Developing a raytracer for GPUs

Albin Jonsson

Expected grade: A

Github: https://github.com/albinJoss/DD2360

Introduction

For the final laboration of the course DD2360 at KTH the students were tasked with creating a raytracing engine. This engine was to be very simple. It was to only handle one sphere at a time and only uses one light. Code for this was provided in python. This code however was only made to use the CPU. The students' task was to either keep this in python and then make a GPU version of it as well or to translate this implementation into C or C++ and then make a version running on the GPU.

Ray tracing is one of the pillars when it comes to computer graphics. Realtime ray tracing is one of the most performance demanding workloads and something which has been worked on for many years. Today Nvidia and AMD have both developed GPUs that have the required performance for this. The engine built for this assignment is however not done in real time. Ray tracing is used in 3D computer graphics and is a technique for modeling the light which is transported in a scene.[1]

Ray tracing can be achieved by using many different algorithms. The one used here is called the Blinn-Phong reflection model and is a modified version of the phong reflection model.[2] The Blinn-Phong algorithm is essentially an approximation of the phong algorithm as the latter is much more computationally expensive. In the Phong algorithm there is a need to continually recalculate the dot product of the reflection of the light beam on the surface as well as the distance to the viewer. In the Blinn-Phong algorithm the calculation is instead the halfway vector between the viewer and light source. The dot product of this halfway vector and the normalized surface normal is then taken which replaces the previous dot product. This dot product is then the cosine of an angle that is half of the angle of Phong's dot product, if all of the vectors lie on the same plane.

When Phong shading is applied it can be thought of as three separate light calculations. The first is applying the ambient lighting onto the surface. This is the lighting that is present everywhere in the image. The diffused reflection of the lighting source that may intersect an object is then applied. Lastly the specular reflection is calculated and added onto the object, this is also only from the light source/sources which might be present. When these three components are added together a very realistic lighting model has been created and can be used.[3]

Methodology

To create this software two versions of the same basic program were created within one cuda file. This being one to be executed on the device and one to be executed on the host. There were a few other modules which were also created. One of those being a .h file which linked this cuda file to a timer. This timer was made so that it could be compiled on both windows and unix operating systems without making any changes to the code.

The first step of the implementation was to create a few structs, these being to contain the data for the light, camera and sphere. This was chosen over having a lot of global variables to make it possible to change during run time, however that function is not implemented as of now. Next the host implementation was to be done first, and the first step to do that was to make some helper functions to perform the arithmetic functions. The arithmetic functions were addition, subtraction, the dot product, multiplying a vector with a scalar, and normalizing a vector. Another helper function was needed and was making sure that the values of the colors were between 0 and 255 while also converting from double to a 32 bit int. The reason behind needing all of these helper functions was that it was decided to use the double3 data structure instead of an array of doubles or ints. After this the same helper functions were made for the device version.

With all of those things set up the program first enters through the main function. Here it will allocate the memory needed for the sphere, light and camera. After that is done it will send these structures to an init function. In this function the value of these structures are set to the values which were chosen in the already provided python code. It will also create all timers which are used in the code. It will then call the run function which will be expanded on later on. After the run function is done, it will print the time it took for the total time it took to allocate everything and run the ray tracer. It will then write the result to a .bmp file to display the image. When that is done it will open up the .bmp file for the GPU and start the execution of the GPu part of the program. The first step is to allocate all the required memory on the device and initialize all of the values. After this it starts up the run_gpu function which is the same as the run function but adapted to parallel devices. it also starts the timers. When this is done it will return back and print the time it took to set everything up as well as the time it took to execute the kernels.

The run function is constructed of three parts. The first is initiating all of the variables to the right values, most often values which will be reused many times. After that is done it goes and creates all the function pointers which are needed. Lastly it goes into a nested for loop. In this for loop it first calculates the direction of the vector which points towards the pixel that it will evaluate. It then calls the trace function. It then updates which pixel it will point to and saves the value which was returned by the trace_ray function by calling the clip help function. The run_gpu function functions in a similar way but does not include the nested for loops or any for loop at all nor does it include the second part of creating function pointers. It also does not contain any for loops. It instead creates the variables needed, which are almost the same as in the run function, but done in a different way in some cases. It then calculates the direction and then calls the trace_ray_gpu function and saves that value to the array of ints using the clip_gpu helper function.

In the trace_ray function the first line of code checks if there is an intersection with the sphere, if there isn't it returns to the run function with a value of null in all three doubles. This gives us black in the background. If it however does intersect the sphere the function will calculate the color of that pixel using the Blinn-Phong algorithm which was previously explained. It will then return the color which was calculated. The exact same process is done with the device version of this function.

The intersect_sphere function and it's device equivalent are also very similar. In fact they are the exact same code. Their function is to check whether or not the ray will be in contact with the sphere in the specific pixel which the function that calls it is calculating. This will return two values as it essentially finds where a squared variable is zero, it then checks if both are above 0 and if so which one of them is the smallest. It then returns the smallest value to the trace_ray function.

Experimental set up

To run this program a computer running a Ryzen 5900x running at 4.8ghz locked on one CCX and 4.725GHZ locked on the other is used. The computer also has a RTX 3090 from Asus. This can't be locked to a single frequency sadly but it remains over 2.35GHZ (with the maximum being 2.5GHZ and average 2.428GHZ) at all times during the testing of this software. It is running in a stripped down version of Windows 10 with most of the services which are integrated in Windows turned off. It has access to two dual rank 16GB sticks of memory running at 3800CL14. This should provide the software with enough hardware power to stretch its legs and as windows is stripped down as much as possible it shouldn't be bogged down because of that. Both the CPU and the GPU have ample cooling and the thermals will not become a problem at all during the execution. The program was compiled using nvcc. Level zero, one, two and three optimizations were tested to see how much of an impact they would have on the host code.

Results

The validation was kept to a very simple test. It was only checked whether or not the two versions matched each other and that was followed by a visual check.

	Optimization level 0			
	CPU:		GPU:	
	execution	total	execution	total
MIN:	5858,0876	5861,6036	6,1191	144,4172
AVG:	5838,4645	5838,4645	6,2485	112,6343
MAX:	5861,6036	5904,1528	6,1574	118,8248
	Optimization level 1			
	CPU		GPU	
	execution	total	execution	total
MIN:	1154,8402	1149,2834	6,1191	117,4511
AVG:	1146,6759	1157,2526	6,1301	98,6798
MAX:	1458,5043	1461,4248	6,175	103,3034
	Optimization level 2			
	CPU		GPU	
	execution	total	execution	total
MIN:	863,7195	866,2591	6,1954	111,8329
AVG:	863,2193	872,2845	6,2888	105,8329
MAX:	869,5933	866,278	6,4361	105,5465
	Optimization level 3			
	CPU		GPU	
	execution	total	execution	total
MIN:	883,5884	868,351	6,1266	109,4101
AVG:	867,3896	869,863	6,1358	94,8879
MAX:	866,1621	886,2676	6,3153	113,6426

Table 1: A comparison between the optimization levels in the compilers and the time (in milliseconds) it took to execute the kernels and the time to do everything when it comes to that part of the program.

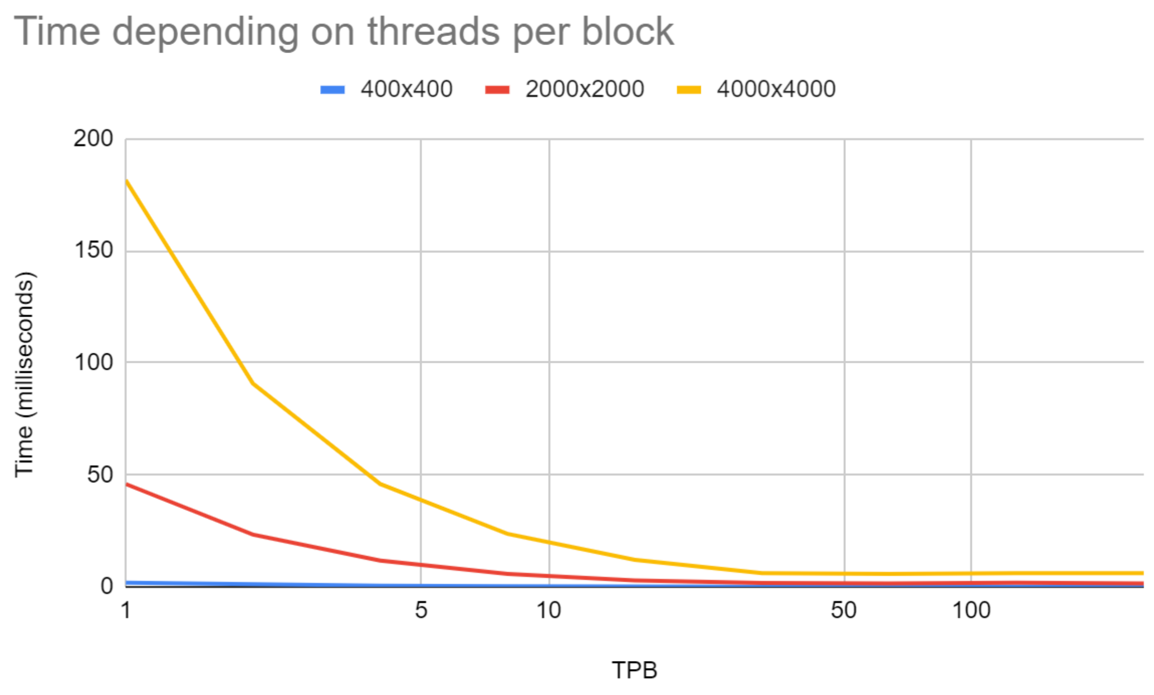

Figure 1: Testing the time it took depending on the threads per block

Discussion

The first and largest optimization which was done to the code which was provided was to write it in cuda C. This will in the end give it much better performance than it would get if it was to run on an interpreted language like python. The choice was also made to instead of using the standard square root function to calculate the normal an approximation which was first used in the game Quake 3 would be used to calculate the inverse square root. This algorithm is called fast inverse square root. However it’s originally used with floats. As such it had to be adapted to work with doubles. Luckily this had already been done and there were resources online to do this. A further optimization can however be made to the way that the trace_ray function handles the normalization of it’s vectors. To do this AVX instructions would be used and as such this optimization was not done. In this function there are a total of four calls to the normalize function. This would mean that a total of 12 doubles are used. This also means that the functions require the calculation of four 64 bit square roots. This could be batched together and as such a total of 256 bits would be used. The AVX instruction set was created for just a task like this. Using the AVX instruction would be a bit slower to calculate one number but this would provide a much faster way to calculate multiple square roots.[4] In source four it is stated that the standard way to compute the inverse square root is much faster than the fast inverse square root algorithm. This however is only true for floats as it adds about 400ms of execution time when running 4000x4000 pixels with optimization level zero compilation.

Another optimization which was made possible by using cuda C is using function pointers. This does not provide as large of an optimization as using the fast inverse square root algorithm. It does however provide a little bit of a performance boost. This is only done for the host code however. Another obvious optimization which was done was that all the values which would be constant for all of the loops were calculated in the beginning of the functions and used thereafter. This is especially important when it comes to divisions as this is a very expensive operation to perform. It takes about 12-24 times longer to perform a division than it does to perform a multiplication[5] and as such it’s avoided as much as possible.

There are also some other optimizations which could be done with the algorithm. The most relevant for this task is to actually do the whole ray tracing process backwards. Instead of starting out at the ray and working towards the viewer, it would start at the viewer and move to the ray. This would avoid a lot of unnecessary computation as it would only calculate the things which are visible to the viewer. This would avoid getting a little bit through the calculation to then interrupt the calculations after getting done with almost the whole intersect sphere function. If the scene was more complex there would be more optimizations to do. SUch as using a KD-tree and bounding volumes.

Another change which would be very helpful when it comes to performance is to optimize how many if statements are used. This would particularly help the performance of the device section. This is as GPUs are SIMD devices or rather each warp is. This means that if the different sections have different outcomes and need to execute different instructions this will devastate the performance. In the RTX 3090 each SM has 4 warp[6] schedulers and as it will most likely be a large bottleneck.

Conclusion

The optimization of the performance for this ray tracing engine is in reality very poor and given a bit more time it could most likely be reduced by a lot, this is especially true for the host code as this could most likely be done in half the time with optimization level zero. In general this performance is very poor. Using shared memory could have helped but not as much as some other optimizations. Another optimization would be to use asynchronous memory which would most likely improve the performance of the GPU the most. Sadly this was not implemented before the deadline of this project.

As can be seen in Figure 1 multithreading is a tremendous help when it comes to doing ray tracing. This despite the fact that the GPU doesn’t use nearly all of the resources available as it focuses all the calculations on the fp64 and the fp32 parts of the devices are very underutilized. The performance of this application is very lacking and is something which should be improved upon. This ray tracing engine could however also be extended when the performance has been improved to also include different kinds of objects as well as more kinds of lights and ray tracing algorithms. This kind of engine can be found for example in the ray tracing in one weekend course/book, which I read and used to prepare for this assignment.

References:

1 https://en.wikipedia.org/wiki/Ray_tracing_%28graphics%29
2 https://en.wikipedia.org/wiki/Blinn%E2%80%93Phong_reflection_model
3 https://en.wikipedia.org/wiki/Phong_reflection_model
4 https://www.linkedin.com/pulse/fast-inverse-square-root-still-armin-kassemi-langroodi
5
https://stackoverflow.com/questions/2858483/how-can-i-compare-the-performance-of-log-and-fp-division-in-c
6
https://images.nvidia.com/aem-dam/en-zz/Solutions/geforce/ampere/pdf/NVIDIA-ampere-GA102-GPU-Architecture-Whitepaper-V1.pdf

Ray tracing in one weekend:
https://raytracing.github.io/books/RayTracingInOneWeekend.html