids calculating any bounding boxes, which saves computation.

The communication for this implementation is very low, since everything happens on the device. The only data each pixel really needs is the location and radius of each circle, along with the current image values.

I tried a couple previous other solutions. My first iteration was to simply parallelize the rendering of all pixels in the bounding box around a given circle. I adjusted the grid size to accommodate bounding boxes of different sizes. Each circle was processed serially in a loop on the CPU. This resulted in a correct solution, but was fairly slow. I toyed around with several ideas, mostly trying to figure out clever ways to find the circles that corresponded to each pixel. Then I thought of trying the "dumb" solution and simply have each pixel check it's correspondance to all of the circles, and it ended up working very well!

I could probably further improve the current solution, by pre-computing (in parallel) which circles were contained within the grid the pixel belonged to (since the image was gridded up by blocks to put it on the GPU) using the circle-in-box check, in order to reduce the number of checks required by each pixel, but my solution worked so well I decided to leave it.