ELEKTROTEHNIŠKI VESTNIK 79(1-2): 19–24, 2012
ENGLISH EDITION

Implementation of the r.cuda.los module in the open source GRASS GIS by using parallel computation on the NVIDIA CUDA graphic cards

Andrej Osterman

Telekom Slovenije d.d., Cigaletova 15, 1000 Ljubljana, Slovenija
E-mail: andrej.osterman@telekom.si

Abstract. Parallel computing is in expanding phase in GIS applications. A very attractive solution for parallel computing are the NVIDIA graphic cards, with a parallel computing platform and the CUDA (*Compute Unified Device Architecture*) programming model. The basis for this paper is the $r.los$ module used to calculate optical visibility (*LOS - Line of Sight*), which is already implemented in the GRASS GIS environment. A completely new $r.cuda.los$ module with the same functionality as the $r.los$ module is presented. By using the $r.cuda.los$ module for radio planning purposes of limiting the computation along the vertical and horizontal angle is also make possible. Visibility is calculated for each slice. The responsibility for the calculation of each slice is with its own thread from the parallel processor. At the size of the map of $28161\,x\,17921$ points with the resolution $12,5m\,x\,12,5m$, the computation time is 18 s. In parallel computing the GIS data, the performance can be one, two or even three size classes faster than in the sequential computing.

Keywords: CUDA, parallel programming, GRASS GIS, GPU, Line of Sight, LOS, cartography

1 INTRODUCTION

The use of geographical information systems (GIS) is in ascent. As in other areas, here, too, are gaining importance of open-source packages. The paper is based on the GRASS GIS (Geographic Resources Analysis Support System) open-source package [1]. GRASS GIS is an open-source environment providing uniform way of editing and analyzing the geographical data. It also contains tools producing the desired services. Poor characteristic of individual tools in GRASS is their relative calculation slowness, because of the usually used relatively large maps on which is preformed sequential computing. The time needed for the calculation is in the sequential method proportional to the product of the calculation time of the function itself and the number of points (size of the map).

A better option is parallel calculation. In parallel computing, the computation time can be pretty much reduced by converting parts of the program from the sequential to the parallel way.

In parallel computing we need to solve some other problems which are not occur at sequential computing, such as synchronization of parallel tasks. One of the most currently popular solutions to represent the parallel programming are graphical units from CUDA (*Compute Unified Device Architecture*) family. The new architecture enables parallel computing (also non-graphical mathematical problems) on the graphic card itself ([2]).

In this paper we focus on the $r.los$ module to calculate optical visibility (*LOS - Line of Sight*), which has already been implemented in the open-source GRASS GIS environment. The $r.los$ sequential module is used to calculate visibility from the observer. The main input is a digital map of the terrain height (*DEM - digital elevation model*). The module must also be provided with the point of interest POI (the coordinates x, y), height above the ground of observation and the maximum distance of observation (*max_dist*). The $r.los$ sequential module is relatively slow for computing. Because of the long computation time it is practically unusable over the maximum distance of observation of more than 40 km in the digital map with the resolution of 100 x 100 m, since the computation time is more than 30 s. In this paper we present a new module for computing visibility (*LOS - Line of Sight*). The new module gets insertion $cuda$ and is called $r.cuda.los$. In the main computation, parallel processing CUDA on NVIDIA graphics card is used. It turns out that the $r.cuda.los$ parallel module calculates the visibility by a factor of 10 to 1000 faster than the $r.los$ sequential module. The main time consuming factor in no longer computation but the read and write digital map from the hard disk into the host memory and then to the GPU global memory, and the way back.

Because of the possible use of the $r.cuda.los$ module for radio planning purposes, parameters are added for the direction, scope azimuth (horizontal angle of the radiation 'antenna'), slope (*tilt*) and the scope of tilt

Received December 12, 2011
Accepted March 12, 2012

(vertical angle of the radiation 'antenna'). The module has been successfully used in the initial planning of the radio sector in mobile technology.

The paper is organized as follows. In section 2, preparation of the digital relief map is described. Section 3 talks about preparation of the global memory. Section 4 describes geometries of individual slices (thread) in the line of sight calculation. The heart of the program, the kernel, is described in section 5, where the truncated source of the kernel is given. In section 6, the results of the calculation and a comparison between the $r.los$ sequential module and $r.cuda.los$ module are given. In section 7, the pros and cons of the new implementation are discussed. The possible improvements of the new module are also proposed.

2 Preparation of the digital maps

The digital elevation map (*DEM*) is stored on the hard disk in the raster shape. The digital map is read from the hard disk into the host memory and then copyed to the global memory on the GPU on which parallel computation over data is performed. After the calculation is completed, the required result is transfered from the GPU device to the host memory and then written to the hard disk. Reading and writing to the hard drive is done with conventional C functions `fread()` and `fwrite()` . Data transfer from the host to the GPU device and back is made by the `cudaMemcpyHostToDevice()` and `cudaMemcpyDeviceToHost()` functions.

It turns out that the most of computer time (more than 90%) is consumed by the above-mentioned functions. These sequential functions transfer data from one medium to another. Parallel computation on the data is performed on the GPU device.

The memory is allocated as a global memory on the GPU. This means that any thread can access this memory from any block equally. This is why it doesn't matter how the threads are organized into blocks.

3 Preparation of the memory

The dimension of the input digital map is $R * C$, where R is the number of rows and C the number of columns. Each point of the map contains an integer value (two or four bytes), which represents the altitude. The input memory is shown in Fig. 1.

If the size of the digital map is greater than approximately half of the available memory on the graphics card, the input map must be divided into several bands. Computation is made by the graphics card first to the band where the point of interest is located, from which we wish to calculate visibility. The computation order is: *input memory 2; input memory 1; input memory 0; input memory 3*.

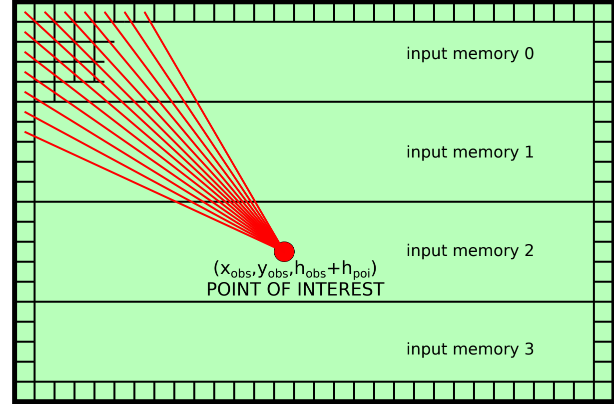

Figure 1. Input memory

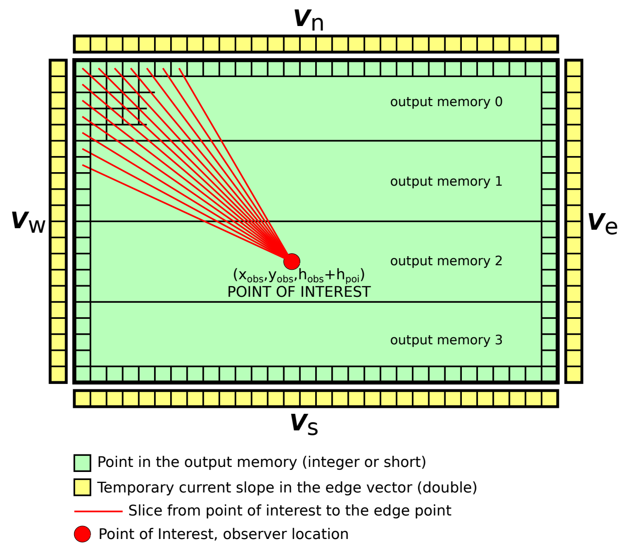

Figure 2. Output memory and tilt vectors

For the output digital map (where the result is stored) the same size of $R * C$ should be provided. First , the values of *null* must be filled in. This can be done quickly by parallel computing. Also, reservations must be made for four *double* vectors $\{V_n, V_s, V_w, V_e\}$. V_w, V_e have R elements and V_n, V_s have C elements. In these four vectors there are temporary vertical angles stored. The units are in radians. Prior to computing, each element of all the four vectors are filled with the value $-\pi/2$. This value represents the view directly to the ground by the observer. Looking straight toward the horizon has the value 0, looking straight up has the value of $\pi/2$.

4 VISIBILITY CALCULATION FOR INDIVIDUAL SLICES

Visibility is calculated for each slice. It is always started at the point of interest. According to the pre-calculated step, we move to the edge point of the map (see Fig. 2, the red line from POI to the edge of the map). Their own thread from parallel processor is responsible for the computation of each slice. For each point of each slice, distance d is computed from POI to the current point (equation (1)). (x_{obs}, y_{obs}) is the coordinate of the observer (POI). (x_d, y_d) is the coordinate of the current point. d is the distance between these two points.

$$d = \sqrt{(x_{obs} - x_d)^2 + (y_{obs} - y_d)^2} \quad (1)$$

For each point in the slice, vertical angle α is computed (see Eq. (2) and Fig. 3).

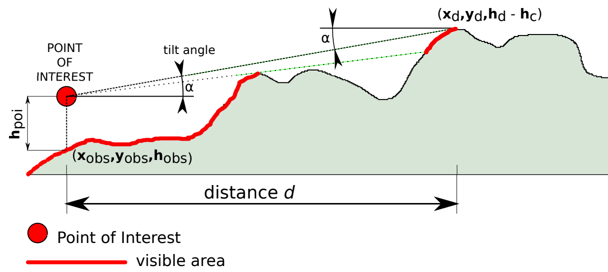

Figure 3. Slice

α is the angle (*tilt*) under which the observer sees point (x_d, y_d). h_d is the above sea level of point (x_d, y_d). The altitude at which the observer is located is marked by h_{obs}. The height above the ground observer is h_{poi}.

$$\alpha = \arctan\left(\frac{(h_d - h_c) - (h_{obs} + h_{poi})}{d}\right) \quad (2)$$

In equation (2) the height correction factor is h_c. It occurs due to the curvature of the Earth (Fig. 4) calculated by using equation (3) in which the average radius of the Earth is $6370.997\,km$. The observer altitude is not considered because of the resulting error just on the fourth decimal place when the average distance from the center of the Earth for a particular area is entered.

$$(h_c + r_{earth})^2 = d^2 + r_{earth}^2$$
$$h_c = \sqrt{d^2 + r_{earth}^2} - r_{earth} \quad (3)$$

New value α is written to the element of vector $V[tid]$ when new computed value α is greater than the value in the element of vector $V[tid]$. At the same time, the current point is characterized as visible. This means α is written to the output map.

However, if value α is smaller than the value in the element of vector $V[tid]$, nothing has been done (in the output map there is already *null* written).

In this way, the visible points are written with the α values at the output map. At invisible points there is *null* written. If the output folder is written in the integer format, $\pi/2$ must be added to the result and multiplied by 10^5 before it is written to the output map.

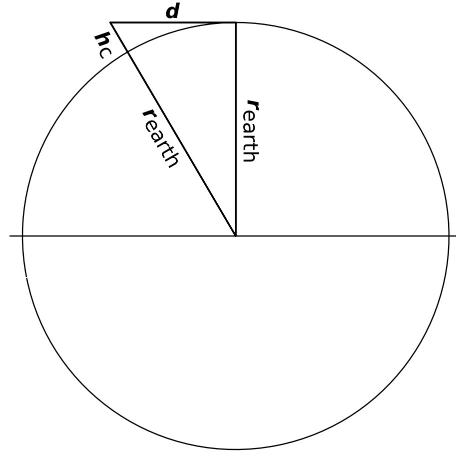

Figure 4. Correction of the height due to the Earth curvature

5 REALIZATION OF THE PROGRAM, KERNEL

A simplified source code of the kernel (a piece of the code performed by the graphics card with CUDA architecture [2]) for computing the visibility is listed below.

Organization of threads by blocks is in this case not important as they do not cooperate with each other. Global memory is used for storing the digital map to which all the threads have an equally shared access. The weakness of the global memory access is relatively slow latency of a few of the 10 clock periods [2]. However, there is no other option because of the huge size of the digital maps.

For any input data which during the kernel operation does not change (maximum distance, azimuth limits, restrictions on the slope ...), the so-called constant memory (`__constant__`) is used.

At the beginning, each thread gets its own index (*tid*). Index *threadIdx.y* represents the so-called four subthreads. It is designed to set and enforce computation lobes to the north, south, west and east side of the digital map. At the end, the step displacement (x_{step} and y_{step}) is calculated leading the computation from the POI to the map boundary point. In the listing there is no sign of calculation step given. However this depends on the direction of the sky the computation is moving to (i.e. the sign depends directly on *threadIdx.y*).

The main loop is actually running variables x and y from the POI to the edge point. Inside the main loop distance d, angle α and height adjustment h_c are computed for each point in the slice. In the original code, the azimuth (angle from the north) is calculated. Depending on azimuth and tilt, computation can be limited (if so set in the input parameters).

Angle α is compared with the value in the element of vector $V[tid]$. If angle α is greater than or equal to the element of vector $V[tid]$, the current computation point is visible and angle α is written to the output map.

The number of threads in the kernel depends on the size of the map. The number of threads is $N = 2 * C + 2 * R$, where C is the number of columns and R is the number of rows. The way the threads are organized into the blocks in this particular case is not important because they all use the global memory of the GPU.

```
tid=threadIdx.x + blockIdx.x * blockDim.x;
x_step=1.0;
y_step=1.0;

switch( threadIdx.y )
{
case 0:
  if(tid>cols) return;
  x_end=tid;
  y_end=0;
  V=V_n;
  x_step=fabs((x_end - x_obs)/(y_end - y_obs));
  break;
case 1:
  if(tid>cols) return;
  x_end=tid;
  y_end=rows;
  V=V_s;
  x_step=fabs((x_end - x_obs)/(y_end - y_obs));
  break;
case 2:
  if(tid>rows) return;
  x_end=0;
  y_end=tid;
  V=V_w;
  y_step=fabs((y_end - y_obs)/(x_end - x_obs));
  break;
case 3:
  if(tid>rows) return;
  x_end=cols;
  y_end=tid;
  V=V_e;
  y_step=fabs((y_end - y_obs)/(x_end - x_obs));
  break;
}

tilt=-PI/2.0;

// loop from POI to edge point
for(x=x_obs,y=y_obs; (x<=cols)and(y<=rows);
  x+=x_step,y+=y_step)
{
  d=sqrt((x-x_obs)*(x-x_obs)
        +(y-y_obs)*(y-y_obs));
  h_c=sqrt(d*resol*d*resol+6370997.0*6370997.0)
      -6370997.0;
  h=read_input_map(x,y);
  h=h-h_c;
  alpha=atan((h-h_obs)/d);
  if(alpha > (V[tid]-0.00005) )
  {
    write_output_map(x,y,alpha);
    if( alpha > V[tid] ) V[tid]=alpha;
  }
  // POI red color
  if(d<2.5)
  {
    write_output_map(x,y,PI/2);
  }
}
__syncthreads();
```

6 Comparison of r.los and r.cuda.los

The $r.cuda.los$ module is run from the terminal command prompt as $r.los$ module. Both modules are tested in a PC computer running Linux. The graphics card is NVIDIA GeForce GTX 560Ti, CPU is Intel Core(TM) Duo CPU E8200 @ 2.66GHZ and memory size is 2.0 GiB.

NVIDIA GeForce GTX 560Ti Graphics Card has the following important properties:

- global memory: 1024 MB
- CUDA cores: 336
- maximum threads per block: 1024
- version: 2.1

More detailed information about the above graphic card are available on the website [8].

Fig. 5 shows an example of computing the $r.los$ module on a map with the resolution of $25m\,x\,25m$. The maximum distance of computing is 10 km, and the height above the ground of the observer is 20 m. The same computation is made with the new $r.cuda.los$ module (Fig. 6). There is no difference in the pictures. They are intentionally shown in different colors, which is made with a $r.color$ module.

The warm colors (yellow, orange, red) in Fig. 6 represent the observer view above the virtual horizon line (negative tilt), while the cold ones (light blue, dark blue) represent the observer view below the virtual horizon line (positive tilt).

Table 1 shows a comparison between the computation times for the two modules. A combination of three different input maps with three different resolutions is shown ($100m\,x\,100m$, $25m\,x\,25m$ in $12,5m\,x\,12,5m$). For each resolution, the combinations of four distances were 5 km, 10 km, 20 km and 50 km. The computation time for each combination is given in seconds.

When calculating with many points (or longer distances with a dense resolution map), the $r.los$ module is no longer appropriate, since computation time is practically useless (on the table it is marked with a dash).

For the map size of $28161\,x\,17921$ and resolution $12,5m\,x\,12,5m$ (the entire territory of Slovenia), LOS can be computed with the $r.cuda.los$ module, the computation time in this case is 18 s.

Table 1. Comparison of the computation times for the r.los module and the r.cuda.los module

Map [m x m]	max_dist	r.los	r.cuda.los
100m x 100m	5 km	0.1 s	0.05 s
	10 km	0.2 s	0.06 s
	20 km	2.2 s	0.09 s
	50 km	44 s	0.15 s
25m x 25m	5 km	2.4 s	0.3 s
	10 km	30 s	0.3 s
	20 km	511 s	0.6 s
	50 km	-	1.3 s
12.5m x 12.5m	5 km	32 s	0.7 s
	10 km	516 s	1.2 s
	20 km	-	3.1 s
	50 km	-	6.8 s

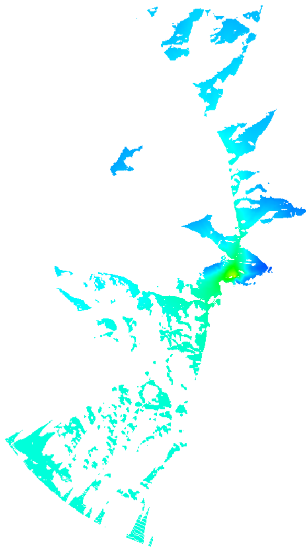

Figure 5. LOS computation with the r.los module

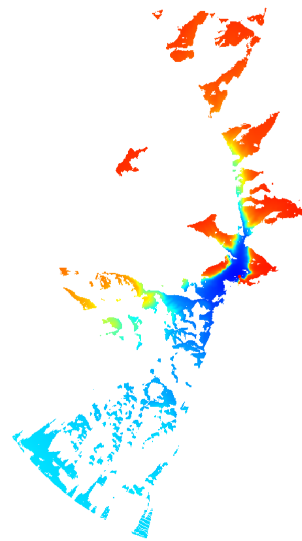

Figure 6. LOS computation with the r.cuda.los module

7 Discussion and Conclusions

Anyone dealing with the GIS environment knows that producing geographic data is an accurate, diligent and, above all, time-consuming work. Deliberate preparation of the GIS data contributes to faster and easier processing. However computing the data is also very important. A key role is an appropriate hardware supported by an optimal software.

Processing geographical data just cries out for parallel computation. If we compare the sequential and parallel computation of the GIS data, in parallel computation the execution can be for one, two or even three size classes faster!

Paper [2] writes in general about the CUDA architecture. Besides the technical value of this paper, its importance for the Slovenian terminology in this new technical branch is considerable, too.

In the GRASS-GIS environment there have been several researches on parallel computing made. One of them is [3] in which parallel computing deals with clusters of servers with each server doing a task.

In paper [4] a general procedure of programming GIS in CUDA is described and an example is given on an averaging filter ($meanfilter$).

The paper [5] describes implementation of digital map projection transformations they using CUDA.

Paper [6] does not describe parallel computing but the group already working on implementation of modules in a parallel computing mode.

Our paper does not address the general problem of migration from the sequential to parallel computing mode. Its focus is on improving the existing $r.los$ module.

A general shift to using the parallel computing method will probably happen after the number of specific modules in the parallel computing mode has reached the critical point.

For the $r.los$ module, the on-line source code is freely available. Nevertheless, the $r.cuda.los$ module is completely new.

Both modules produce the same result and use the GRASS data file recording method. Running the program for both modules are quite similar. Most of the arguments are same. For additional functionality of $r.cuda.los$ module some further arguments are required.

The new $r.cuda.los$ module is written for the CPU and GPU. The compiler is $nvcc$ (instead of gcc for GRASS).

GRASS has made an extremely effective framework for building modules. This framework also includes a GUI to run the module itself and an attached frame to support the application module. As the new $r.cuda.los$ module has not yet joined the GUI frame, it can only

run through the command line.

The new module is limited to two modes of the record input and output map. Namely, the $r.cuda.los$ module can read the uncompressed mode of the record in the int or $short$ type. The results (the output map) are obtained in the same format as the input data (the input map). Disadvantages of the current implementation are shown here, because we are writing down only the integer values to the output file. If are small values written (as in our case where the written angles α are between the values of $-\pi/2$ and $\pi/2$), multiplied values must be written down, e.g.multiplied by 10^5. Record in the form of $double$ is waiting for implementation.

If the input map is recorded in compressed mode, it must be expanded with a $r.compress$ module.

Table 1 shows advantages of parallel computing. It shows the execution time of the modules $r.los$ and $r.cuda.los$ according to the input data. The $r.los$ module becomes very slow for large values of max_dist. We can see that computation of $r.los$ becomes very slow for values of max_dist greater than 50 km for digital maps with the resolution of $100m\,x\,100m$, and for the values of max_dist greater than 5 km for digital maps with the resolution of $12.5m\,x\,12.5m$.

The $r.cuda.los$ module still has some room for improvement. It is important to improve the time-consuming reading of the digital file from the hard disk and writing back to it. One of the options is to record the digital map in a compressed format. For the GPU unit a kernel should be written for fast finding the values of the points (quick indexing) in a compressed format.

Certainly, parallel computing is very promising for GIS systems, for its speeding up the execution times for some size classes.

REFERENCES

[1] *GRASS GIS,* http://grass.fbk.eu

[2] Tomaž Dobravec, Patricio Bulić, *Strojni in programski vidiki arhitekture CUDA*, Elektrotehniški vestnik 77(5): 267−272, 2010

[3] Fang Huang, Dingsheng Liu, Peng Liu, Shaogang Wang, Yi Zeng, Guoqing Li, Wenyang Yu, Jian Wang, Lingjun Zhao, and Lv Pang, Chinese Academy of Sciences, Beijing, China, Northeastern University, Shenyang, China *Research On Cluster-Based Parallel GIS with the Example of Parallelization on GRASS GIS* , Grid and Cooperative Computing, 2007. GCC 2007. Sixth International Conference on

[4] Yong Zhao, Zhou Huang, Bin Chen, Yu Fang, Menglong Yan, Zhenzhen Yang, Institute of Remote Sensing and Geographic Information System, Peking University *Local Acceleration in Distributed Geographic Information Processing with CUDA*, Geoinformatics, 2010 18th International Conference on.

[5] Yanwei Zhao, Zhenlin Cheng, Hui Dong, Jinyun Fang, Liang Li, Institute of Computing Technology, Chinese Academy of Sciences, Graduate University of Chinese Academy of Sciences *FAST MAP PROJECTION ON CUDA*, Geoscience and Remote Sensing Symposium (IGARSS), 2011 IEEE International.

[6] Andrej Hrovat, Igor Ozimek, Andrej Vilhar, Tine Celcer, Iztok Saje, Tomaž Javornik, *An Open-Source Radio Coverage Prediction Tool*, Department of Communication Systems, Jozef Stefan Institute, Mobitel, d.d. ISSN: 1792-4243, ISBN: 978-960-474-200-4

[7] Jasons Sanders, Edward Kandrot, *CUDA by Example*, Addison-Wesley, 2011

[8] *NVIDIA GeForce GTX 560 Ti,* http://uk.geforce.com/hardware/desktop-gpus/geforce-gtx-560ti

Andrej Osterman graduated in 1991 from the Faculty of Electrical Engineering, University of Ljubljana with his thesis entitled Graphical presentation of the antenna direction diagrams in the programming language C++. He works at the Telekom Slovenia, Radio network department. His current work includes mobile network statistics, GIS tools and programming in open-source systems.