A simple GPU ray tracer: Project report DD2360

Jacob Wahlgren <jacobwah@kth.se>
Expected grade: A

January 11, 2022

1 Introduction

Ray tracing is a 3D rendering technique enabling realistic optical effects by simulating light rays. It is often used for visual effects and animation in film and television. Using ray tracing is computationally expensive since a ray has to be simulated for each pixel. However, since the rays are independent the problem is embarrassingly parallel and easy to accelerate using GPUs.

A ray tracer casts a ray from the camera, through each pixel, into the scene. The intersection points of the ray and all objects are computed and the point closest to the camera is used to color the pixel. Material properties, reflection angles, light sources etc are used to compute the final color.

2 Methodology

We used the Blinn-Phong shading model, where the color C at an intersection point is given by the following formula. Let C_a be the ambient color, D the amount of diffuse lighting, N the normal, L a normal vector pointing to the light source, C_m the material color, c the amount of specular lighting, C_l the color of the light source, H the normal vector halfway vector between the ray and the light source, and k the specular exponent.

$$C = C_a + C_m \cdot D \cdot \max\{(N, L), 0\} + C_l \cdot c \cdot \max\{(H, N), 0\}^k \tag{1}$$

We render a scene with a single light source and a single solid sphere. The resulting image is written to a file in the PPM format using 3 bytes per pixel. Each pixel is computed by one thread in square tile blocks of configurable size.

The kernel is similar to the Python code in structure and content. The thread and block indices are used to calculate the pixel coordinates. Operator overloading on the `float3` type is used to simplify vector operations. The color is computed in $[0, 1]$ float space and at the end converted to $[0, 255]$ int space.

Since the task is IO bound, we investigate three different techniques for writing the image to file. The **fwrite** version is the simplest, where the whole image is rendered, then copied to RAM, and then written to file with a single

GPU	Implementation	Execution time (s)
NVIDIA Quadro K240	streams (pinned)	0.95 ± 0.38
NVIDIA Tesla K80	fwrite (pinned)	1.32 ± 0.48

Table 1: Fastest configurations at 10,000x10,000 pixels.

write call. The **mmap** version truncates the file to the output size and then memory maps the whole file. Once the image is rendered it is copied directly into the memory mapped file. The **streams** version uses multiple overlapping streams to render, copy, and write chunks of the image in parallel. Each chunk corresponds to a horizontal line of blocks in the image.

The code is available at `https://github.com/jacwah/cuda-raytracer`.

3 Experimental setup

Experiments are run on the Tegner cluster at the PDC Center for High Performance Computing at KTH. These nodes have two 12 core Intel E5-2690v3 Haswell processors with 512 GB RAM. Files are written to a parallel Lustre file system. The performance is evaluated both on nodes with NVIDIA Quadro K240 and the NVIDIA Tesla K80. The GPUs are programmed using CUDA.

The reference CPU implementation in Python was modified to call `savefig` to write the image to a file instead of displaying it interactively.

4 Results

Visual inspection of the generated images using the `display` command validated the results. For the largest image size ImageMagick was used to resize the image before viewing. Example images are shown in figure 2.

The performance comparison of the CPU and GPU versions showed that the GPU vastly outperforms the CPU version on this task. The difference was more noticeable at larger image dimensions. In fact, results were not obtained for the CPU version above image dimension 1000 since it was too slow. The results are presented in figure 1, were the Quadra K240 fwrite 8x8 configuration is used to represent the GPU.

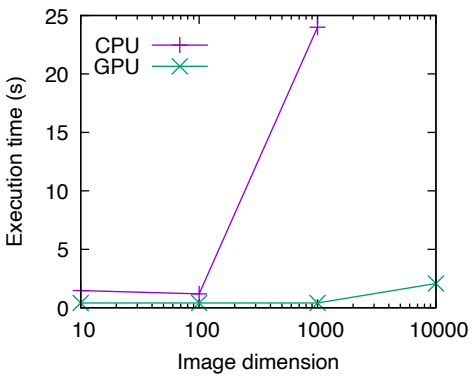
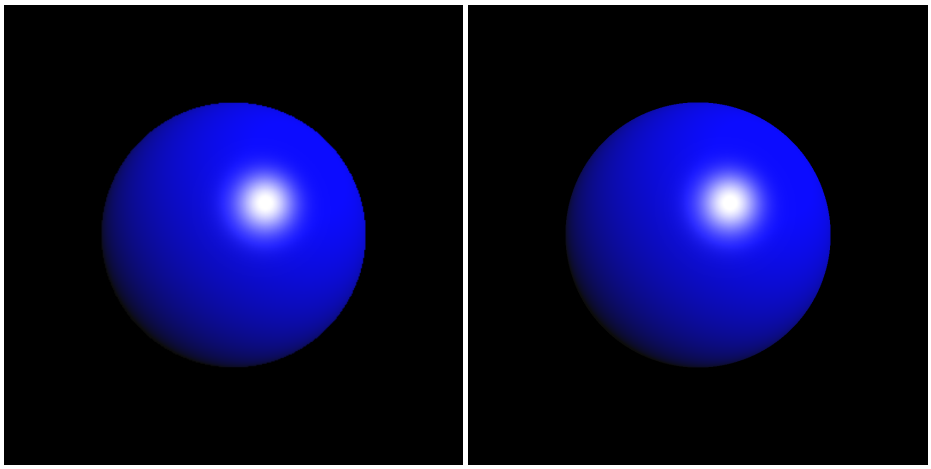


Figure 1: Comparison between Python CPU and fwrite 8x8 GPU, average of 10 runs, on Tegner thin node.



(a) CPU (b) GPU

Figure 2: Sample output images.

We evaluate the performance of different GPU configurations at image size 1000x1000 (normal) and 10,000x10,000 (huge). The image files are 3 MB and 287 MB respectively.

At the normal image size, all configurations perform similarly to each other. The K240 is consistently faster. Figure 3 shows the results for all configurations.

At the huge image size, the results were more varied. On the Quadro K240 the fastest configurations were streams (pinned) with 8x8 or larger block size. Also close were fwrite (pinned) with 8x8 block size. On the Tesla K80 the fastest configuration was fwrite (pinned) 16x16. Also close were regular fwrite 16x16 and streams 8x8 and 16x16. The execution time of the fastest configurations are presented in table 1. Figure 4 shows the results for all configurations.

5 Discussion and conclusion

The massive parallelism offered by the GPU vastly outperforms the reference CPU implementation. However, a more fair comparison would use a faster language than Python and utilize all the available compute power of the CPU rather than a single thread.

The specular highlight in output images from the CPU and the GPU versions are slightly different. This is likely caused by using different floating point precision (double in Python, single in CUDA).

When it comes to comparing the various GPU implementations, the differences at a normal image size of 1000x1000 pixels are negligible. The reason could be that a configuration independent overhead, such as initializing the CUDA context, dominates the run time.

In contrast, for the huge image size of 10,000x10,000 pixels the differences

are very large. The various implementations perform differently on the two GPUs used. On the K240 usage of pinned memory clearly improves the results for fwrite and streams which both perform well. On the K80 fwrite performs well both with and without pinned memory, while streams performs well only without pinned memory.

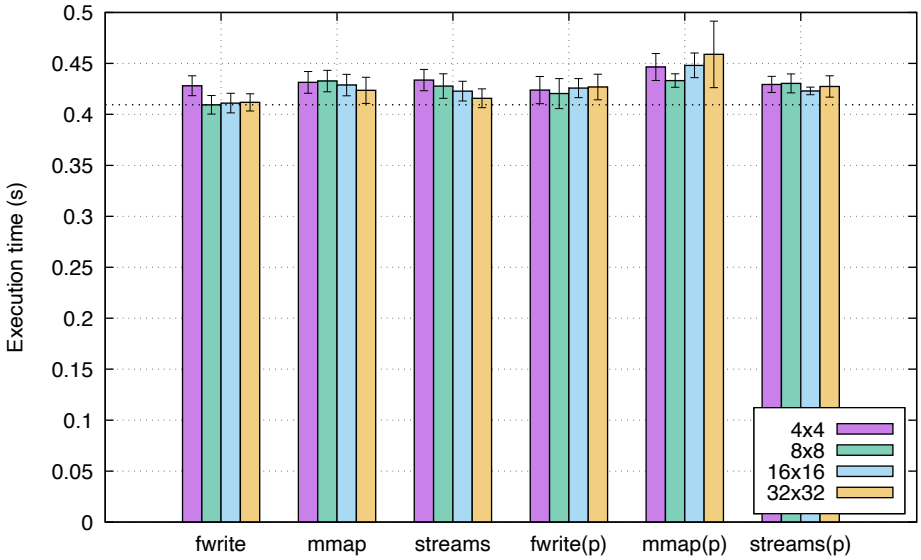
The mmap implementation has very large variations in run time, and is overall the slowest. The results indicate that the overhead of memory mapping is not worth it, since each byte is only accessed once. In the words of Linus Torvalds, "playing games with the virtual memory mapping is very expensive in itself"[1].

In conclusion, ray tracing is a task that can greatly benefit from GPU acceleration. Depending on which GPU is used various techniques can be used to reduce IO cost.

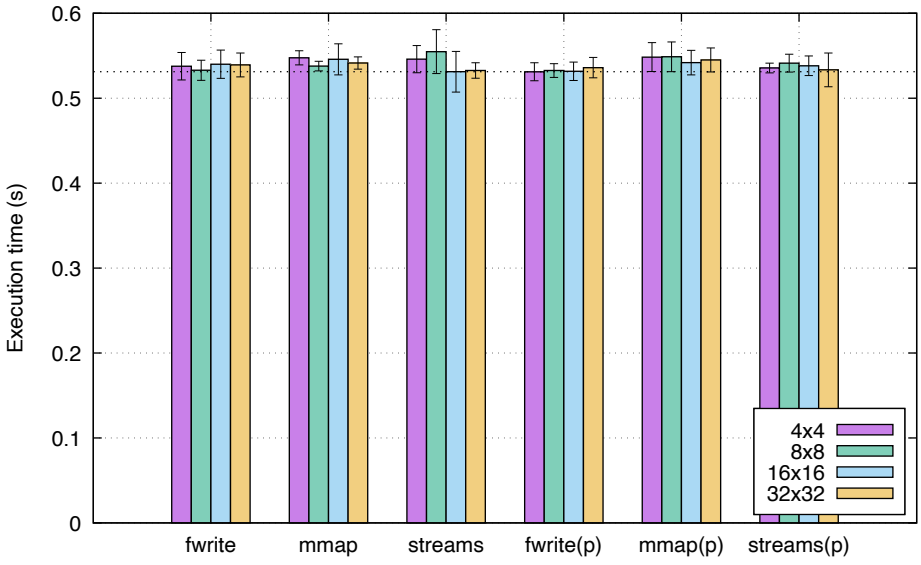
Future work could investigate how the streams implementation can be tweaked to increase performance. What chunk sizes and number of streams are optimal? Why does the pinned version perform poorly on the K80? On a parallel file system, is it better to buffer the writes?

A more complex scene and shading model could also change the performance characteristics, since the computation would occupy a larger fraction of the run time. How would this change the behavior of the different implementations?

[1] `https://yarchive.net/comp/mmap.html#6`

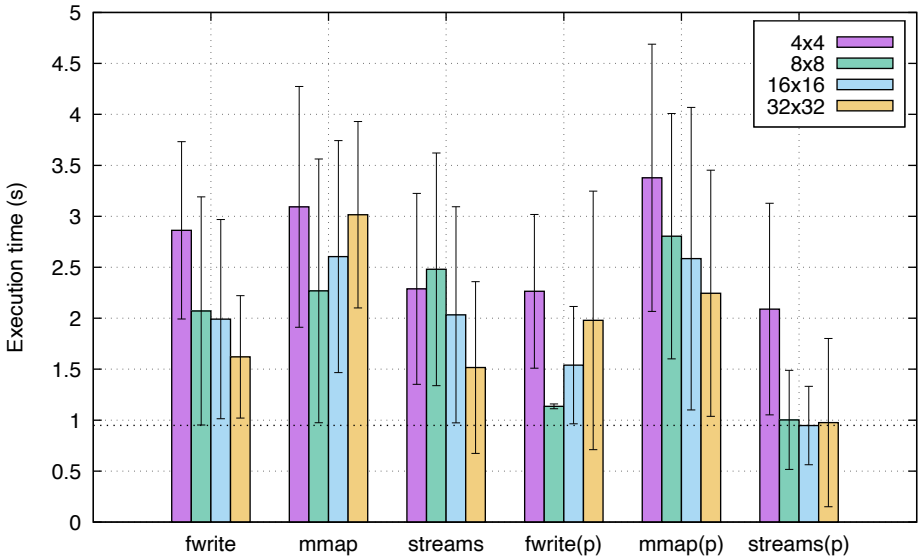


(a) NVIDIA Quadro K240

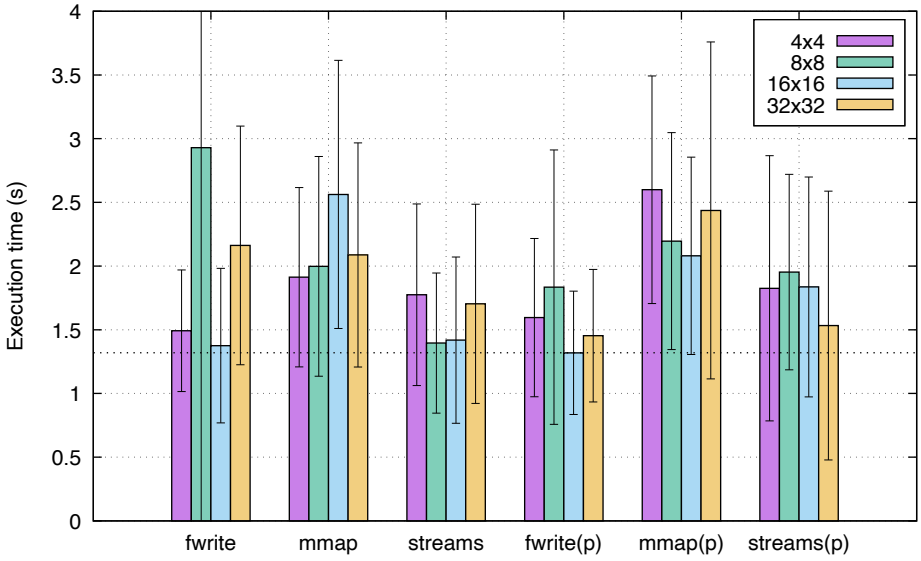


(b) NVIDIA Tesla K80

Figure 3: Performance results for 1000x1000 image with various block sizes. Bars show average, whiskers show standard deviation. Usage of pinned memory is indicated by (p). The black dotted line shows the minimum value.



(a) NVIDIA Quadro K240



(b) NVIDIA Tesla K80

Figure 4: Performance results for 10,000x10,000 image with various block sizes. Bars show average, whiskers show standard deviation. Usage of pinned memory is indicated by (p). The black dotted line show the minimum value.