CS 4F03: Distributed Computer Systems
Final Project
Rendering a 3D MandelBulb

Josh Eppel — 001132624
Kevin Tan — 000963136
Fauzia Khanum — 001209252
Eric Amshukov — 001133146

April 11, 2016

Contents

1 Introduction

A Mandelbulb is a 3D fractal constructed using spherical coordinates. This project involved parallelizing an algorithm that constructs it and adding the functionality to compile multiple frames into a video animation. In this implementation, a data file containing coordinates for ray casting an image is used as input by the program in order to produce the bulb. Through meticulous inspection, it was discovered that certain code fragments could be optimized at compile time. This was done by modifying the code using more efficient C programming techniques. Furthermore, while time profiling the serial code, it was discovered that certain functions were significant bottlenecks during the execution of the program. But, these regions could be parallelized in order to improve overall performance. In order to produce an immense speedup, the OpenACC library was utilized to offload parallelizable computation onto the NVIDIA GPU. In conjunction, these improvements yielded the smoothest video frame rate.

2 MandelBox vs. MandelBulb

In comparison to the MandelBox, the geometry of the MandelBulb is more complicated to compute and so the program requires more time to generate an image. This can be observed in the following table which illustrates the runtime of both unoptimized implementations executed on an Intel Xeon 3.20GHz CPU.

	MandelBulb	MandelBox
Time	40.719	31.875

3 Algorithm

3.1 Video

The video output of the MandelBulb is produced by encoding a series of jpg frames into mp4 format. This was accomplished by introducing a bash script named `makeFrames.sh` which executes the bulb renderer a number of times but with differentiated parameters. Within this script, a for loop pumps out a hard-coded number of frames (currently set to generate frames numbered ff00000.bmp, ff00001.bmp, ..., ff07198.bmp, ff07199.bmp) . Each iteration of the loop renders a unique image with camera coordinates (position) multiplied by a floating point factor valued close to but less than 1.0 (These new coordinates replace the old ones in the `paramsBox.dat` file with each iteration). This produces the effect where the camera in each image seamlessly moves asymptotically closer to the center of the bulb. Furthermore, each image is stored and named uniquely after the current iteration value. Lastly, once all the images have rendered, the ffmpeg video encoder is used to compile all the frames produced into an mp4 video format by invoking the following commands:

```
$ fmpeg -framerate 30/1 -pattern_type glob -i '*.jpg'
    -c:v libx264 -r 40 -pix_fmt yuv420p out.mp4

$ ffmpeg -i out.mp4 -i timeHZ.mp3 -codec copy
    -shortest outAudio.mp4
```

Alternatively invoking the makeVideo.sh script should also create the video (details below in 'Running the Program').

4 Parallelization

4.1 Preliminary optimization

4.1.1 Inlining

At its current release, OpenACC appears to have trouble locating code that is called from a function within a function. Much of the serial code funtions therefore had to be inlined, to generate their code in place of where they were called. Every function called from UnProject, getColour, and rayMarching required this procedure. Additionally, UnProject and getColour had to be inlined to execute code on ray collision with the fractal, but attempting to inline rayMarching proved to blur the image.

To ensure all code could be accessed, all original source C++ and C files were remade into header files, and included in an appropriate order within main.cc. This required that much of the pre-existing includes/linking code be removed in favor of the simplified structure.

The code also had to be adjusted to used accelmath.h rather than math.h in areas to be parallelized on the GPU. Without this adjustment, functions such as sqrt() and cos(), which are called in functions targeted for parallization, would not have been visible to the GPU.

4.1.2 Macros

The original serial code used a class for vector objects. To make the code parallizable, this class was replaced by a simple struct. While this made copying to the GPU possible, it also eliminated the overloads which the class had been using to quickly solve mathematical operations on the vectors. Each of these overloads had to be replaced with a definition replacement. A provided vector3d.h file had many of the necessary operations, but many more had to be created to ensure all operations were possible without the class. Here is a list of added macros:

- SUBTRACT_NUM
- ADD_NUM
- MUL_NUM
- DIV_NUM
- ADD_VEC
- SUBTRACT_VEC
- MUL_VEC
- MATCH_VEC (sets one vector equal to another)

4.2 OpenACC

The performance of this program significantly improved by offloading certain regions onto the NVIDIA GPU for processing. Adhering to the shared memory parallelism paradigm, OpenACC provided an effective means for accomplishing this task through its API. This library contains useful compiler directives that allows code to specify which segments should be parallelized and which to remain serial on the GPU.

The code was analyzed using Intel's vTune software. From the hotspot results it was clear that the outer `for` loop in rendererCC.h would be the most effective region for parallelization. Most of the time-consuming work is done inside of that loop, and it has an optimal level of granularity with no dependencies. In order to parallelize this region, first specific data had to be traced to determine what would be to be sent onto and out of the GPU. To simplify this trace, all three parameter structs were "disassembled" by copying data to be used into independent variables. The functions called from within the loops were updated accordingly. The variable "color", requiring access each iteration, was remade into an array of the same type. In this way, each thread could access it's own designated slot and not overwrite/read incorrect data. There were four variables that were identified to require private access within the for loops. Rather than create an array for each, the acc private() clause was utilized to generate a local copy for each thread.

The specification of "gang" and "vector" were added to the outer and inner for loops, respectively. Without these clauses specifying how the code should be divided, the executable would "hang" at runtime. The vector specification, being the innermost, was assigned the private() clause.

The previous regions dicussed were acc for loops, and as such, were inside a main acc parallel region. Using parallel rather than kernels enabled the use of the private() clause. Upon calling the acc parallel pragma, data flow was explicitly outline using

1. `present_or_copyin` (to deal with an instance of redundant memory writing) and,
2. `copyout` (to ensure the image[ ] was fetched after processing.

As stated, the outer loop of this region was divided up into gangs using the `acc loop gang` pragma. This created an even distribution of iterations that could execute independent from one another. Subsequently, the crucial `acc loop vector private(...)` directive was added to the beginning of the inner loop. The parameters in the private clause were specifically `k, farpoint, to, pix_data_escaped, pix_data_hit,` and `pix_data_normal`. As a result of these measures, efficiency was preserved since these steps eliminated the need for large array data transfers of between hardware via a slow PCI bus.

4.2.1 Sequential

In order to implement parallelism correctly, certain methods which had dependencies or other characteristics that made them unparallelizable had to run sequentially. In earlier versions of the solution, `pragma acc sequential` statements were added to explicitly identify these regions to the compiler:

- getcolor
- rayMarch
- DE

However, after some cleaning up, and the full inlining process, they were removed as the compiler was successfully identifying these regions implicitly.

5 Running the Program

The Mandelbulb program can compile and run by following these steps:

Compile the program by executing the following code in the terminal:

```
$ make
```

Next, run it by issuing the following command:

```
$ ./mandelbox paramsBulb.dat
```

This should produce the following output:

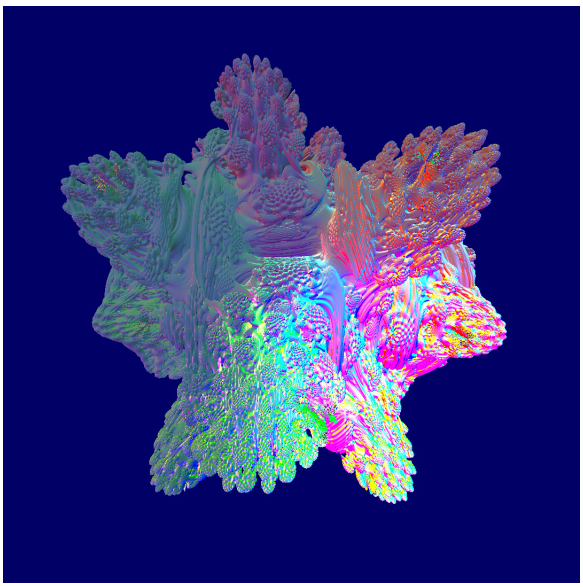

Figure 1: A single frame of the MandelBulb video

To generate our video one must swap out the raymarchingCC.h file with the raymarchingCC_box.h (replace the name of the file) to be able to generate a mandelbox instead of mandelbulb.

Then simply run the makeVideo.sh script which:

- Remakes the executable (invokes 'make clean' followed by 'make')
- Calls the makeFrames.sh script, which creates and converts frames from .bmp to .jpg by:

1. Replace the appropriate camera paramters (in this case the camera position with varX, varY and varZ) in the parameters file.
2. Invoke the executable with the parameters file.
3. Replace the image name.
4. Run the conver.sh script to convert the image from .bmp to .jpg format.
5. Calculate the next set of camera parameters and store to varibles (in this case the camera position with varX, varY and varZ).
6. Repeat until the number of iterations is completed (while the iteration number is less than 7200 in our script).

- Creates a 4 second video of title screen frames.
- Creates a 30fps video using all available .jpg frames (in the directory).
- Combines both videos into a single video.
- Adds audio to the video.

6 Benchmarks

As we adjusted the code, we ran versions of it to test for speedup results. The unoptimized, without any modifications aside from changing the MandelBox to a MandelBulb; the optimized, which included several compiler optimizations; and the GPU parallelized version were run on different machines from the CAS department. Each server has a different NVIDIA GPU which allowed the execution time to vary.

The elapsed time was calculated by running the bash command as follows:

```
$ time ./mandelbox paramsBulb.dat
```

Table 1 illustrates the time and speedup corresponding to a single frame being produced from running each of the three implementations.

Table 1: MandelBulb Performance (Single Frame)

			Elapsed Time (seconds)			
Machine	GPU	Cores	Unoptimized	Optimized	Parallellized	Speedup
gpu2	Quadro K5000	1536 CUDA	43.149	30.617	2.205	19.57
Tesla	Tesla k40c	2880 CUDA	41.116	31.639	0.743	55.38