Studying and profiling the GPU performances of 3D ray-marchers

HUYNH Danh Chieu Phu
Team MAVERICK, LJK-INRIA
Grenoble, France
danh-chieu-phu.huynh@etu.univ-grenoble-alpes.fr

Abstract

GPU shaders look like threaded C, but hardware constraints (SIMD, warps, groups, small caches, slow memory access) hugely impact performances on unintuitive ways. In the case of ray-marchers, neighbor rays having different fate locally breaks the parallelism, at places where the algorithm is already the slowest (e.g. near silhouettes).
The target of this research is to study various simple test case and implementation variants, getting cost heat-map and timing profiling, to understand how bad some cases can be, interpret what happens, and get some good practice recommendations.

1 Introduction

Ray-marching is a rendering method which is applied widely in game programming because of its high performances which are very important in real-time rendering.
However, this rendering method still has some weakness which combine with hardware constraints may slow down the game performances.
Our purpose is to do a number of experiments to understand how GPU works when rendering a ray-marching scene, and how is the cost structure.

1.1 Previous works

A GPU is designed to control multiple threads which are organized into multiple parallel thread blocks. A streaming multiprocessor (SM) executes one or more thread blocks. The SM executes threads in groups of 32 threads called a warp. The screen is divided into many tiles, each tile has 4 × 2 = 8 warps called block and all warp in the same block will start at the same time.
There are many GPU profilers available but most of them support only Windows platform and have a many inconvenience points (such as cannot provide the exact rendering time of each pixel or cannot show quick parts of the scene that cost more than the others). Therefore in this research, we will develop a few tools to do the experiments.

1.2 Ray-marching

Ray-marching is a variants method of ray-casting, a rendering method which calculates the intersections point of the "view-ray" with the objects' surface and base on them calculate the object's diffuse color in the 3D screen.
With ray-marching, the shading program needs not compute the intersections point but simply walks along the ray and checks each step whether it hit the objects. *Figure 1* shows the idea of ray-marching.

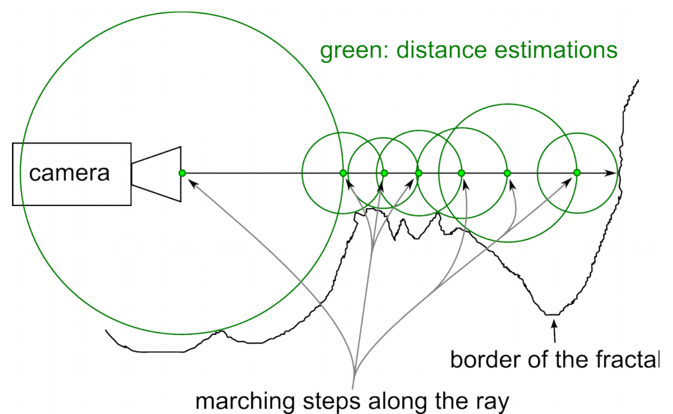

Figure 1: The idea of Ray marching

The larger the distance the program walks each step, the faster it moves but the less accurate it is. To avoid "step-over" an object, the program will try to reduce the distance when it gets close to the object. Therefore it may take many steps when the ray comes closer to the object's silhouettes. This is one of the main reason that causes the performance reduction. Which will be investigated in this report.

2 Our approach

We will applies ray-marching plus some injected analyzing shading code into a simple scene to create cost heat-maps and timing profiling of the scenes. By analyzing the output information, the research may figure out how much reduction each case may cause, interpret what happens, and propose some recommendations to walk around.

For the experiment, the shader code will be written on Shadertoy.com and C++ native program then will be analyzed by Octave script and C++ native program.

The difficulties

GPU shader is a closed process whose the output is only a color vector of 4 float number RGBA of a pixel on the screen and no other output channel.

Moreover, the color buffer's precision is very limit which may loose the precious information.

To overcome this difficulty we will propose a strategy to encode the desired information into the pixel color then display it to the screen. By analyzing the output image we may retrieve back that information. Detail method will be represented with the below correspondent experiments.

3 Our experiment

We will setup a simple scene for the experiment: the scene is made by 2 spheres (a big one overlaps a smaller one) whose silhouettes overlap each other and 1 light source. The scene doesn't have any other effects (no texture access) to reduce any unnecessary effect that may affect the complexity and the result of experiment.

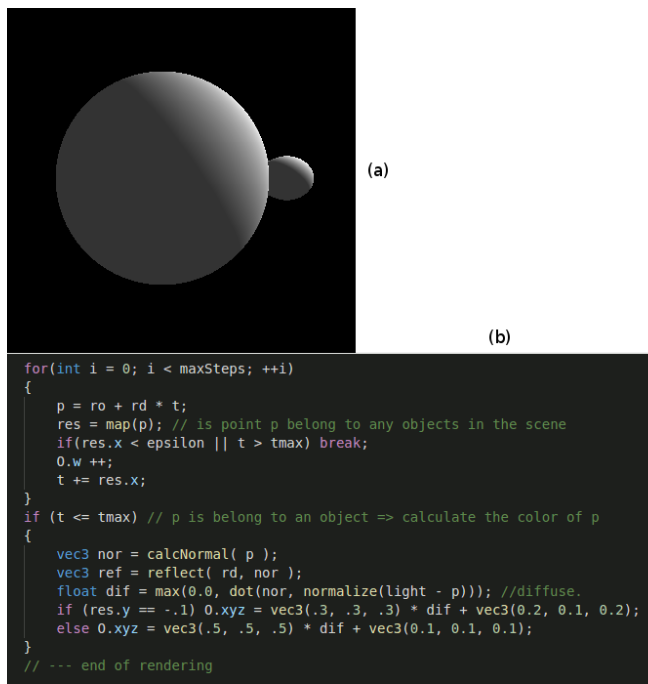

Figure 2: (a)-Scene of the experiment (b)-Ray-marching shader code

3.1 Algorithmic cost analyzing

Algorithmic cost of a pixel is the number of loops needed to detect the intersection between the view ray and an object.

Experiment's steps:

Step-1: Add a variable to count the loop then assign it to the output color of the fragment shader. The output image of this step is the heat-map whose the lighter pixels are the one cost more time.

Step-2: Use Octave to calculate the histogram of the image and base on this create the probability distribution diagram.

In the Heatmap in *Figure 3* (a), the pixel which are brighter are the ones that cost more intersection test than the others.

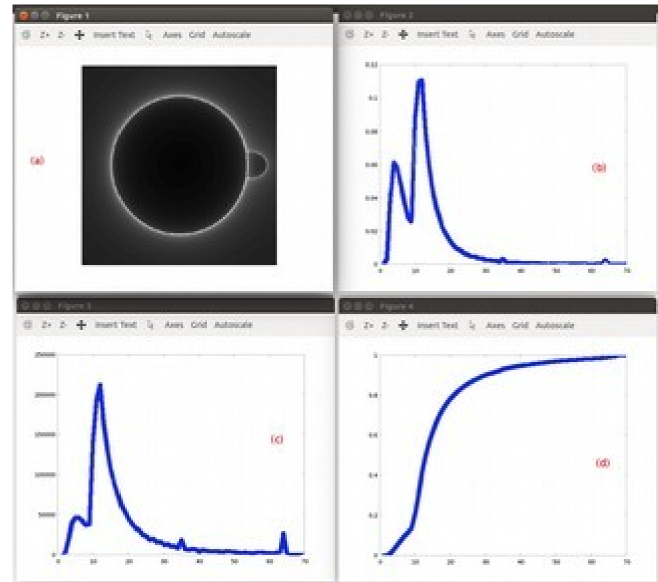

Figure 3: (a)-Heatmap of loop (b)- Percent of cost (c)-Histogram (d)-Cumulative distribution function

From the (d) Cumulative distribution function, we can see that the slot reaches 80% on Y axis (the number of intersection test) when it got only 20% on X axis (number of pixel). We can come to the conclusion that 80% of intersections test is used to render only 20% of the pixels which are the pixels around the object's silhouettes.

The reason is when the view-ray comes close to an object's surface, it has to "walk" a smaller step to avoid step-over the surface – unable to detect the intersection.

3.2 Time cost analyzing

The experiment above considers only how much marching step a program would take to render a pixel. In fact, there are other reasons that may reduce rendering time.

We want to know the GPU's efficiency in this case. If the time cost matches exactly with the algorithmic cost we can say that GPU's efficiency is high. Other wide, there should be a problem to be investigated.

The research will use the OpenGL extensions provided by Nvidia to measure the elapsed time for rendering each pixel.

Experiment's steps:

Step-1: Call function *clock2x32ARB()*[1] before and after RenderSpheres() to get start time and end time in ***tick***.

Step-2: Subtract end time by start time to get elapsed time.

---

[1]GLSL function that retrieves the GPU time and store into a vector of 2 integers. URL can be found in the reference.

Step-3: The time value is stored in a vector 2 of integers. Because the time cost for a pixel in this experiment is short and suppose to be fit in the lower vector, we will ignore the higher one. This value will be assigned to the Blue color of the fragment shader. The pixels which got more Blue are the one which takes more time for rendering. The output image will be like in *Figure 4*.

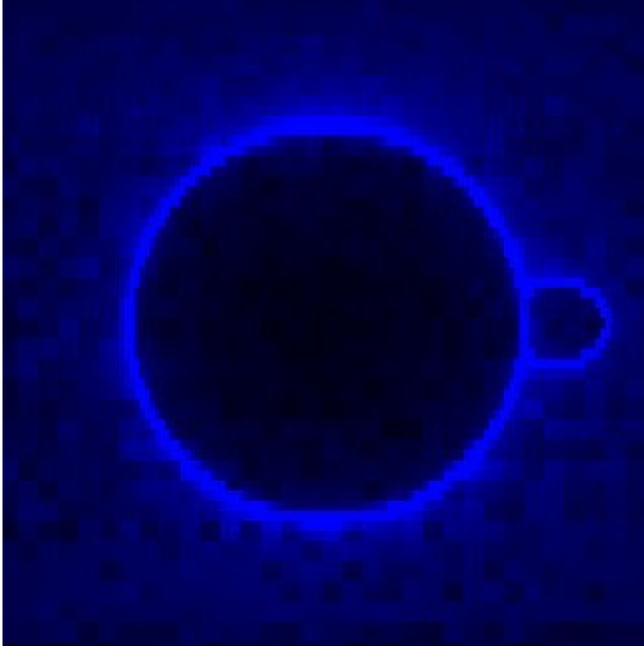

Figure 4: Heatmap of rendering time

From the image, we can notice also that the pixel closed together would take the same time to rendering even they are on the different parts of the scene.

This synchronization look similar to the physical organization of the GPU: **those pixels are rendered by the same warp,**

3.3 GPU organization and Time cost

By using Nvidia's extension *GL_NV_shader_thread_group* we can identify that the GPU used in this research got 7 SMs and each SM got 64 logical warp units. This extension also provides us id of the SM and the warp that each pixel belongs to. So we expect that if we can get the information of warp id and SM id we can confirm the question which was risen in section 3.2.

Experiment's steps:

Step-1: Insert 2 lines of code to get the id of warp and SM and assign them to the color Red and Green of the fragment shader.

The output image will be like *Figure 5*

Step-2: Develop a program reads the colors of this image and generates a timetable storing the following information:

List of SM (0 → 6)
- List of Warp (0 → 63)
  - List of warp call (0 → max call)
    - Start time
    - End time

Step-3: Based on the timetable, draw a timeline diagram of all SMs and warps.

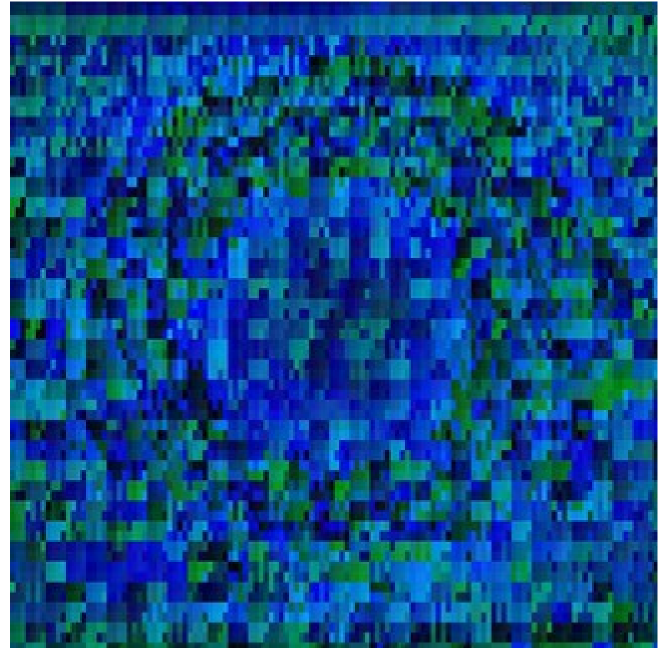

Figure 5: Heatmap with SM ID and warp ID

The timeline we have in result is very synchronized. Most of the warp start and end at the same among of time. This is because the problem of precision.

3.4 The problem of color buffer precision

By doing an acid test, we can see that the precision of color buffer is very limited.

Output value in fragment shader	Value receive in the color buffer
0.111	**0.1**09804
0.345	**0.3**45098
0.346	**0.3**45098
0.347	**0.3**45098
0.548	**0.5**49020
0.549	**0.5**49020
0.550	**0.5**49020
…	...

The accuracy is guaranteed from 0.1 or above. That means with 1 color value, we can encode a digit (0 → 9). In a warp, we have 32 pixels × 4 color = 128 digits, which is far enough to encode SM id, warp id, start time and end time.

Be cause the information of rendering is very important to making conclusion about GPU's efficiency, we must keep all the digits. To do that we propose a strategy like below:

Encode and decode strategy:
We use only 8 first pixels of a warp to store the information

R = SM ID G = warp ID B = 0.1 A = 0	R, G, B, A = 4 higher digit of **start time**	R, G, B, A = 4 lower digit of **start time**	R = SM ID G = warp ID B = 0.2 A = 0
R = SM ID G = warp ID B = 0.3 A = 0	R, G, B, A = 4 higher digit of **end time**	R, G, B, A = 4 lower digit of **end time**	R = SM ID G = warp ID B = 0.4 A = 0
No use	No use	No use	No use
…			

With this strategy, we redraw the timeline and the result will be like in *Figure 6*

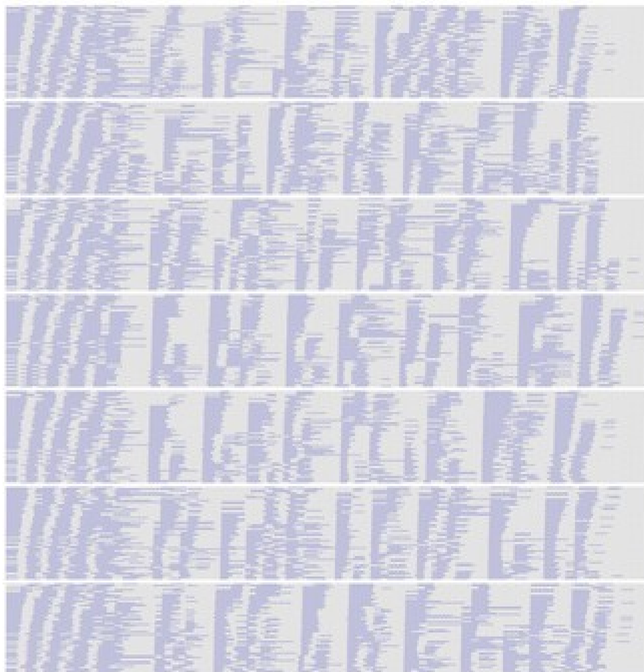

Figure 6: Complete percision timeline

By doing statistic on the rendering and idle time we can see that 57% of rendering time is idle, which means the GPU's efficiency isn't high as expect.3.5 Determine rendering tiles

We know that the screen is divided in to tiles. Using an image processing application, we can determine those tiles by picking the pixels which have the same SM id.

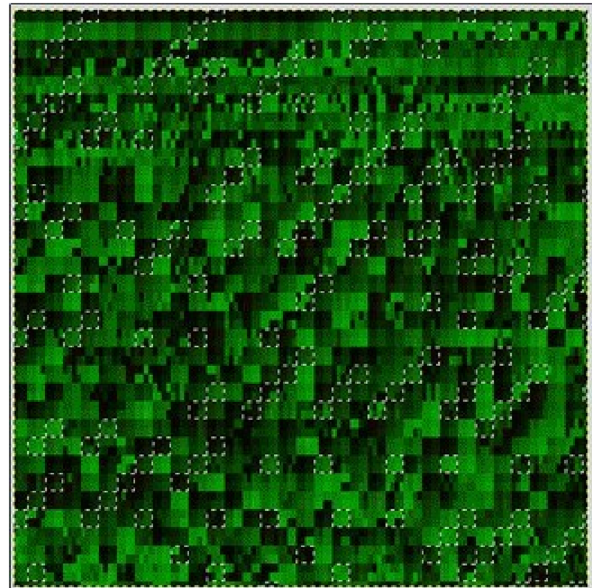

Figure 7: The blocks of SM-5

On the GPU using for the experiment, there are 7 SMs, divided into 4 group of blocks: SM-0 and SM-4, SM-1 and SM-5, SM-2 and SM-6, and SM-3.

By developing an algorithm assigning the color to each warp in the timeline, the warps in the same block and start at the same time will be assigned the same color, we can see clearly the wrap in a block, and when they start and end like in *Figure 8*

Figure 8: Timeline with block info

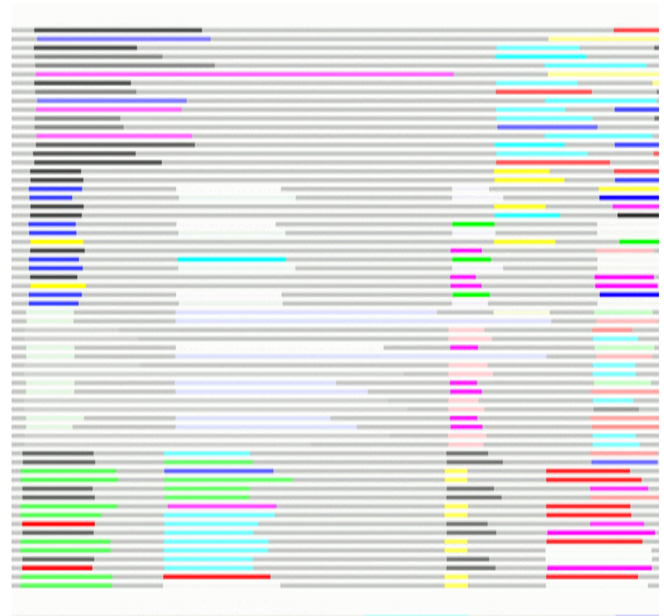

Figure 9: A case of breaking parallelism

4 Conclusion

So far, we developed testing tools and scripts. We did some experiments. We made early analyzing base on that.
From the result of the experiments we can come to the conclusion:

- For ray-marching shader, the rendering cost of a pixel at the object's silhouette is very high comparing to other regions of the screen. This is true in term of both the algorithmic cost and the time cost.

- Because all the warps in the same block will start at the same time, if all 7 warps finished their task they still have to wait for the last one before starting any new task. This breaks the parallelism of GPU and reduces frame rate. *Figure 9* is a zoomed-in view of a case like this.

Therefore, the rendering time of a pixel near object's silhouette is not only longer than other pixel but also delays the whole warp and even more delay the whole block of 8 warps = 32 pixels
This is the reason why a few complex pixels may hugely impact the rendering performances.

5 Further research

The problem of color buffer precision is the main difficulty that slow down the researching process. By figuring out an efficient method to encode the information in section 3.4 allow us to advance into more complex cases such:

- Try to figure out the cause of the gaps (idle time) appeaser between each warp render call.
- A scene with complex objects
- Improving the visualization of the timeline.

References

[Shadertoy] Shadertoy.com

[Adok] Adok. On ray casting, ray tracing, ray marching and the like. Website *http://www.hugi.scene.org*

[Haugo, 2013] Simen Haugo. Raymarching Distance Fields. Website *http://9bitscience.blogspot.fr*, July 2013.

[Khronos] OpenGL Extension, ARB_shader_clock. URL: *https://www.khronos.org/registry/OpenGL/extensions/ARB/ARB_shader_clock.txt*

[Neyret and Crassin] Understanding G80 behavior and performances, Research report, LJK-INRIA. 2008

[Nvidia] NVIDIA's Next Generation CUDA™ Compute Architecture: Fermi™ – Whitepaper

[Nvidia] NVIDIA GeForce GTX 980 Featuring Maxwell, The Most Advanced GPU Ever Made - Whitepaper

[Nvidia] NVIDIA GeForce GTX 750 Featuring Maxwell, The Most Advanced GPU Ever Made - Whitepaper

[Celarek] Adam Celarek, Real Time 3d Mandelbulb, Website *http://celarek.at/tag/ray-marching*