Modelling Gravitational Lensing on a GPU

Yonggang Li (18043158)

**Abstract**

This report is about modeling Gravitational Lensing with Graphics Processing Unit (GPU) by applying “inverse ray tracing" technique, which is computing effective and adaptive to both simple and very complex lens configurations. The program is built with NVIDIA CUDA parallel computing platform and API (version 7.5). The GPU card will be introduced first and followed by the design of the program. Images for different resolutions and different sizes of the lensing system are generated.

**GPU card**

There is a GeForce GTX 470 GPU card is installed on the workstation in Computer Vision Laboratory of Massey University. Here are some outputs from running device query utility from CUDA sample bin/x86\_64/linux/release/deviceQuery

Device 0: "GeForce GTX 470"

CUDA Driver Version / Runtime Version 7.5 / 7.5

CUDA Capability Major/Minor version number: 2.0

Total amount of global memory: 1279 MBytes (1341325312 bytes)

(14) Multiprocessors, ( 32) CUDA Cores/MP: 448 CUDA Cores

Total amount of constant memory: 65536 bytes

Total amount of shared memory per block: 49152 bytes

Warp size: 32

Maximum number of threads per multiprocessor: 1536

Maximum number of threads per block: 1024

Max dimension size of a thread block (x,y,z): (1024, 1024, 64)

Max dimension size of a grid size (x,y,z): (65535, 65535, 65535)

…

The device information shows that there are 448 CUDA cores which mean 448 threads can be executed concurrently. Since there can be maximum 1024\*65536^3 threads and the largest lens image is 1601x1601, we can assign one thread to process a single pixel for simplicity.

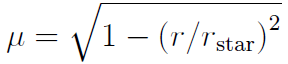
**Inverse Ray Tracing and Limb Darkening**

Lens plane is divided into pixels and a ray is shot from each pixel position. The corresponding position on the source plane is then calculated directly from the lens equation. If the ray shot from a particular pixel happens to land on the source star, then that pixel forms part of the gravitationally lensed image of the star.

As the stars appear darker towards the edge, this "limb darkening" should be taken into account by weighting the lens image pixel according to where the ray lands within the source star disc.



where



r is the distance from the centre of the star where the ray lands, rstar is the radius of the star, and λ is the limb darkening coefficient, use 0:5 typically.

**Program Design**

The program flow on the host is pretty clear and simple. First a 2 dimension float array is to store the brightness of each pixel of lens plane, zero means that the pixel is outside of the source star disc.

Array<float, 2> lensim(npixy, npixx);

The lens plane is divided into 801x801 and 1601x1601. As the pixel number are much smaller than the threads allowed, a thread will be created for calculating one single pixel.

int threadsPerBlock = 1024;

int blocksPerGrid = lensim.ntotal / threadsPerBlock + 1;

Copy the lens configurations to GPU space before calling the kernel

float\* d\_xlens = copy\_array\_to\_device(xlens, nlenses);

float\* d\_ylens = copy\_array\_to\_device(ylens, nlenses);

float\* d\_eps = copy\_array\_to\_device(eps, nlenses);

float\* d\_lensim = copy\_array\_to\_device(lensim.buffer, lensim.ntotal);

Now, it is ready to call the kernel for computing each pixel

cuda\_shoot<<<blocksPerGrid, threadsPerBlock>>>(d\_lensim, npixx, npixy, lens\_scale, d\_xlens, d\_ylens, d\_eps, nlenses);

Copy back the lens data from GPU space and dump to a fits file

cudaMemcpy(lensim.buffer, d\_lensim, sizeof(float)\*lensim.ntotal, cudaMemcpyDeviceToHost);

dump\_array<float, 2>(lensim, "../lens\_x.fit");

Figure 1 show how the kernel works.

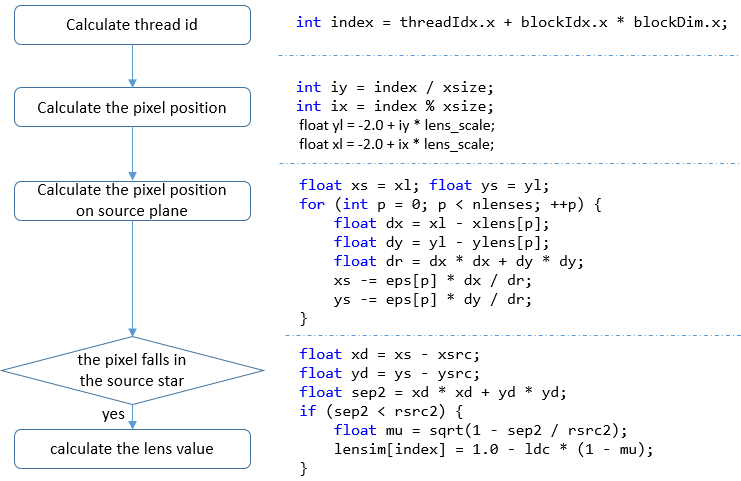


Figure 1 Flow char of the kernel

**Testing and Result**

Two lens image resolutions, 801x801, 1601x1601, combined with 1, 2 3 and 100 lens are tested in both sequential and GPU enabled parallel programs. From Table 1 we can see as the lens increase, so does the time for computing. On the contrary, GPU enabled parallel processing keeps a stable time for each resolution group.

|  |  |  |  |
| --- | --- | --- | --- |
| resolution | lens | Time for Inverse Ray Tracing | |
| Sequential | Parallel |
| 801x801 (lens scale 0.005) | 1 | 19.597 | 0.075 |
| 2 | 23.899 | 0.076 |
| 3 | 28.496 | 0.074 |
| 100 | 614.725 | 0.075 |
| 1601x1601 (lens scale 0.0025) | 1 | 68.086 | 0.088 |
| 2 | 94.852 | 0.095 |
| 3 | 112.846 | 0.089 |
| 100 | 2512.41 | 0.095 |

Table 1: Time for modeling different resolution lens image

The test is a well demonstration for both **Amdahl’s and** Gustafson’s Law. As the more fraction of the serial is parallelized, there will be a big performance speed up. On the hand, a thread is dedicated to calculate a single pixel which means that more computing power is employed. The time for computing the lens image of different resolution is almost stable.

During the test, I find that he time for copying data to and from GPU space is fluctuant in the test. Table 2 shows testing result with long arrays. Several tests proves that the first time always takes a long time.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Array size | 250 | 250 | 1458 | 98 | 5120 | 3072 | 5120 | 10 | 4096 |
| Time (ms) | 36.402 | 0.144 | 0.155 | 0.138 | 0.579 | 0.181 | 10.193 | 0.137 | 0.186 |

Table 2 time for allocating memory and copying data to GPU space

**Appendix**

All the pictures below are 1601x1601 pixels

