Parallel Performance Analysis of a Ray Tracer Algorithm Implementation

Marco Moroni

December 2018

Abstract

This report presents an attempt to parallelise a ray tracer. Two methods are used: OpenMP (CPU parallelisation) and CUDA (GPU parallelisation).

1 Introduction and Background

Ray tracing is a rendering algorithm that colors the pixels by casting rays from the camera view point. When a ray hits an object it bounces and every time time it happens it collects color data which are then combined. The *quality* of an image created though this method is given by the *samples per pixel* (or *SPP*): within each pixel, multiple randomly distributed rays are cast. The noise of picture will be lower as the number of SPP increases.

A ray tracer parallelisation is an *embarrassingly-parallel* problem because each pixel color is calculated individually. Data-level parallelism can therefore be attempted—in particular, parallelisation within the GPU can be extremely beneficial.

Initially, the code from *Ray Tracing in One Weekend* [2] was used and an initial analysis was undertaken, but it was later scrapped in favour of a simpler version. The reason is that implementing CUDA would have required advanced knowledge of the system.

2 Initial Analysis

The source code used for this project comes mostly from Beason, K. [1], but it was changed and simplified in different parts following the example of Lapere, S. [3]:

- The random function uses the default C++ `rand()`
- Pixel subdivision was removed
- Tent filter was also removed
- It only supports diffuse materials
- Recursion of the function that calculates the colour of a ray has been substituted with a *for* loop

Pixel subdivision and tent filter are unnecessary features removed in order to make the code as simple as possible.

The reason why only one type of material (diffuse) is supported is due to the future implementation of a GPU parallelisation technique, in this case CUDA. The GPU works best when there is minimal code branching, and this eliminates the need to check the type of material. The same reason applies to the removal of recursion in favor of a *for* loop in the function that calculates the colour of a ray (`radiance()`): before, when a ray bounced, it would recursively check for the next bounce, now, the number of times a ray bounces is predefined and everything happens inside a *for* loop.

These changes have been made to allow a more precise investigation, in which the same code can be used to test the different parallelisation techniques.

The algorithm of the ray tracer used for this analysis can be summarised as:

- For each pixel, for each SPP, calculate the radiance
  - Radiance iterates as many times as the number of bounces per ray
    - At every iteration of the radiance function, a loop goes through all the object in the scene (to check for collision with ray)

The code was initially analysed through the Visual Studio Performance Profiler tool, which identifies the most CPU intensive parts of the code (figure 1). Of course, inside the loops that iterate though every pixel and every sample, the `radiance` function (the function that gets the final colour of a ray) is the most intensive. In that function, the one that gets the ray collision data is the most intensive (`intersect`). This brings to the following conclusions:

- As expected, the higher the resolution (number of pixels) of the image, the higher the amount of rays that are needed, therefore the `radiance()` function will be called more times. This is the loop that will be parallelised, as this fits well the type of problem a GPU can solve.

- Similarly, `radiance()` will be called for each sample per pixel, making this function even more resource consuming. This loop could also be parallelised.

- The reason why `intersect()` is so intensive is because it loops through every element in the scene (in this case spheres) and for each of them it checks whether a collision occurs. Therefore, the more the objects in the scene, the intensive this function will be.

  - Although this is not relevant to this investigation, it should be noted that this process can be improved by different space partitioning techniques, by which a ray can check only some of the objects in the scene.

- `intersect()` is called every time a ray bounces. Considering that the number of ray bounces in this path tracer implementation is limited to 4 (the reason is explained above), the `intersect()` function would be called even more times with a recursive `radiance()` function. This loop cannot be parallelised, because a ray requires the one before to be calculated.

To summarise, the variables that most influence the performance of the algorithm are:

- the resolution of the image;

- the number of samples per pixel;

- the number times a ray can bounce;

- the number of elements in the scene.

Table 1 defines the default values that will be used throughout this report and figure 2 shows how these values affect the output.

Multiple images were generated with different values for each variable. For each instance, 10 images were created. Figure 3 shows the results gathered by measuring the time it takes to generate an image (without including the time taken to write to a .ppm

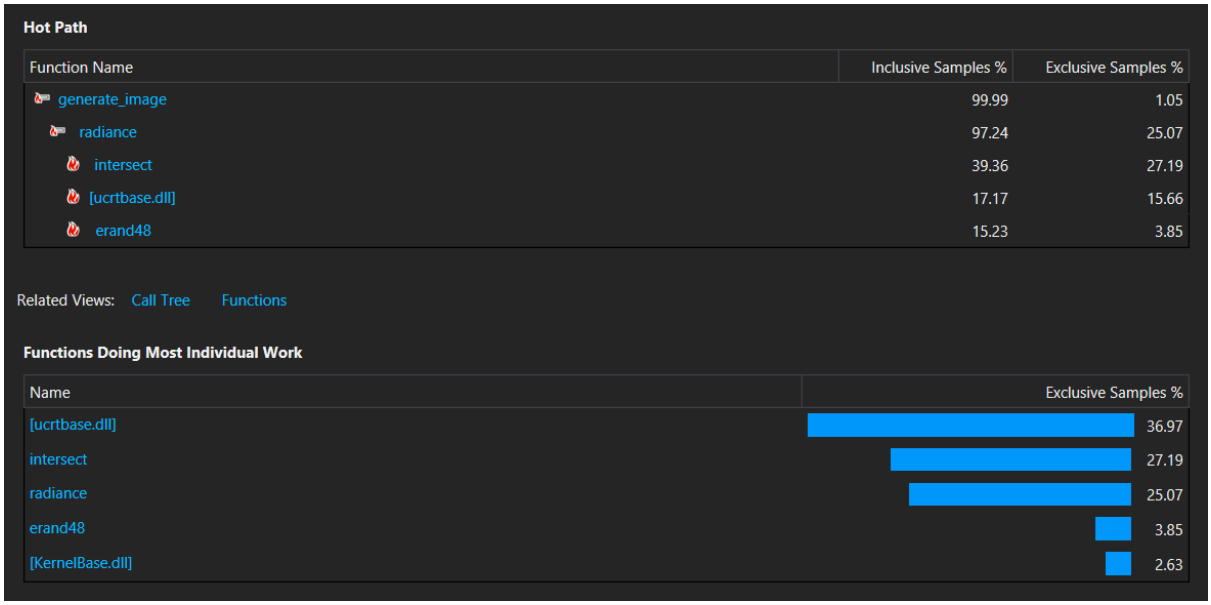

Figure 1: Part of the Visual Studio CPU Performance Profiler results.

Variable	Value
Resolution	1024 x 768 (786432 pixels)
Samples per pixel	100
Bounces	4
Spheres in the scene	9
Images generated per test	10

Table 1: Default values.

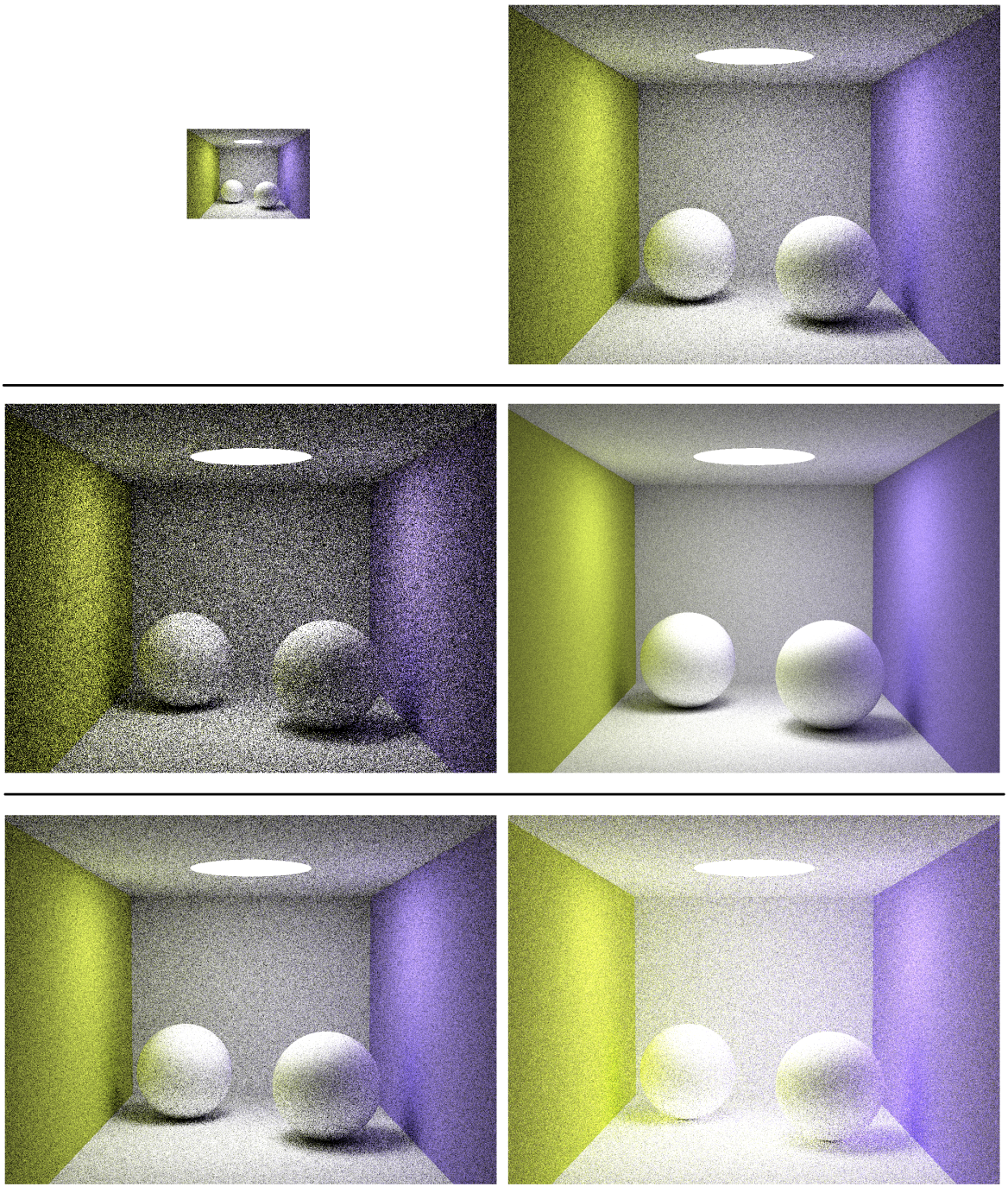

Figure 2: Variables comparison. The rows represents resolution, samples per pixels and ray bounces. The latter seems brighter, and the reason is that a ray, by bouncing more, gets more light.

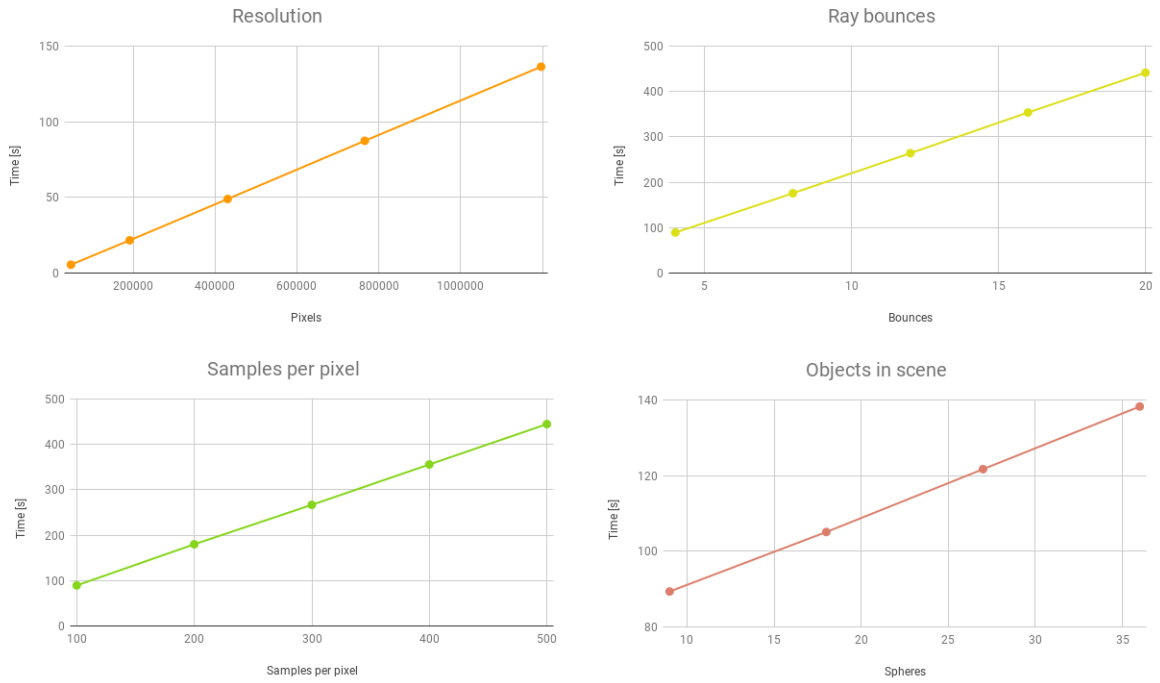

Figure 3: Time taken to generate images with different properties.

file). For the rest of this investigation, only the resolution and the SPP will be considered, as the other two variables are very specific to this implementation and do not represent adequately ray tracers in general.

3 Methodology

The objective is to parallelise the algorithm by using the GPU. The code chosen to analyse has been already been optimised for that in some parts, as stated in *Initial Analysis*, in order to allow a more precise comparison between every version of it. The *Initial Analysis* section also suggests that the part of the code to target is the *outer for* loop, the one that iterates though all the pixels. SPP, however, is the most important variable when defining the quality of an image generated through a ray tracer, so it may be valuable to investigate different configuration event though the loop concerning the SPP is not parallelised.

The PC specifications are shown in table 2.

Before attempting GPU parallelisation, OpenMP was used to test if code could actually be paralysed. A `#pragma omp parallel for` line was added to the pixels loop

CPU	Intel(R) Core(TM) i7-4790K CPU @ 4.00Hz
RAM	16GB
GPU	NVIDIA GeForce GTX 980
OS	Windows 10 Enterprise

Table 2: PC specifications.

and time was calculated while this loop was running. The results were satisfactory and showed that the code could be correctly parallelised.

The CUDA implementation required various changes to the code:

- Functions and variables had to be declared as ones that can be used by the host and/or the device (GPU) by using `__host__` and `__device__`.
- Variables initialised outside a function are not supported, therefore the declaration of the scene had to moved inside the `main()` function. The scene consists in an array of `Sphere`, which now is created in the stack and copied to the GPU. The functions that need to access the array now take two additional parameters: a pointer to the beginning of the array and the number of elements contained.
- The *for* loop the iterates through the pixels was replaced by the function that will be executed in the GPU's kernels. The function calculates the colour of one pixel and its index is calculated from the `blockIdx` and `blockDim` values.
- The method to get a random value had to changed because `rand()` is not supported by CUDA.
- Code branching is already reduced to a minimum in order to allow a fair comparison between the different methods.

Additional timing data was gathered to measure how long it takes to copy data from and to the GPU.

CUDA implementation presented some problems while implementing. Multiple CUDA implementations of *smallpt* are available online, but running them lead to a GPU crash. To have a working version of it, the original was changed step by step, testing every time whether there were problems. This helped discovering that the reason why the GPU was

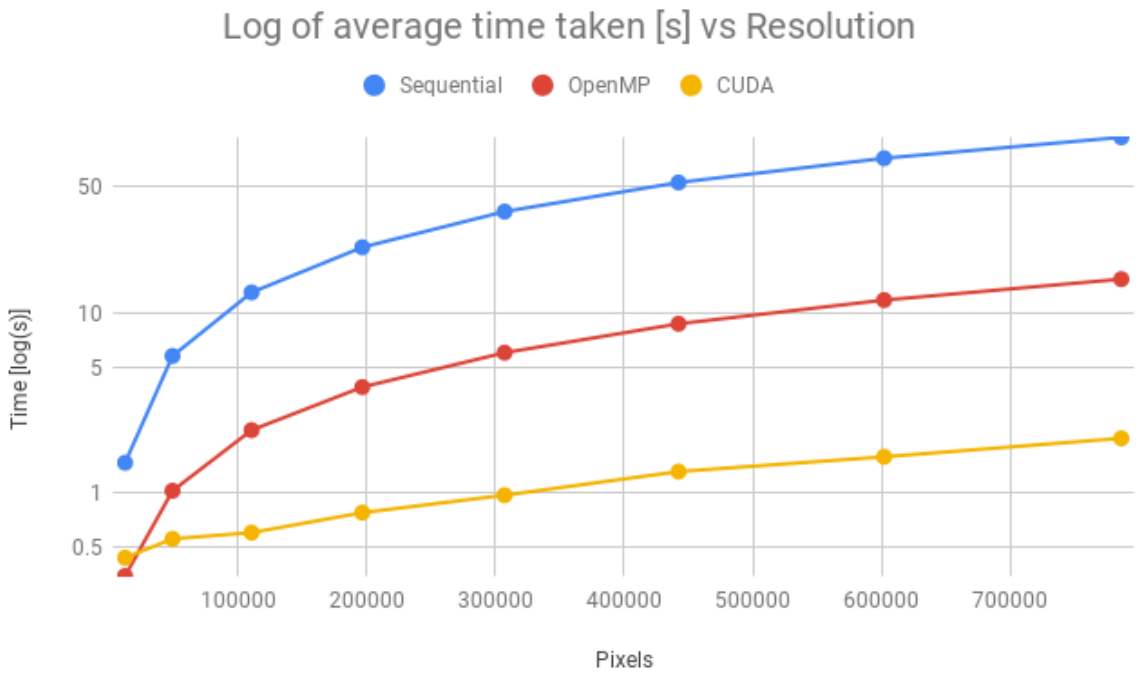

Figure 4: Comparison of results by method and image resolution.

crashing was due to the limited memory the GPU can hold: 140 or more SPP will lead to that. Lowering the resolution will allow for more SPP and vice versa.

Given these memory restrictions, new timing data has been collected for all the methodolgies in order to be able to compare them. If previously the range of SPP was $\{100, 200, 300, 400, 500\}$, now it will need to contain lower values: $\{20, 40, 60, 80, 100\}$. Similarly, the resolutions range had to be modified in order to accomodate the GPU.

4 Results and Discussion

For each image, 10 of them were generated and then the average time was calculated. The results are presented in figure 4: the values are the average times.

Table 3 shows the time speedup and efficiency compared to the sequential algorithm.

The results shows that the implementations were successful in almost consistently speeding up the ray tracer and CUDA in particular makes the image generation extremely faster. However, CUDA seems not to be beneficial when it doesn't have a lot of work to perform, most likely the reason is because the time for memory management is higher than the actual kernel execution. Figure 5 shows that, as expected, if the work the kernels

		OpenMP	CUDA
128 x 96	(12288 px)	4.234, 52.9%	3.356, 41.9%
256 x 192	(49152 px)	5.575, 69.7%	10.29, 128.6%
384 x 288	(110592 px)	5.785, 72.3%	21.36, 267.0%
512 x 384	(196608 px)	5.941, 74.3%	29.44, 368.0%
640 x 480	(307200 px)	6.041, 75.5%	37.15, 464.3%
768 x 576	(442368 px)	6.055, 75.7%	39.82, 497.7%
896 x 762	(602112 px)	6.096, 76.2%	44.91, 561.4%
1024 x 768	(786432 px)	6.096, 76.2%	46.35, 579.4%

Table 3: Speedup and efficiency (in percentage) for different image resolutions. Efficiency is calculated with 8 cores.

have to undertake is minimal, the time to copy the memory to and from the CPU will be higher, making the use of the GPU not preferred.

Being SPP such an important variable, it was interesting to see what changes occur with the current implementations. Figure 6 shows a consistent performance increase. Having measured the number of ray bounces and objects in scene, would have probably resulted in similar outcomes (but, as stated earlier, these are not good variables to investigate).

Overall, parallelisation with the GPU gave the high performance increases that were expected, especially compared with CPU parallelisation through OpenMP.

5 Conclusion

This report presented a GPU parallelisation attempt of a ray tracer. The code used was beforehand changed to fit better the GPU architecture to allow a fair comparison between the various methods. The methods used were:

- OpenMP, used mostly to test wheter the code would correctly parallelise
- CUDA, to solve the problem through the GPU

The results obtained show that GPU parallelisation is indeed faster that its CPU counterpart. The exception is when the GPU has to perform minimal work: in such case the time to copy memory from/to the GPU will make the use of GPU not beneficial.

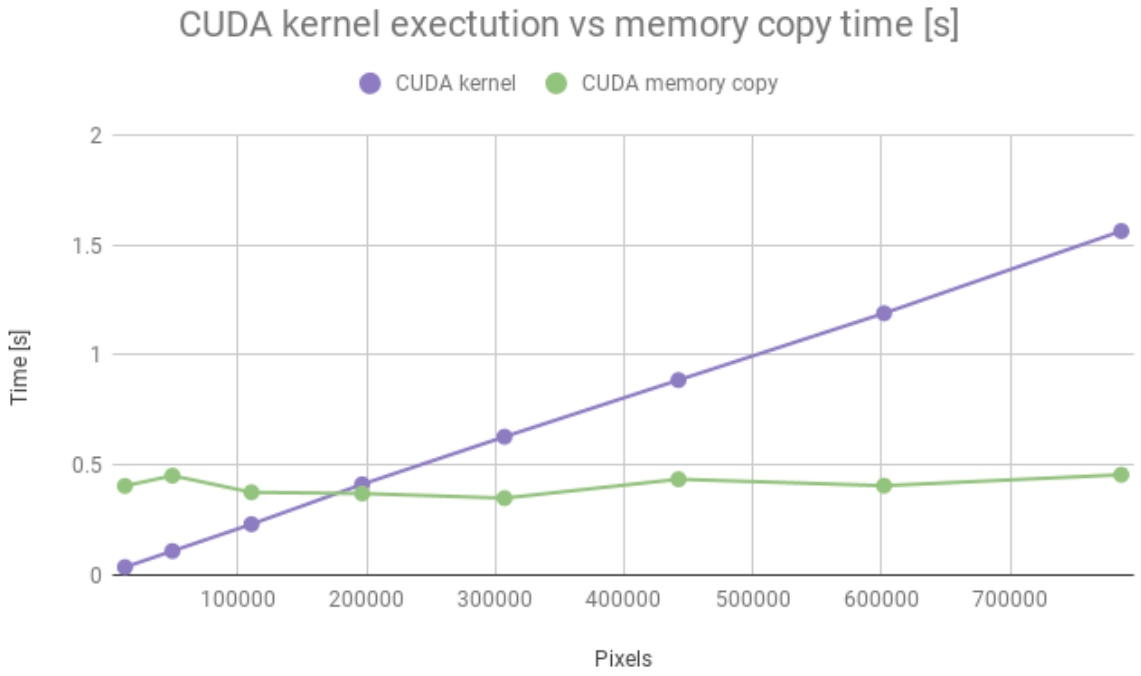

Figure 5: CUDA memory copy time vs kernel execution time with different image resolutions.

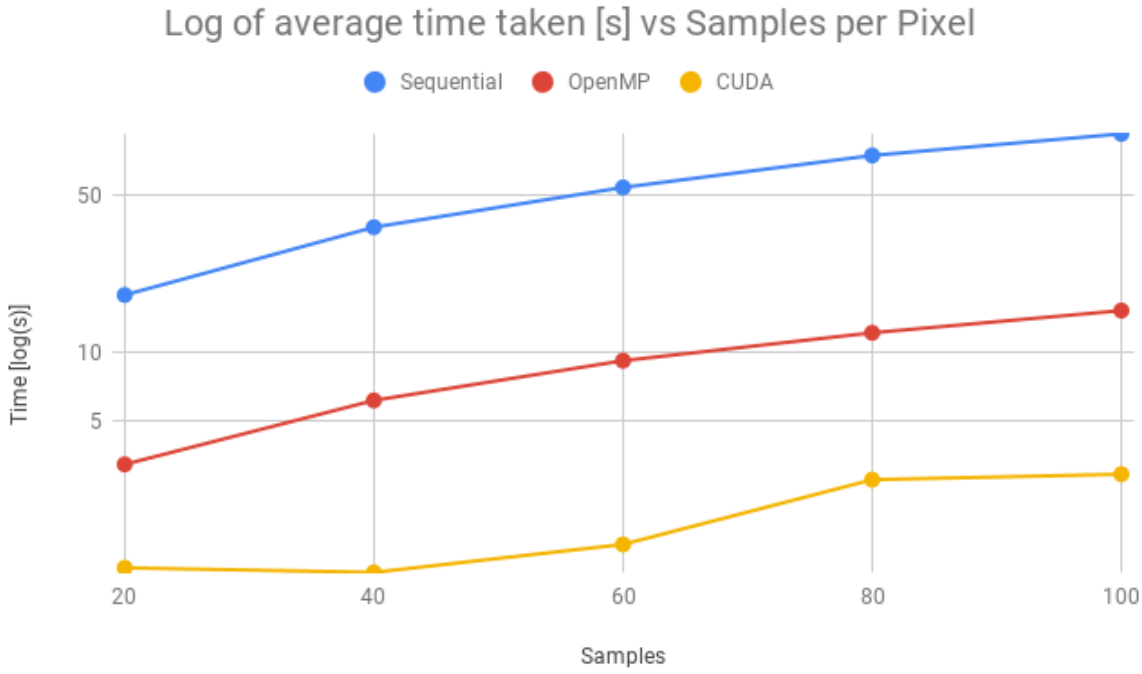

Figure 6: Comparison of results by method and samples per pixel.

There are different ways this investigation can be improved:

- Parallelisation of the SPP *for* loop can be attempted, either individually or in combination with the pixels loop.
- To allow higher SPP and resolution on the GPU, better memory management should be used. For example, the image can be divided in sections (e.g. stripes) and calculated one after the other.
- This ray tracer supports only basic functionalities (one material, spheres only, etc.). It may be worth to implement more advanced features.

References

[1] Beason, K. *smallpt: Global Illumination in 99 lines of C++*. `http://www.kevinbeason.com/smallpt/` (accessed 07/12/2018)

[2] Shirley, P. *Ray Tracing in One Weekend*. `http://in1weekend.blogspot.com/2016/01/ray-tracing-in-one-weekend.html` (accessed 07/12/2018)

[3] Lapere, S. *GPU path tracing tutorial 1: Drawing First Blood*. `http://raytracey.blogspot.com/2015/10/gpu-path-tracing-tutorial-1-drawing.html` (accessed 07/12/2018)