Multicore Architecture
Ray-Tracing

CS 535 - Multicore Programming Term Project

Doğa Yılmaz, Onur Kirman

June 6, 2022

1 Project Definition

We have implemented a simple ray-tracing program that graphically renders the physical behavior of light. We have created vector and color functions for multiple objects such as spheres. Also, we have generated various materials such as glass, metal, and plain paint. These materials have different surface reflection properties, which can absorb some of the lights according to the light reflection coefficient. After constructing a sequential ray-tracing algorithm, we analyzed it using a profiler to detect any bottlenecks which are discussed in the evaluation metrics section in detail. Following the analysis, we diagnosed the bottlenecks and parallelized the sequential program incrementally to boost its efficiency per bottleneck. Finally, we analyzed each version of the program using the profiler and discussed the results, and mentioned possible future work.

2 Terminology

2.1 Rendering

Rendering is the process of generating an image from a model. The process is relatable to taking a photograph in the real world which requires a light source and some reflective surfaces with a light collecting device (i.e. human eye, camera).

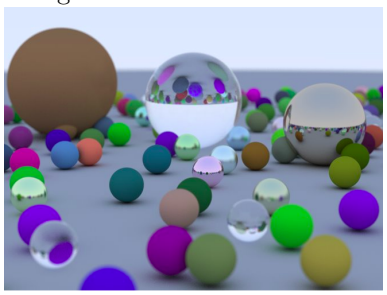
2.2 Ray-Tracing

Ray tracing is a technique for modeling light transport, which models the light rays as vectors over the scene which has a reflective surface within. This operation is computationally expensive and nowadays, it is accelerated by GPUs efficiently.

3 Tools and Methods

For the implementation, we used C++ as our programming language which provided decent flexibility to manipulate multiple threads. To compile our code, we used GNU Compiler Collections (GCC). Also, as our profiler, we used Intel's VTune profiler which is sufficient for evaluating the performance of our code using the metrics that are provided in the next section. In addition to VTune, we used Valgrind which is an instrumentation framework for building dynamic memory and cache analysis tools.

Figure 1: Benchmark Scene



4 Evaluation Metrics

Before performing any analysis, we defined a benchmark scene to test different versions of the program in a controlled and equal manner. The scene has random little spheres and ranging 3 big spheres over a plain surface which can be seen in Figure 1. To measure the performances of the programs, we will use the Hotspots, Memory Consumption, CPU wall clock time, and Threading

analysis of our profiler (Intel VTune). In Hotspots analysis, we will observe the elapsed time and parts of the program that consume most of the time which will help us to parallelize our program. In Memory Consumption, we will be able to see the memory allocations of our programs. Also, in Threading analysis, we can measure the parallelism of our algorithms.

5 Experiments and Analysis of Sequential Ray-Tracing

CPU:	Intel Core i7 3770k (Ivybridge)
Frequency:	3.5 GHz
Virtual Cores:	8

Figure 2: Platform Information

After we created our base sequential program that ray-traces using a sequential manner, we experimented with different parameters and observed the results using the evaluation metrics we have discussed. The independent variables of our tests are resolution (height and width of the rendered image) and number of rays. Also, our platform information is as shown in the Figure. 2 which has a total of 8 threads. Using these different independent variables and the platform, first we observed the 150x100 resolution with 100 rays performance of sequential code. It performed poorly with an execution time of 82 seconds with 1 thread utilization out of 8.

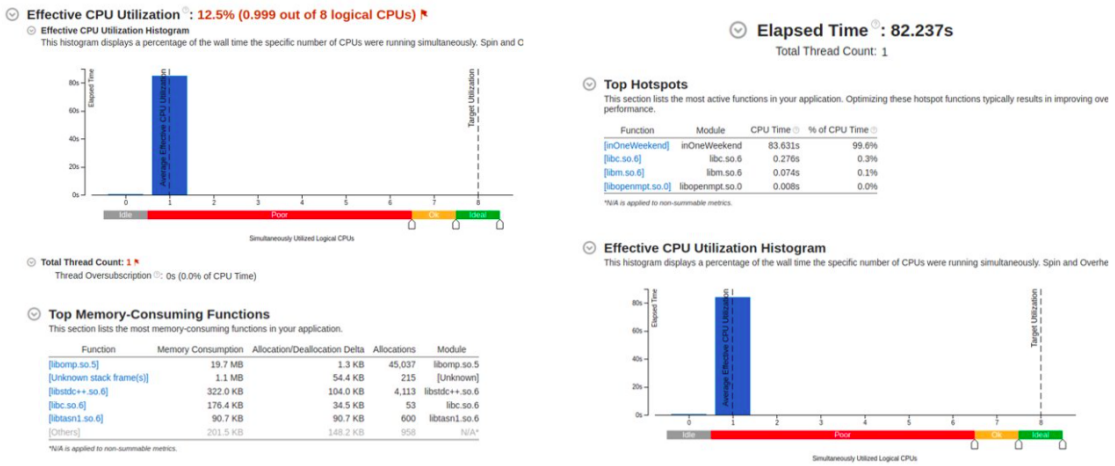


Figure 3: Sequential Code Evaluation with Resolution = 150x100 and Ray = 100

As we can clearly see from the Figure. 3, program uses the same function 99.6% of the time. This shows that there can be improvement within the function. Moreover, we increased our resolution by making it double in both width and height to see performance on higher resolutions, and ended up getting the results in Figure. 4. We can say that the performance decreases a lot when we increase the resolution with an exponential manner, the execution time scaled to 328 seconds which is unacceptable for a single image. Thus, after checking our render function and code sections where we gather pixel color values, we saw three different performance improvements which can be defined as bottlenecks.

5.1 Outermost Loop - Height

First, the algorithm traverses each row of the render image one by one. This region can be parallelizable since each row will be using its own region and calculation to find pixel values within the row, which does not cause any loop dependency or race condition.

5.2 Middle Loop - Width

Second, similar to first part, algorithm traverses each column of render image to find individual pixel values within the row. Also, this part can be parallelizable since each pixel will require the

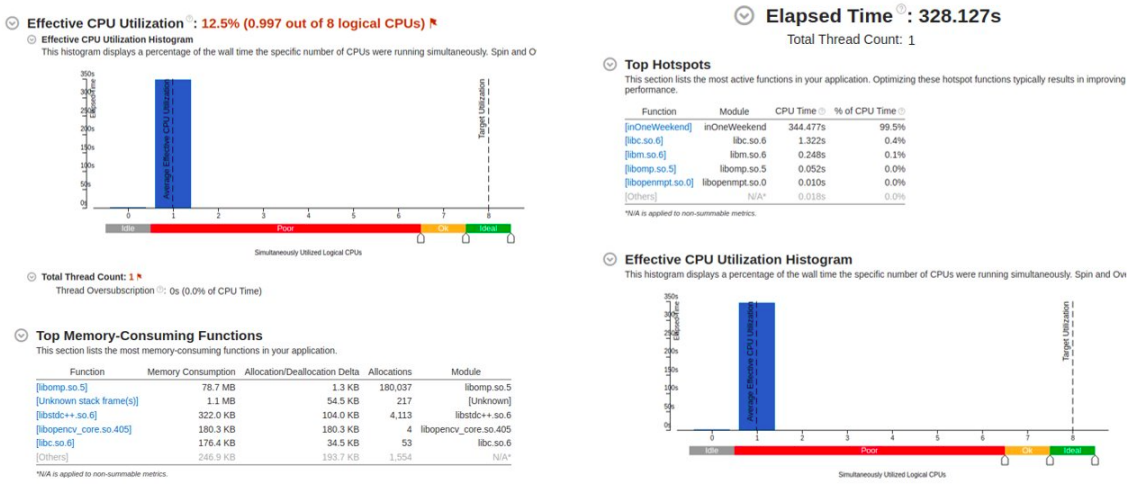


Figure 4: Sequential Code Evaluation with Resolution = 300x200 and Ray = 100

calculation of color in different region, similar to height part.

5.3 Innermost Loop - Ray Trace

Lastly, the innermost loop can be parallelized using multiple threads in parallel. Because the program launches N number of rays to calculate the corresponding pixel's color value which are independent between pixel values.

6 Proposed Solution

As we mentioned in the previous sections, there are some bottlenecks in the sequential code. In this section we will introduce our parallelization strategies which we have designed by using the evaluations and analysis made over the sequential code. We have parallelized the sequential code step by step, detailed explanation of each step is explained below.

6.1 First Level of Parallelization: Iteration Over the 2D Output Array

Our first parallelization attempt is to parallelize the double for loop which is responsible for iterating the output array which calculates the corresponding RGB values for them. We have used the following expression for the parallelization of the section.

```
# pragma omp parallel for num_threads(thread_count) collapse(2)
```

As seen above we have used the collapse clause which is used for parallelizing nested for loops. In our case since we only need to parallelize a double for loop, adding collapse(2) was sufficient for the parallelization.

6.2 Second Level of Parallelization: Accumulating the Values From N Number of Rays

In order to further increase the performance of our implementation, we have also parallelized the calculation of accumulating the return results of different rays. To do so we have used the reduction clause provided by OpenMP. The whole pragma is shown below.

```
# pragma omp parallel for num_threads(thread_count) reduction(+: col)
```

6.3 Experiments of Multi-Core Ray-Tracing

In this section we will present our test and experiment results of parallelized code which we have obtained using VTune profiler and Valgrind memory analysis tool.

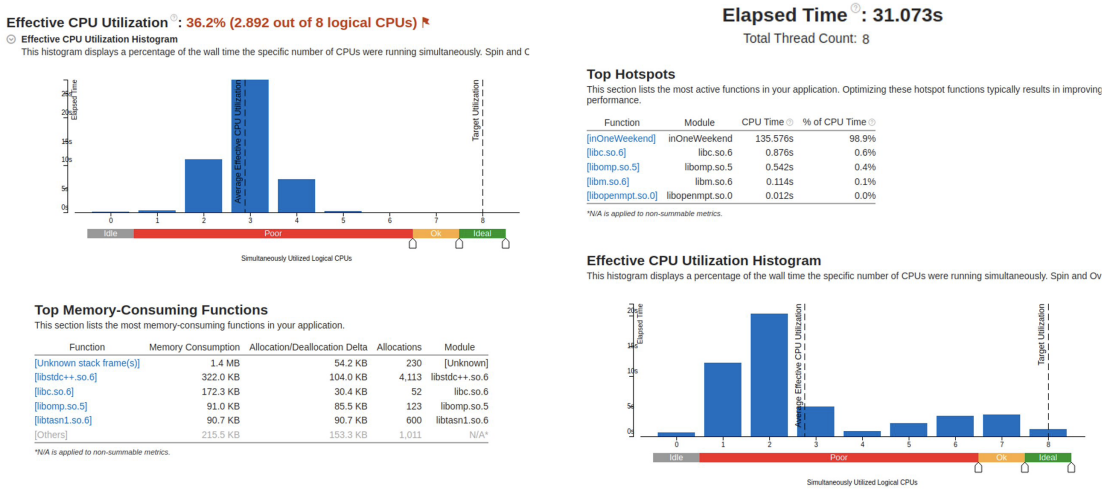


Figure 5: Parallel Code (First Level Parallelization) Evaluation with Resolution = 150x100 and Ray = 100

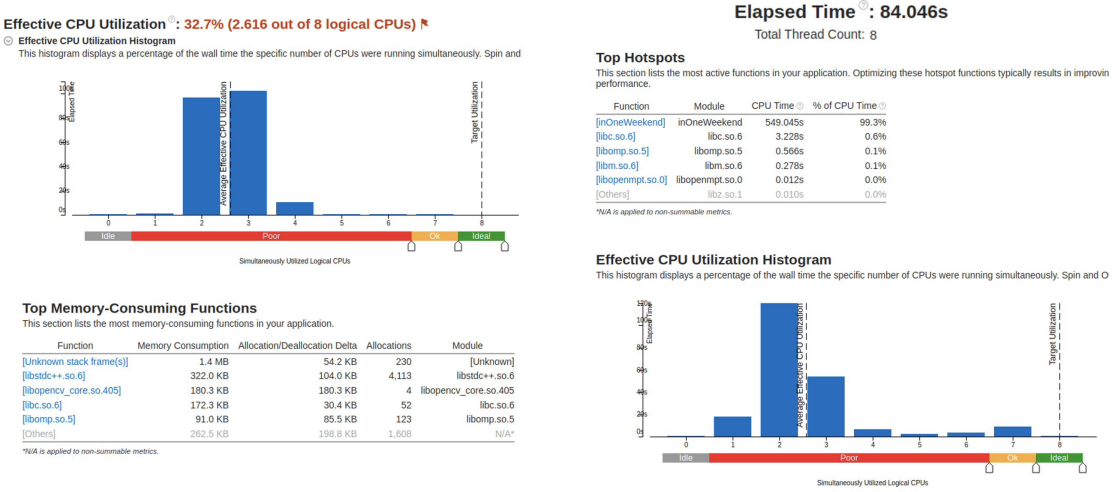


Figure 6: Parallel Code (First Level Parallelization) Evaluation with Resolution = 300x200 and Ray = 100

6.3.1 Experiments of First Level Parallelization

As it can be seen above, our parallelized code (first level) utilizes around 36% of the avaliable CPU cores and if has finished rendering our benchmark scene in 31 seconds for 150x100 image and 84 seconds for 300x200 image.

6.3.2 Experiments of Second Level Parallelization

As it can be seen above, our parallelized code (second level) utilizes around 72% of the avaliable CPU cores and if has finished rendering our benchmark scene in 22 seconds for 150x100 image and 1.6 seconds for 300x200 image.

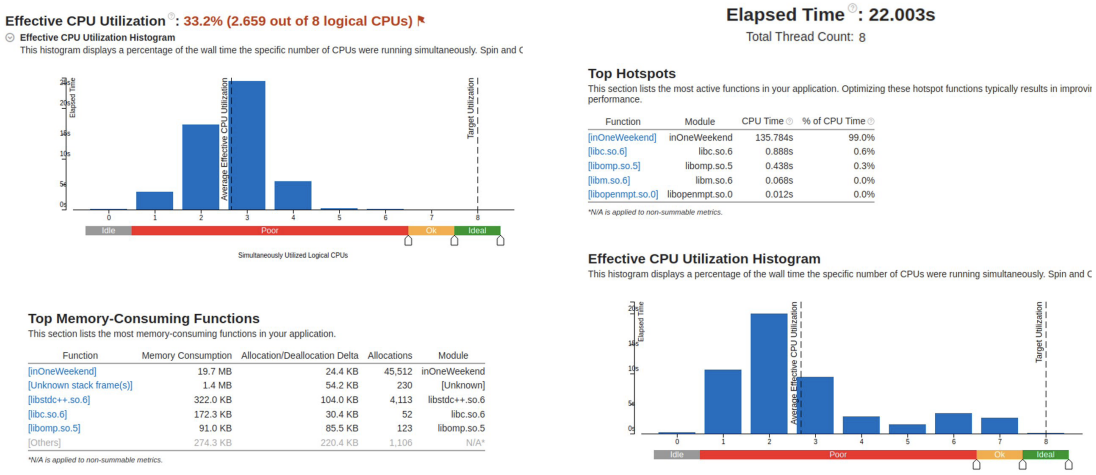


Figure 7: Parallel Code (Second Level Parallelization) Evaluation with Resolution = 150x100 and Ray = 100

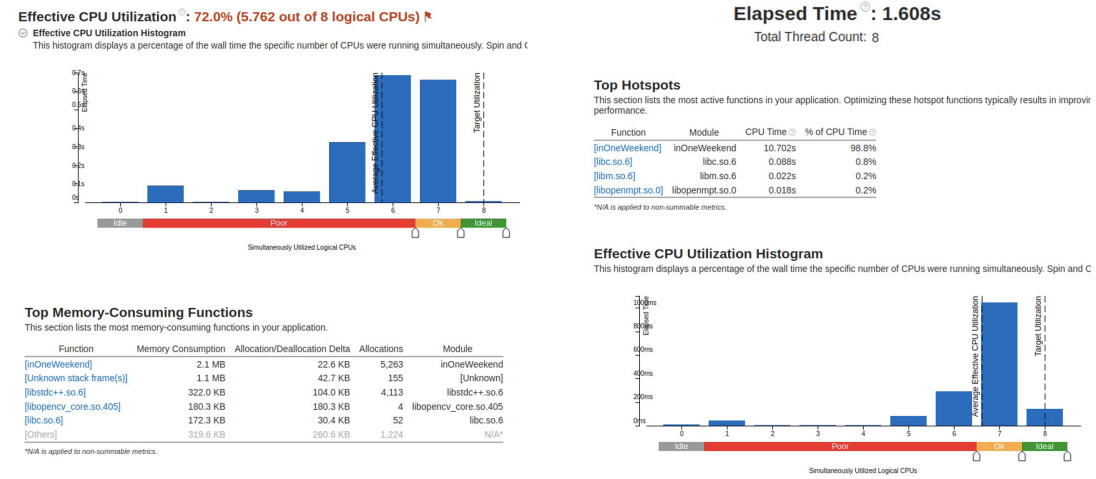


Figure 8: Parallel Code (Second Level Parallelization) Evaluation with Resolution = 300x200 and Ray = 100

6.4 Valgrind (Cachegrind) Analysis of Parallel Ray-Tracing

Again to further analyse the performance of the code we have also investigated the memory and cache usage of our program.

```
==2901== Cachegrind, a cache and branch-prediction profiler
==2901== Copyright (C) 2002-2017, and GNU GPL'd, by Nicholas Nethercote et al.
==2901== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==2901== Command: ./inOneWeekend
==2901==
--2901-- warning: L3 cache found, using its data for the LL simulation.
duration (ms): 52597
free(): invalid next size (normal)
==2901==
==2901== Process terminating with default action of signal 6 (SIGABRT)
==2901==    at 0x11403E87: raise (raise.c:51)
==2901==    by 0x114057F0: abort (abort.c:79)
==2901==    by 0x1144E836: __libc_message (libc_fatal.c:181)
==2901==    by 0x114558B9: malloc_printerr (malloc.c:5342)
==2901==    by 0x1145D07C: free (malloc.c:4316)
==2901==    by 0xF553898: cv::Mat::deallocate() (in /home/parallels/installation/OpenCV-master/lib/libopencv_core.so.4.6.0)
==2901==    by 0xF55399C: cv::Mat::release() (in /home/parallels/installation/OpenCV-master/lib/libopencv_core.so.4.6.0)
==2901==    by 0xF5539AC: cv::Mat::~Mat() (in /home/parallels/installation/OpenCV-master/lib/libopencv_core.so.4.6.0)
==2901==    by 0x406B07: main (in /media/psf/ubuntu_shared/ray-tracing/build/inOneWeekend)
==2901==
==2901== I   refs:      11,928,914,756
==2901== I1  misses:            29,260
==2901== LLi misses:            11,003
==2901== I1  miss rate:           0.00%
==2901== LLi miss rate:           0.00%
==2901==
==2901== D   refs:       9,630,262,180  (5,824,148,498 rd   + 3,806,113,682 wr)
==2901== D1  misses:        11,012,149  (    10,708,632 rd   +       303,517 wr)
==2901== LLd misses:           226,937  (       147,883 rd   +        79,054 wr)
==2901== D1  miss rate:            0.1% (            0.2%     +           0.0%  )
==2901== LLd miss rate:            0.0% (            0.0%     +           0.0%  )
==2901==
==2901== LL refs:           11,041,409  (    10,737,892 rd   +       303,517 wr)
==2901== LL misses:            237,940  (       158,886 rd   +        79,054 wr)
==2901== LL miss rate:             0.0% (            0.0%     +           0.0%  )
```

Figure 9: Valgrind Low Res

```
==4647== Cachegrind, a cache and branch-prediction profiler
==4647== Copyright (C) 2002-2017, and GNU GPL'd, by Nicholas Nethercote et al.
==4647== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==4647== Command: ./inOneWeekend
==4647==
--4647-- warning: L3 cache found, using its data for the LL simulation.
duration (ms): 12183
==4647==
==4647== I   refs:      3,113,838,512
==4647== I1  misses:           27,154
==4647== LLi misses:           12,245
==4647== I1  miss rate:          0.00%
==4647== LLi miss rate:          0.00%
==4647==
==4647== D   refs:      2,433,688,059  (1,483,152,237 rd   + 950,535,822 wr)
==4647== D1  misses:       10,569,173  (   10,396,912 rd   +     172,261 wr)
==4647== LLd misses:          224,960  (      148,202 rd   +      76,758 wr)
==4647== D1  miss rate:           0.4% (          0.7%     +         0.0%  )
==4647== LLd miss rate:           0.0% (          0.0%     +         0.0%  )
==4647==
==4647== LL refs:          10,596,327  (   10,424,066 rd   +     172,261 wr)
==4647== LL misses:           237,205  (      160,447 rd   +      76,758 wr)
==4647== LL miss rate:            0.0% (          0.0%     +         0.0%  )
```

Figure 10: Valgrind High Res

As it can be seen from Figures 9 and 10, we can say that cache miss reduction in higher resolutions are causing the further increase in performance and lower execution times. This can be caused by temporal locality of algorithm, while running the first low resolution render, it may keep some of the cache which is directly available for some other parts in the higher resolution that does not require any calculation. In the figure, D1 and I1 refers to L1 and LL refers to last L which is L3 in our case. Also, in further inspection the read and write of data decreases in higher resolution while in lower resolution it calls higher number of times. Thus, the simulation also shows that we miss less in higher resolution due to temporal locality in the memory.

7 Results and Conclusion

In this section we will analyse our experiment results and discuss the performances of various versions of our ray-tracing program.

7.1 Quantitative Analysis

As you can see from the table, the sequential version is the worst performing one among the others as expected. The increase in resolution exponentially affects the run-time of the program which is what we do not want in the first case. After incrementally solving parallelization, we can see that low and high-resolution execution times decrease significantly. But the ones with ray parallelization enabled are the fastest. Thus, we achieved our goal of parallelization of ray-tracing in a much sharper performance increase than we wanted. But we should also point out that if we look at the Figure. 11, we can see that while we increase the resolution of the image from 150x100 to 320x200, the execution time drops even further from 20 seconds to 3 seconds. This

Type	Resolution	Parallel-Ray	# Threads	Elapsed Time (s)	CPU Utilization %
Sequential	150x100	No	1	82.237	12.5
Parallel	150x100	No	8	31.073	36.2
Parallel	150x100	No	16	20.333	48.6
Parallel	150x100	Yes	8	22.003	33.2
Parallel	150x100	Yes	16	19.696	37.5
Sequential	320x200	No	1	328.127	12.5
Parallel	320x200	No	8	84.046	32.7
Parallel	320x200	No	16	77.208	88.2
Parallel	320x200	Yes	8	1.608	72
Parallel	320x200	Yes	16	3.395	80.4

Figure 11: Experiment Results

is unexpected, so we used the Valgrind tool to inspect the memory utilizations, and commented detailly in the previous section.

7.2 Qualitative Analysis

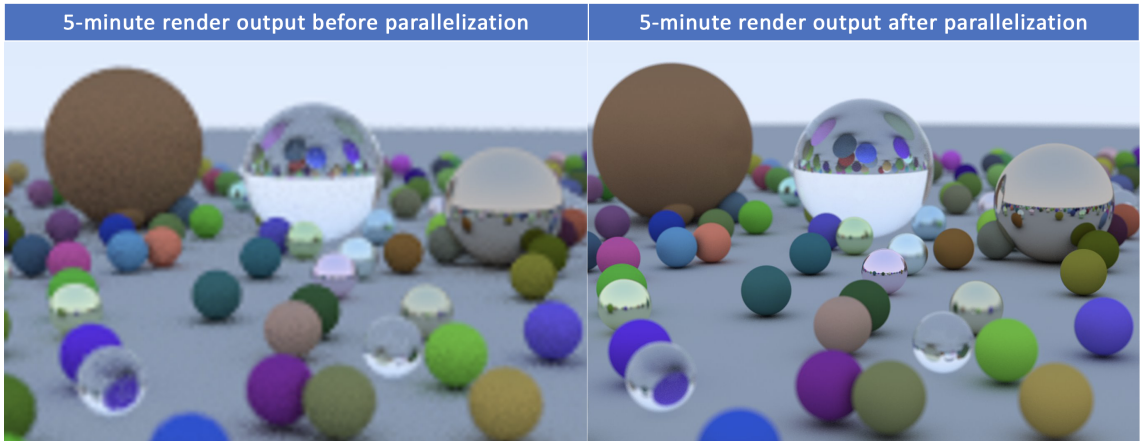


Figure 12: 5 minute rendering result of sequential code and parallelized code (second level)

As it can be seen from the figure above, qualitatively the parallelized code generates much better looking output images at a fixed amount of time. In this case we have picked 5 minutes for both versions of the code (Figure. 12). Not only we got crisp clear render, we increased the resolution by double in each axis to match the execution times.

8 Future Work

In addition to our project, the scope can be extended to GPU computed version, which can show a better performance compared to a CPU. This can be done using the CUDA library which is created just for GPU acceleration concept. Thus, the overall run-time may decrease and the high resolution performances may be visible and attainable.