CS 4F03 – Distributed Systems:

Final Project Report

Stuart Douglas – 1214422
Matthew Pagnan – 1208693

April 4, 2016

Contents

1 Description of Parallelization **3**
- 1.1 OpenACC Overview . . . 3
- 1.2 OpenACC Data . . . 3
- 1.3 Writing BMP to Disk . . . 3

2 Computing Parameters for Frames **3**

3 Mandelbox Performance vs. Mandelbulb **3**

4 Source Code **3**

5 Running the Program **4**

6 Bonus Features **4**
- 6.1 Automatic Navigation . . . 4

1 Description of Parallelization

1.1 OpenACC Overview

The parallelization of the project is based off of the nested for-loop in `renderFractal` that iterates over each pixel in the output image, calculating the correct colour for that pixel based off of the passed parameters. When the loop is encountered, the necessary data from the CPU's DRAM is copied in. A kernel is then launched on the GPU, and each thread begins the processing for one pixel. Every function call within the kernel is run sequentially on the calling thread, with all subsequent routines from *those* functions inlined. Finally, the image data is copied back to the CPU.

1.2 Vector3D

Originally, a `vec3` was represented by a C++ class. We changed this implementation to a struct, and wrote macros to perform computations on the vector. There were a few macros originally included with the project, but many more had to be written to ensure all `vec3` operations could be run on the `vec3` struct. Using macros simplified integration with OpenACC, as routines are a relatively new feature and not fully robust yet.

1.3 OpenACC Data

The complex data that is private to each thread, such as the pixel data objects, the vectors storing the `colour` and the `to` vector for the pixel, and the double array containing the `farPoint` for the pixel are all stored in arrays, where each thread on the GPU accesses one element of each array. These values are not needed by the CPU at all, so the arrays are allocated on GPU memory using `acc_malloc`, and declared as device pointers. Note that they are allocated once at the beginning of the program and freed just before the program exits.

1.4 Writing BMP to Disk

We observed that once the program renders an image, it must wait for the CPU to write that image to disk before continuing to the next frame. We introduced a simple optimization to allow execution to continue, so that the GPU can be rendering the next frame while the CPU is writing the previous one to the disk. This was a simple matter of creating a new thread to write the image out, then continuing to the next iteration for rendering, swapping out the image buffer with another. Once execution reaches the `saveBMP` call again, it waits for the "write-out" thread to finish, then spawns a new thread and continues. Having two image buffers does increase memory usage, but it allows both the host and device to do time-consuming work at the same time.

2 Computing Parameters for Frames

3 Mandelbox Performance vs. Mandelbulb

4 Source Code

The following source code files were not changed from the serial version of the program, and as such will their contents will not be reproduced in this document.

- `camera.h`
- `color.h`
- `getparams.c`
- `init3D.cc`
- `mandelbox.h`
- `renderer.h`
- `savebmp.c`

Due to OpenACC requirements for nested inline routines being in the same source file, the `MandelBoxDE` and `DE` functions were moved to `raymarching.cc`. Functions for printing progress and timing data were removed, as they were no longer deemed necessary due to the speedups from running the program on the GPU. As such, the following source code files were removed.

- `distance_est.cc`
- `mandelboxde.cc`
- `print.c`
- `timing.c`

5 Running the Program

To generate the video, follow the following steps.

1. Run `make mandelbox` from the project directory
2. Execute the program with `./mandelbox params.dat`
3. Convert the images to a video by running `convert_to_video`

If you wish to change the number of frames, simply open `main.cc` and replace the value for the `NUM_FRAMES` pragma with the desired number of frames.

6 Bonus Features

6.1 Automatic Navigation

The automatic navigation functionality for the project was implemented. That is, the pgroam will determine a path through the mandelbox that does not "hit" any walls, nor leave the box itself. For more details on the algorithm used, refer to section 2.