4F03 Project

Authors

Name	Student Number	Email	Website
Jamie Counsell	1054209	counsej@mcmaster.ca	jamiecounsell.me
James Priebe	1135001	priebejp@mcmaster.ca	

Description

This C program renders a mandelbulb or mandelbox 3D fractal with optimizations in OpenACC.

Dependencies

- `pgcc` or compiler with support for OpenACC

Installation

- Install the dependencies
- Clone the repository

Operation

To run the program, first run make:

```
$ make clean
$ make [type]
```

Where `type` is one of:

- **`mandelbulb`** - Compute a Mandelbulb fractal
- **`mandelbox`** - Compute a Mandelbox fractal
- **`boxserial`** - Compute a Mandelbox fractal using a serial implementation
- **`bulbserial`** - Compute a Mandelbulb fractal using a serial implementation

Then run the command with the optional runtime flags:

```
$ ./mandel[box, bulb, bulb_serial, box_serial] params.dat [-f n] [-v]
```

where:

- **params.dat** is a file containing the Mandelbulb or Mandelbox parameters.
- **f** - Instruct the program to generate `n` frames. Default is 1 frame.
- **v** - Instruct the program to generate a video when it is complete (calling `genvideo.sh`)

For example, to generate a 7200 frame video at 30FPS (4 minutes) of the mandelbulb:

```
$ ./mandelbulb params.dat -f 7200 -v
```

To generate the mandelbulb given in the assignment, one can use the command:

```
$ make clean; make mandelbulb
$ ./mandelbulb paramsBulb.dat
```

OR the serial version:

```
$ make clean; make bulbserial
$ ./bulbserial paramsBulb.dat
```

The resulting images will be in the frames directory as `00000.bmp`. The filename used in the parameters is not used here to follow convention and ensure this frame can be used in the video.

Speedups

For the first frame of the submitted video, the following times were recorded:

Server	OpenACC	time
tesla	NO	108.16456s
tesla	YES	1.236812s

The server was under heavy load during testing, so future results may vary, but this shows a significant speedup (~87.5x faster with OpenACC than without). The same CPU was used to show speedups related purely to OpenACC acceleration.

Parallelization

The only region that was parallelized was the nested loop in `renderer.cc`. This loop is the program's largest bottleneck and also supports parallelization quite intuitively. OpenACC pragmas were used to identify the region as an OpenACC compute region, as well as transfer the data to the device from the host. The outer loop was explicitly marked as parallel, and other optimizations were left up to PGCC.

Functions called inside the compute region were identified as ACC Routines, and any functions called within such routines were inlined, due to the issue mentioned in class, where variables seem to take on a NULL or somewhat undefined value when they are passed to a function called by a Routine, even if that function is also marked as a Routine.

The data structures were flattened to be more easily passed between methods. To accomodate for this, additional parameters were added to the ACC Routines called inside the compute region. All data (including the flattened parameter structures) was explicity copied to the device using data pragmas before the beginning of the compute region.

Early on, we faced an interesting problem (and a great example of the proper use of the `present_or_copy[in, out]` ACC Methods. When image data (`image`) was marked as `copy` , `copyin` , or `copyout` , the compiler would generate code to reallocate `image` on the device each time the compute region began. This is because it has no way to maintain state across compute regions, and is therefore unable to maintain the pointer to `image` without explicitly checking if it is present first (then reuisng it). By adding the `present_or` prefix, we were able to instruct the compiler to not reallocate the memory, and instead overwrite the existing memory allocated for `image` . Since every pixel in `image` is changed before it is copied out, there are no problems here with risk of using old data.

Since frame parameters were not generated asynchronously, no parallelization was done to compute more than one frame at a time.

Frame Generation

Frames are generated sequentially from an array of `CameraParams` structures. The first image generated is always the same as what is identified in the input parameters. This ensures that the assignment requirements can be properly met with the given `paramsBulb.dat` file. After the first image, the camera rotates around the fractal, slowly decreasing its position in the `z` axis from `1` to `-1` across 7200 frames. The position is computed as follows:

- the `x` coordinate is `cos(frame_number/500)`
- the `y` coordinate is `sin(frame_number/500)`
- the `z` coordinate is `1 - (frame_number/3600)`

This will guide the camera around the object in a circular motion along the `x, y` plane such that it will complete one full rotation every `500*pi` frames. The z value decreases individually from `1` to `-1`

between frames `0` and `7200`, respectively. This creates a sort of *spring* path, showcasing all sides of the fractal.

Each iteration, `init3D` is called again to ensure the camera is still facing the center point at `(0,0,0)`.

With the exception of the first frame, each subsequent frame's parameters are generated during the previous frame's position in the loop. That is, the parameters for frame `i` are computed before rendering frame `i-1`. An array of `CameraParams` structures are kept in order to keep track of current and previous configurations. One could add support for rendering multiple frames at once, since the configurations are all available in memory. This would be a reasonable next step, and a good use for something like OpenMP.

Final Result

To compute the final result, the following configuration file (`bulb_params.dat`) was used:

```
# CAMERA
# location x,y,z (7,7,7)
1 0 1
# look at x,y,z
0 0 0
# up vector x,y,z; (0, 1, 0)
0 0 1
# field of view (1)
2.0
# IMAGE
# width height
3840 2160
# detail level, the smaller the more detailed (-3.5)
-3.45
# MANDELBULB
# ignore the first number, 0.
# the second and third numbers are escape (or bailout) time and power
0 4.0 9.0
# ignore the second number; the first number is the max number of iterations
100 0
# COLORING
# type 0 or 1
0
# brightness
1.2
# IMAGE FILE NAME
imageBulb.bmp
```

The command used to generate the result is:

```
$ ./mandelbulb bulb_params.dat -f 7200 -v
```

The URL for the video will be available on the GitHub repository and will be sent via email to the course instructor and TAs.

Source Code

See attached.