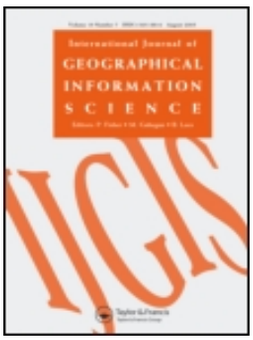


International Journal of Geographical Information Science

ISSN: 1365-8816 (Print) 1362-3087 (Online) Journal homepage: https://www.tandfonline.com/loi/tgis20

An IO-efficient parallel implementation of an R2 viewshed algorithm for large terrain maps on a CUDA GPU

Andrej Osterman, Lucas Benedičič & Patrik Ritoša

To cite this article: Andrej Osterman, Lucas Benedičič & Patrik Ritoša (2014) An IO-efficient parallel implementation of an R2 viewshed algorithm for large terrain maps on a CUDA GPU, International Journal of Geographical Information Science, 28:11, 2304-2327, DOI: 10.1080/13658816.2014.918319

To link to this article: https://doi.org/10.1080/13658816.2014.918319

Published online: 27 May 2014.

International Journal of Geographical Information Science, 2014
Vol. 28, No. 11, 2304–2327, http://dx.doi.org/10.1080/13658816.2014.918319

Taylor & Francis
Taylor & Francis Group

An IO-efficient parallel implementation of an R2 viewshed algorithm for large terrain maps on a CUDA GPU

Andrej Osterman[a]*, Lucas Benedičič[b] and Patrik Ritoša[b]

[a]*Access Networks Department, Telekom Slovenije d.d., Cigaletova 15, SI-1000 Ljubljana, Slovenia;*
[b]*Research and Development Department, Telekom Slovenije d.d., Cigaletova 15, SI-1000 Ljubljana, Slovenia*

(*Received 7 March 2013; final version received 22 April 2014*)

A rapid and flexible parallel approach for viewshed computation on large digital elevation models is presented. Our work is focused on the implementation of a derivate of the R2 viewshed algorithm. Emphasis has been placed on input/output (IO) efficiency that can be achieved by memory segmentation and coalesced memory access. An implementation of the parallel viewshed algorithm on the Compute Unified Device Architecture (CUDA), which exploits the high parallelism of the graphics processing unit, is presented. This version is referred to as r.cuda.visibility. The accuracy of our algorithm is compared to the r.los R3 algorithm (integrated into the open-source Geographic Resources Analysis Support System geographic information system environment) and other IO-efficient algorithms. Our results demonstrate that the proposed implementation of the R2 algorithm is faster and more IO efficient than previously presented IO-efficient algorithms, and that it achieves moderate calculation precision compared to the R3 algorithm. Thus, to the best of our knowledge, the algorithm presented here is the most efficient viewshed approach, in terms of computational speed, for large data sets.

Keywords: viewshed; line of sight; large terrain maps; CUDA; GPU

1. Introduction

Many interesting applications, such as visibility studies in archeology (Lake *et al.* 1998, Lake and Woodman 2003) or siting problems (Franklin and Vogt 2006, Magalhães Salles *et al.* 2011), require a viewshed analysis. For example, landscape architects can use a viewshed analysis to determine the visual impact of new structures (Kvarfordt 2010). During the geographic planning of modern wireless communication networks, a viewshed analysis is frequently used (Dodd 2001). Viewshed analyses range from determining the impact of a new standing structure to calculating signal-propagation conditions. In some cases, limits must be introduced to the vertical and horizontal angle of view to best represent the antenna radiation pattern. To increase the accuracy of such a representation, the effective curvature of the Earth is also considered. In some applications, it is necessary to adjust the effective curvature of the Earth to best represent the radio signal diffraction effect. Compared to optical visibility, radio signals are less susceptible to shadowing by obstacles.

For a more accurate analysis, detailed, high-resolution maps are required. Due to the increased size of such map data sets, viewshed analysis becomes a very time-consuming

*Corresponding author. Email: andrej.osterman@guest.arnes.si

operation. Using well-known algorithms, the computation time of large data sets can amount to several hours (Lake *et al.* 1998).

Our daily work involves the use of geographic information systems (GISs) for radio network planning tools (Hrovat *et al.* 2010). One of the essential components of radio network planning tools is the viewshed module, which, in cooperation with other radio modules, produces the required prediction results.

Several algorithms for viewshed calculations are currently known (Toma 2012). A straightforward approach to determine the visibility from a given point of interest (POI), called the R3 algorithm, requires $O(n^3)$ time for the entire map. The algorithms R2 and XDraw are adaptations of R3. Both run on $O(n^2)$ time at the expense of accuracy. All the three algorithms were described in Franklin and Ray (1994).

A different approach, the sweep algorithm, was described in van Kreveld (1996); this algorithm requires $O(n\sqrt{n})$ time for calculation. Van Kreveld improved the algorithm to require only $O(n \lg n)$ time for viewshed calculation.

To achieve an acceptable user experience in real-time applications, a faster implementation of the viewshed algorithm is needed. A significant increase in speed can be achieved using the Compute Unified Device Architecture (CUDA) to exploit the parallelism of graphics processing units (GPUs). Various effective methods can be applied, as described in NVIDIA Corporation (2013), and different algorithm implementations can be used (Xia *et al.* 2011, Tabik *et al.* 2013, Zhao *et al.* 2013). However, not all viewshed algorithms are equally suitable to be parallelized. We will show that a parallel implementation of the R2 algorithm can be very efficient. The R2 algorithm has been criticized for not being input/output (IO) efficient (Toma 2012). We will demonstrate that our parallel implementation is IO efficient and that it can be used with massive data sets within a very short computation time. This is, to the best of our knowledge, a novel method for viewshed computation.

In this article, a parallel approach for a derivative of the viewshed R2 algorithm on digital elevation model (DEM) maps is presented. The focus of our work is on the performance attributes of the algorithm, increasing the algorithm's flexibility regarding the map size and maintaining the accuracy of the algorithm within practical limits. The proposed implementation of the R2 parallel algorithm, named r.cuda.visibility, is implemented in a CUDA GPU. This algorithm is accurate compared to the sequential R3 algorithm (Shapira 1990) implemented in the Geographic Resources Analysis Support System (GRASS) GIS environment (GRASS GIS 2013). We also implement a derivative of the R2 algorithm in a sequential version, called r.cpu.visibility. An efficiency comparison was made between the r.cuda.visibility, r.cpu.visibility, and TiledVS (Ferreira *et al.* 2012). The source code of our implementation is available at the following website: http://viewshed.s51mo.net.

The remainder of the article is organized as follows. In Section 2, preliminaries and background information are provided. In Section 3, a brief CUDA tutorial is provided. In Section 4, the proposed version of the R2 parallel algorithm is described. The implementation of the parallel algorithm is described in Section 5. In Section 6, procedures for optimizing the implementation of the algorithm in regard to the variable type and occupancy analyses are presented. Memory considerations are described in detail in Sections 7 and 8. The influence of the number of segments is elaborated in Section 9. A comparison of several algorithms is presented in Section 10. Related works are reviewed in Section 11. Conclusions, discussion, and future work are presented in Section 12.

2. Background

A viewshed analysis is generally calculated on DEM data sets, which are stored in data files. In the context of this work, DEMs assembled from regular square grids (RSGs) are used. Each grid cell of the RSG holds a value indicating the terrain elevation above sea level. In our case, these values are stored as four-byte signed integers, expressed in meters.

A POI, or viewpoint, is the viewer's location on the terrain map (Figure 1). The POI has several parameters: the position coordinates (x,y), the DEM value, and the arbitrary height above ground level (explained in Section 4.4).

For the purpose of a specific application (e.g., antenna radiation pattern representation), the POI can include additional parameters: azimuth, elevation, and the horizontal and vertical view angle ranges.

A ray (calculation path) traversing the grid cells is defined from the POI toward every edge grid cell on one of the four DEM borders. Our algorithm always takes rays from the POI to the edge of the map, in contrast to the original R2 algorithm, which takes rays from the POI to the perimeter of the range. We chose this approach primarily because of its simple implementation (there is no search procedure for grid cells on the perimeter range). At the same time, improvements in accuracy were achieved due to the increased density of rays.

The number of rays N used in a viewshed calculation is therefore independent of the range and is described by the equation $N = 2C + 2R–4 \approx 2C + 2R$, where R is the number of rows and C is the number of columns in the DEM.

3. Introduction of CUDA

Because our parallel implementation of the R2 algorithm is implemented in CUDA, we will briefly introduce some terms. CUDA uses different memory types (Figure 2a), which differ according to their speed, reachability, writability, and validity. Registers are the fastest, and they are accessible only inside a thread. Shared memory is slightly slower. Threads inside one block can communicate with each other through shared memory. The number of registers and the amount of shared memory is defined per block. The content of registers and shared memory are valid inside one thread block.

Global memory is the largest and slowest memory type. The content is retained through all kernel launches until the memory is freed. Constant memory is optimized for broadcast, i.e., when the threads in a warp all read the same memory location. Texture memory has implemented additional hardware functions, such as interpolation. Texture memory is optimized for 2D spatial locality. When a read is being broadcast to the threads, constant memory is much faster than texture memory.

A kernel is parallel code executing on different data sets. Threads are organized into blocks and grids (Figure 2b). During the execution, threads can only communicate with each other

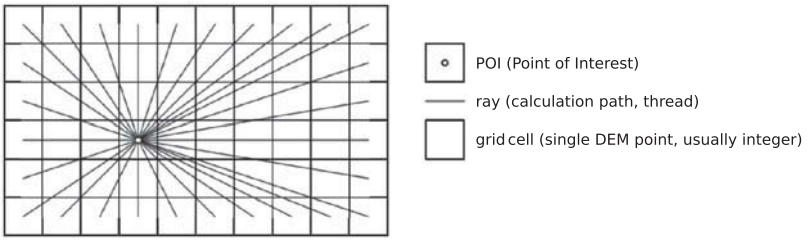


Figure 1. DEM, rays (calculation path), and point of interest.

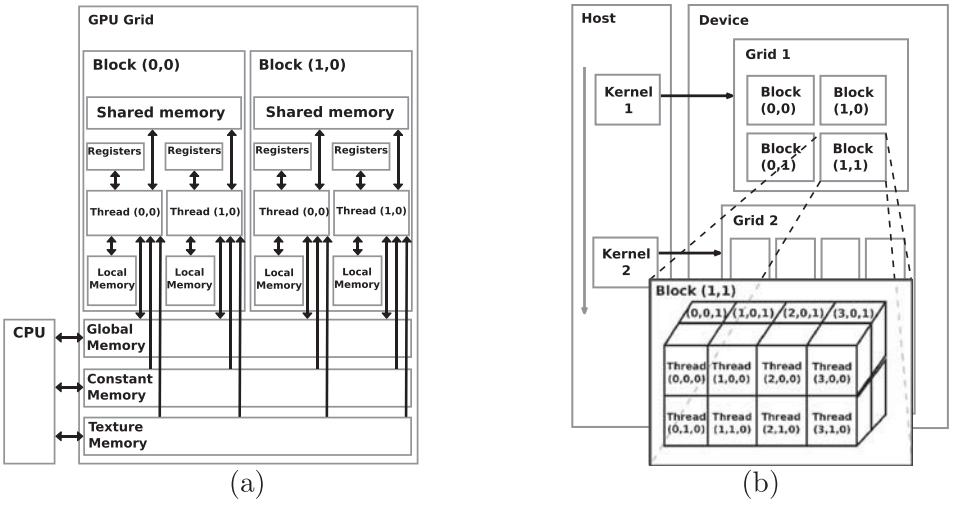


Figure 2. CUDA memory hierarchy (a) and logical representation of the CUDA programming model (b).

inside a single block. Blocks cannot communicate with each other because of the random order of the execution of the blocks inside a grid. If several kernel launches are sequential (which is true in our case), grids can communicate with each other through global memory.

In a host, we use pinned memory for map data storage. Pinned memory is memory guaranteed to be allocated on the main host memory unless there is not enough room. Pinned memory is memory allocated using the cudaMallocHost function, which prevents the memory from being swapped out and provides improved transfer speeds.

4. Algorithm implementation

The main focus of our implementation of the R2 algorithm is to calculate visibility on a huge volume of data that do not fit in the internal memory. At the same time, we want the fastest possible data processing. We completed two main versions of the implementation: sequential and parallel. The key point in the simple and effective implementation of both the parallel and the sequential versions is segmenting the map into bands called segments. In other words, the map is divided into longitudinal segments in the most natural way, which allows for quick read, write, and copy operations using well-known standard C and CUDA functions.

Parallel implementations usually require a different approach than sequential ones. The current implementation of the parallel R2 viewshed algorithm is realized with a combination of sequential and parallel (CPU and GPU) methods. A sequential method is used for copying data sets to and from the GPU, whereas a parallel method is implemented in the kernel to accomplish the viewshed calculation. The main advantage of the parallel version of our implementation is that reading global data inside the kernel is coalesced to a certain degree (Section 7).

In our algorithm, we use interpolation within the kernel. By using texture memory for interpolation purposes, the benefit is an approximately 10% faster computation time of the kernel. The drawbacks are inflexible memory manipulation (segment size is not flexible) and a limited set of interpolation methods. Therefore, we avoid the use of texture memory. The interpolation is described in Section 4.5.

In the following sections, we refer to the desktop computer as the host and to the GPU as the device, i.e., host memory means the RAM memory on the desktop computer, whereas device memory means the global memory on the GPU.

4.1. Terrain segmentation

The input data set used in the viewshed analysis is the DEM stored in a file on the host data storage and is referred to as the input map. The size of the entire input map is $R \cdot C$, where R is the number of rows and C is the number of columns. Each grid cell of the input map contains a value that represents the elevation above sea level. The input map is schematically shown in Figure 3a.

The output data set is the same size as the input data set and is referred to as the output map. The output map is schematically shown in Figure 3b. Each grid cell of the output map contains a value, with information on whether a particular grid cell is visible.

If a large DEM map is used to calculate the viewshed analysis, it cannot be stored on the host or the device in one part. Our goal is to be able to process much larger maps than the memory capacity of a host or device. To achieve this goal, larger input maps and output maps are divided into equally sized strips (segments).

The number of segments (Seg) corresponding to the terrain segmentation can be calculated using Equation (1), where M_m is the memory size of the input map ($M_m = R \cdot C \cdot S_{dtype}$, where S_{dtype} is a variable size in bytes, representing the grid-cell data type; e.g., for the integer type, S_{dtype} is 4) and M_g is the smaller value of the amount of global memory available on the device and the free host memory. Other data must also be stored in global memory (M_a), i.e., the horizon vector (explained in Section 4.3), and the parameter data for cooperation between blocks of kernel code during ray coordination and synchronization. The constant 4 in Equation (1) represents the double buffer used for the input and output maps.

The memory size needed for a segment (M_s) can be calculated using Equation (2).

$$Seg \geq \left\lceil \frac{4 \cdot M_m}{M_g - M_a} \right\rceil \tag{1}$$

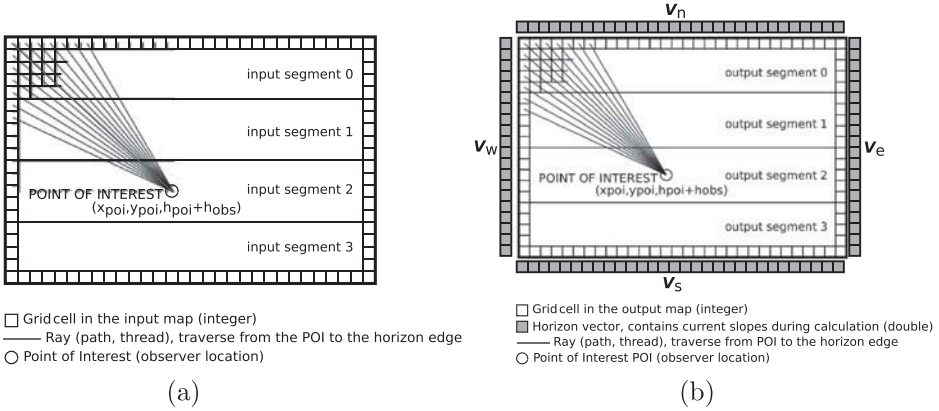


Figure 3. Input DEM map (a) and output viewshed map (b).

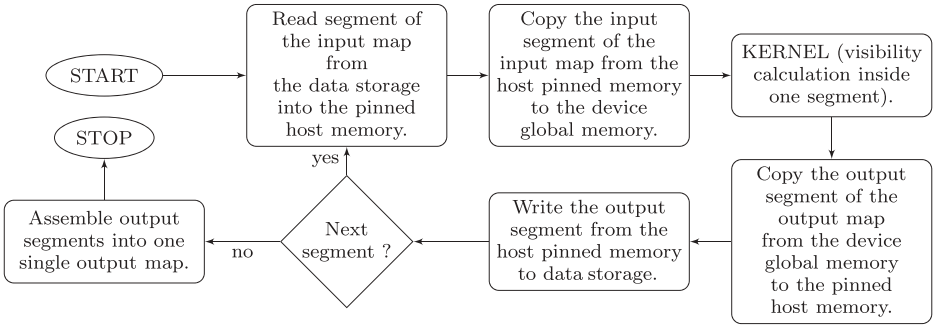


Figure 4. Framework flowchart of the parallel implementation of the R2 algorithm.

$$M_s = \frac{M_m}{Seg} + 2 \cdot C \cdot S_{dtype} \tag{2}$$

A framework of the flowchart of the proposed implementation of the R2 parallel algorithm is depicted in Figure 4. The parallel part of the algorithm is carried out in the element KERNEL. Note that different kernels are executed serially and must not overlap. Each execution of the kernel calculates data in one segment. All other elements on the flowchart are based on sequential methods.

The input segment is read into the host pinned memory (Pinned Memory 2013) and then copied to the device global memory. Once the input segment is in the device global memory, the kernel for the viewshed computation can be started. The kernel executes the viewshed analysis only on the loaded input segment. For proper algorithm operation over all the segments, the intermediate segment results (angular elevation) are stored in the horizon vector. These intermediate results are used as working parameters during the calculation of the subsequent segment.

The segment results of the viewshed kernel (visibility) are stored in the output segment in the device global memory, which is the same size as the input segment. The output segment is then copied from the device global memory to the host pinned memory and then written to the data-storage temporary segment map.

The described steps are repeated as many times as there are segments. The order of processing segments is very important, i.e., starting with the segment where the POI resides, followed by adjacent segments toward the north and continuing until the northern edge of the map is reached. The same procedure is followed toward the southern edge of the map. For example, in Figure 3, segment 2 is processed first. Then, this sequence is followed: segment 1, segment 0, and segment 3.

At the end of the algorithm, all the segments, which are stored in the temporary segment maps, have to be merged into the output map in the proper order.

4.2. Ray path and ray step

As noted in Section 2, rays always start at a POI and travel toward the horizon edge, thus traversing several grid cells along their path. A ray path is a straight line traversing from the POI to the horizon edge. The calculation points are the exact points on the ray where

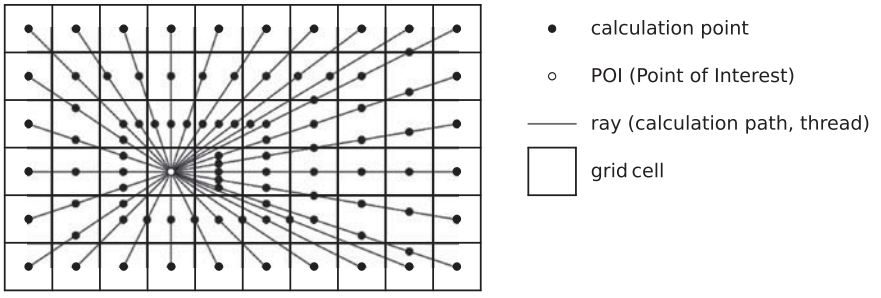


Figure 5. Ray paths and calculation points.

the calculation of the viewshed is performed (Figure 5). The distance between two adjacent calculation points on each ray is determined by the ray step. The ray path and ray step are unique for each ray and are defined before the beginning of the visibility calculation.

The pair (x_{poi}, y_{poi}) represents the POI coordinates, and the pair ($x_{horizon}$, $y_{horizon}$) represents the coordinates of the edge point on the horizon for each ray; thus, each ray path is a line between (x_{poi},y_{poi}) and ($x_{horizon}$, $y_{horizon}$).

For each ray, two unique ray step parameters must be calculated: x_{step} and y_{step}. There is a similar approach for calculating the ray steps toward the north/south (Equation (3)) and east/west (Equation (4)), where *sgn*() is the sign function.

$$x_{step} = \frac{x_{horizon} - x_{poi}}{|y_{horizon} - y_{poi}|}, y_{step} = sgn\left(y_{horizon} - y_{poi}\right) \tag{3}$$

$$x_{step} = sgn\left(x_{horizon} - x_{poi}\right), y_{step} = \frac{y_{horizon} - y_{poi}}{|x_{horizon} - x_{poi}|}, \tag{4}$$

4.3. *Horizon vectors*

The horizon vectors are stored as four double-precision vectors in the device global memory. Altogether, the horizon vectors have N elements. Each vector represents one border edge of the output map, namely, north (V_n), south (V_s), east (V_e) and west (V_w), as shown in Figure 3b. Vectors V_w and V_e consist of R elements and V_n and V_s of C elements. At the beginning of the calculation, each element of all four vectors is set to $-\pi/2$ (minimum value). This value represents the angle of view directly toward the ground. The value 0 represents the view straight toward the horizon and the value $\pi/2$ represents the view straight up toward the sky.

At any given moment during the viewshed calculation, the horizon vector holds the temporary maximum angular elevation (α) for each ray (Section 4.4). These values are based on the maximum obstacle height on the corresponding ray.

4.4. *Visibility calculation*

The visibility property is calculated for each ray. Each thread on the device is responsible for the computation of one ray. At every calculation point on the ray, the distance d from

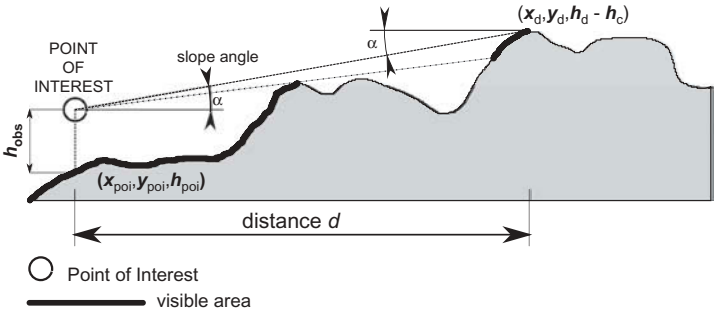


Figure 6. Cross-section of terrain (ray) showing the visible area from the POI and slope angle α.

the POI is calculated using Equation (5), where (x_d, y_d) is the coordinate of the current calculation point.

For every calculation point on the ray, the vertical angle (angular elevation) α is computed (Equation (6) and Figure 6), where h_d is the DEM value at point (x_d, y_d), h_{poi} is the DEM value at point (x_{poi}, y_{poi}), h_{obs} is the additional height above ground level, and h_c is the Earth-curvature height correction factor (described in Section 4.7).

$$d = \sqrt{\left(x_{poi} - x_d\right)^2 + \left(y_{poi} - y_d\right)^2} \tag{5}$$

$$\alpha = \arctan\left(\frac{(h_d - h_c) - \left(h_{obs} + h_{poi}\right)}{d}\right) \tag{6}$$

The value α at a calculation point is compared to the element of the horizon vector V (the horizon vector V could be V_w, V_e, V_n, or V_s, depending on the direction the ray is pointing). When a new value α is greater than the existing one in the element of vector V, the greater value replaces the old value, and the current grid cell in the output segment is marked as visible if additional conditions are fulfilled (the nearest-ray approach, explained below in Section 4.5). If, in contrast, a new value α is smaller than the existing value in the element of vector V, nothing is modified and the visibility status of the current grid cell in the output segment remains the same.

4.5. *Linear interpolation and nearest-ray approach*

The elevation of each grid cell is defined at its center. In viewshed analysis, the calculation points frequently lie outside the grid-cell center, which leads to errors due to the granularity of the elevation map. The map granularity can be reduced using interpolation between neighboring grid cells. For proper interpolation, at each calculation point, two calculation parameters must be determined. The first parameter is the elevation (height) level of calculation point h_{cp}; the second parameter is the distance between the calculation point and the center of the nearest grid cell d_n.

An interpolation example is shown in Figure 7a. For the linear interpolation, the interpolated elevation values at calculation point h_{cp} can be calculated using Equation (7),

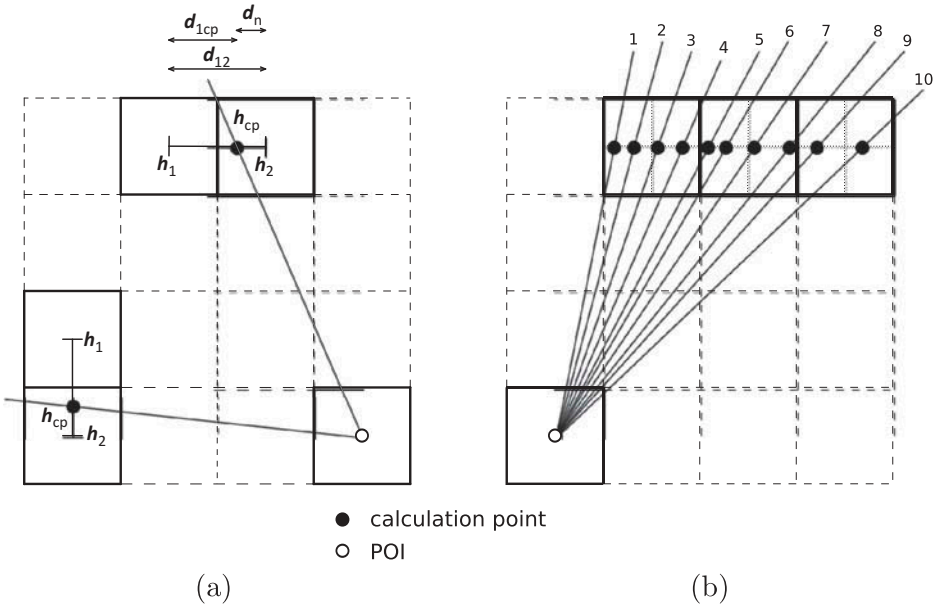


Figure 7. Linear interpolation (a) and nearest-ray approach (b).

where h_1 and h_2 are elevation values of neighbor grid cells, d_{12} is the distance between the centers of adjacent grid cells, and d_{1cp} is the distance between the calculation point and the cell center of the first grid cell.

$$h_{cp} = h_1 + \frac{d_{1cp}}{d_{12}}(h_2 - h_1) \tag{7}$$

In the north/south direction, two horizontally adjacent grid cells are selected for interpolation. In the east/west direction, two vertically adjacent grid cells are selected for interpolation (Figure 7a).

The nearest-ray preference approach contributes to a more realistic computational result of the viewshed algorithm. Many rays can traverse a particular grid cell, especially if the grid cell is near the POI. The nearest-ray strategy determines the visibility criteria of the resulting grid. Logical 'OR' operations, used to combine adjacent rays, lead to an optimistic result. If, however, logical 'AND' operations are used, a pessimistic result is achieved.

The nearest-ray approach, using only the ray that traverses nearest to the cell center, is adopted in the implemented algorithm. The value d_n is used for the proximity criterion. If the ray traverses closer to a grid cell with elevation h_2, then $d_n = d_{12}–d_{1cp}$ (Figure 7a). If the ray traverses closer to a grid cell with height h_1, then d_n becomes equal to d_{12}.

For example, in Figure 7b, rays 3, 7, and 10 are used for the grid-cell visibility decision. Although the majority of rays do not have any effect on the visibility result, they are considered for horizon vector update. The presented approach contributes to a more realistic viewshed analysis (presented in Section 10.2).

4.6. Resemblance of viewshed rays and CUDA threads

Each viewshed ray should be calculated by its own CUDA thread. The basic feature of the R2 algorithm requires that the ray needs information if it is the nearest ray traversing a particular grid-cell center. When more than one ray traverses the particular grid cell, only the nearest ray to the grid-cell center determines whether this particular cell is visible. Clearly, there must be some mechanism to determine which ray is closer to the center of the grid cell. This mechanism works in the following way: for each ray, the distance is calculated from the current traversed grid-cell center. Then, one must determine whether the ray is closer to the grid-cell center than any neighboring ray(s). To achieve this, one must know the distance of the neighboring ray(s) from the center of the grid cell. If the distance(s) of the neighboring ray(s) is greater than the distance of the ray of interest, then the ray of interest is the nearest ray. Only the nearest ray resolves the current cell grid visibility (Figure 7b).

Different rays (threads) can exchange data with each other only through shared memory. In CUDA, there is a limitation that threads can use the same shared memory only within a block, whereas blocks inside one kernel are independent and cannot exchange data among themselves. The typical maximum number of threads per block for current CUDA units is 1024. There are also limitations on the size of the shared memory per block, and for our CUDA device, this size is 49,152 bytes.

In each calculation grid cell, inside a kernel's *for* loop, the current offset data (distance from grid-cell center) are written to shared memory. After that, synchronization among all threads should be performed with *_syncthreads()*. As a consequence, all threads are synchronized, and their offset data are written to the shared memory at this point. The data from the neighboring threads can now be read from shared memory.

Because blocks cannot exchange data among themselves inside the kernel, we take an intertwined thread approach (Figure 8). The last two threads from a particular block and the first two threads from a consecutive block are used for the calculation of the same two rays. Using this technique, all blocks are intertwined by two threads. The first and last thread from each block are not directly used to calculate visibility, but only to calculate the nearest ray. All other threads directly participate in the viewshed analysis.

With this approach, more threads are used than the number of rays. However, no additional technique for data exchange among the blocks is needed, and all blocks can run from one kernel. The number of additional threads is twice the number of blocks used. If we express the additional threads ad_{th} as a percentage, this number is $ad_{\mathrm{th}}[\%] = 100 \cdot 2/th_{\mathrm{bl}}$, where th_{bl} is number of threads in one block (usually 1024 or 512).

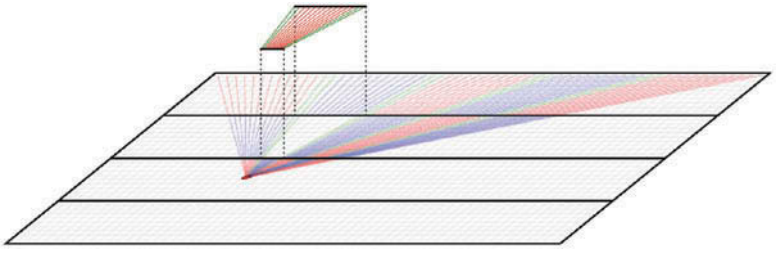


Figure 8. Representation of one CUDA block in a viewshed analysis. The red and blue grouped lines represent unique threads in a block; the green lines represent intertwined threads, which are calculated in two neighboring blocks. The lifted red and green lines represent one block calculation in one segment.

```
__global__ void visibilityKernel()
  __shared__ double offset[]          //initialization part
  calculate x_step and y_step
  calculate initial conditions
  ... //calculation part
  for each grid cell on ray from poi to edge of the map, inside current segment do  //ray generation
    calculate distance between POI and current grid-cell
    calculate interpolated height value
    calculate slope
    calculate ray offset
    store ray offset to shared memory (to offset[])
    __syncthreads()  //all threads in block for all current calculating rays are synchronized at this point
    if slope > V[]then
      V[] = slope     //slope is stored to edge memory
      //neighbor's offsets are read from shared memory (offset[])
      if offset < neighbor offset and thread is not first or last in block then
        current grid-cell is visible
```

Figure 9. Viewshed kernel.

The core of the presented viewshed program is the GPU kernel. An abbreviated listing for clarity purposes is shown in Figure 9. The kernel consists of an initialization and a computational part. The objective of the initialization part is to prepare data variables for later calculation. The objective of the computational part is ray generation (Figure 9, *for* loop in line 6), where we read data from the global device memory, calculate the visibility, and store the results back in the global device memory. Each thread of the kernel is used to calculate only the rays inside the current segment (Figure 8). On each calculation point along a ray path, the following values are calculated: distance, interpolation height, slope, and offset from the current grid-cell center.

4.7. *Earth-curvature consideration*

When working under realistic geographic conditions, the terrain is not a flat plane, but is rather slightly curved due to the shape of the Earth. The Earth is known to be an approximate oblate spheroid. At small distances from the POI, the Earth-curvature effect is negligible. Farther away, this effect has a substantial influence, and a correction of the elevation of distant grid cells must be included in the algorithm for realistic results.

$$(h_c + r_{earth})^2 = d^2 + r_{earth}^2 \tag{8}$$

$$h_c = \sqrt{d^2 + r_{earth}^2} - r_{earth} \tag{9}$$

The effect of the Earth's curvature is shown in Figure 10. Its influence can be estimated using Equations (8) and (9), where r_{earth} is the Earth's radius (average value 6370.997 km). By default, the Earth-curvature correction is included in the proposed algorithm. The exact value r_{earth} depends on the macro-location of the user and can be set arbitrarily.

When dealing with radio propagation profiles, the curved radio rays are replaced with linear rays for the purpose of geometric simplicity (Kukushkin 2004, p. 6). To account for the error introduced if drawing radio rays as straight lines, the Earth's radius has to be increased. The radius of this virtual sphere is known as the effective Earth's radius, and it is approximately equal to four-thirds the true radius of the Earth, i.e., roughly 8500 km.

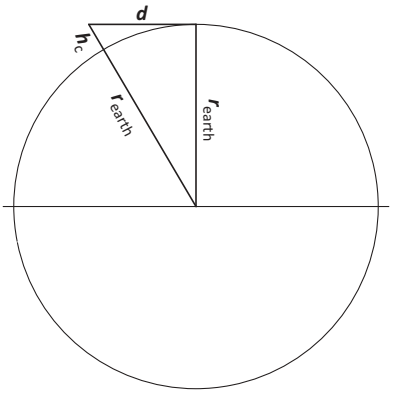


Figure 10. Height correction due to the curvature of the Earth.

5. Implementation of the parallel algorithm on a CUDA GPU

We have implemented two versions of the R2 algorithm. One version is performed by the CPU (r.cpu.visibility) and the other is performed by the CPU and GPU (r.cuda.visibility).

For the parallel version, the NVIDIA CUDA was chosen for the implementation because of the wide availability of hardware (graphical devices) and software (drivers and compilers) for this platform.

The calculations of the algorithm code consist of two main parts: a preparation part and a calculation part. The preparation part contains functions for reading/writing data from/to a hard disk and transferring it to the GPU and calculation calls (basic flowchart from Figure 4). The calculation part implements the viewshed analysis. The CPU code is written in the C++ language, and the GPU code is written using CUDA SDK V5.0. The optimization level for the GPU program is –O3.

Determining the type of CUDA device integrated into the host computer is an important part of the program. This device should support compute capability (CC) 1.3 or higher to provide native support for double-precision floating-point values, i.e., 64-bit values. Another important parameter is the amount of global memory available in the GPU device. This value can range from 1 to 4 GB on currently available GPUs. The amount of free RAM on the host computer should also be determined. Currently, desktop computers have 16 GB or more of RAM. The program should adapt to the smaller free memory value of the two before the number of map segments is determined as in Equation (1).

Memory should be allocated for map segments on both the host and the device. Using page-locked (or pinned) memory, a higher bandwidth is achieved between the host and the device, which is made possible by the function *cudaHostAlloc*. The function *cudaMalloc* is used for memory allocation on the device. The size of the memory allocation is equal to the size of one segment incremented by the size of one row on the map. An additional line is needed for the proper linear interpolation of the last row in each segment (except the southern segment).

The input map is read into host memory in segments. The map file is opened, read, and closed using the well-known functions *fopen*(), *fread*(), and *fclose*(), respectively. To find the starting point of the required segment, the function *fseek*() is used. After the beginning of the segment has been located, it is read in sequential order into the host pinned memory.

The scheduler should determine the order of segment computation and launch the viewshed kernel for each segment.

All tests were performed on a desktop computer with the following component specifications:

- Processor: six hyper-threading cores, Intel(R) Core(TM) i7–3930K CPU @ 3.20 GHz.
- RAM memory: 32 GB
- Hard disk: SSD Corsair Force GT, size: 223 GiB (240 GB)
- CUDA device: NVIDIA GeForce GTX 680.
- Operating system: Linux (UBUNTU 11.04)
- File system: ext4

5.1. Input data preparation

For the input data, two DEMs of the Slovenian territory were used (see the fragment of the input data in Figure 11). The data are in raw format, each grid cell has a value that represents height above sea level in meters, which is stored as an integer value (4 bytes). The first DEM had a grid size of 25 × 25 m^2, whereas the second DEM had a 12.5 × 12.5 m^2 grid size. The memory sizes of the two maps were 0.48 and 1.89 GB, respectively. In the absence of larger maps for testing, we used up-sampled versions of the original map to verify the correct operation of the algorithm. All the test input data sets are listed in Table 1. We prepared ten data sets, with sizes ranging from 0.48 to 186 GB.

6. Variable types and CUDA GPU occupancy analyses

A CUDA-enabled GPU has its processing capability split into streaming multiprocessors (SMs), and the number of SMs depends on the actual device. Each SM has a finite number of 32-bit registers, a shared memory, a maximum number of active blocks, and a maximum number of active threads. These numbers depend on the CC of the GPU.

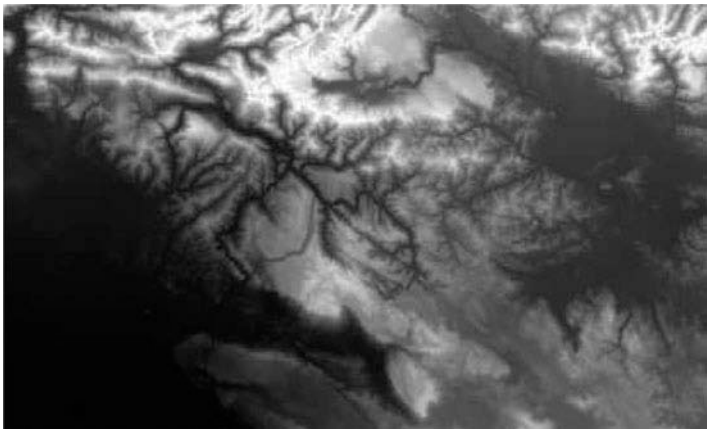


Figure 11. Input data set, DEM example.

Table 1. Test data sets.

Data set	Size (GB)	Grid-cell size	Columns (C)	Rows (R)	Grid cells
1[a]	0.48	25 × 25 m^2	14,081	8961	126,179,841
2[a]	1.89	12.5 × 12.5 m^2	28,161	17,921	504,673,281
3[b]	2.94	10 × 10 m^2	35,200	22,400	788,480,000
4[b]	7.52	6.25 × 6.25 m^2	56,320	35,840	2,018,508,800
5[b]	11.75	5 × 5 m^2	70,400	44,800	3,153,920,000
6[b]	18.36	4 × 4 m^2	88,000	56,000	4,928,000,000
7[b]	23.98	3.5 × 3.5 m^2	100,572	64,000	6,436,608,000
8[b]	47	2.5 × 2.5 m^2	140,805	89,605	12,616,832,025
9[b]	120	1.5625 × 1.5625 m^2	225,288	143,368	32,299,089,984
10[b]	186	1.25 × 1.25 m^2	281,610	179,210	50,467,328,100

Notes: [a]Data sets of the original DEM for a territory of Slovenia.
[b]Data set derived by interpolation from the second data set.

CUDA occupancy is defined as the ratio of the number of active warps per multiprocessor to the maximum number of active warps. Register usage, shared memory usage, or block size can be potential occupancy limiters.

With the first naive implementation, we used double type for all variables (Figure 12), which led to 37 registers used for each thread. Both variables, x and y, were increased on each iteration of the loop with the pre-calculated double value x_{step} and y_{step}. In Figure 13a, the impact on occupancy and kernel duration of varying the number of threads per block is depicted. The theoretical occupancy was calculated using Occupancy Calculator, provided by the NVIDIA Excel spreadsheet calculator

```
double x, y;  //variables must be double, otherwise errors, which increase with each iteration, become visible
double x_step, y_step;
calculate x_step and y_step
for (x=x_poi, y=y_poi; ;x=x+x_step, y=y+y_step)   //at each iteration, x and y are increased by double value
    ...
```

Figure 12. Naive implementation of the main loop in kernel.

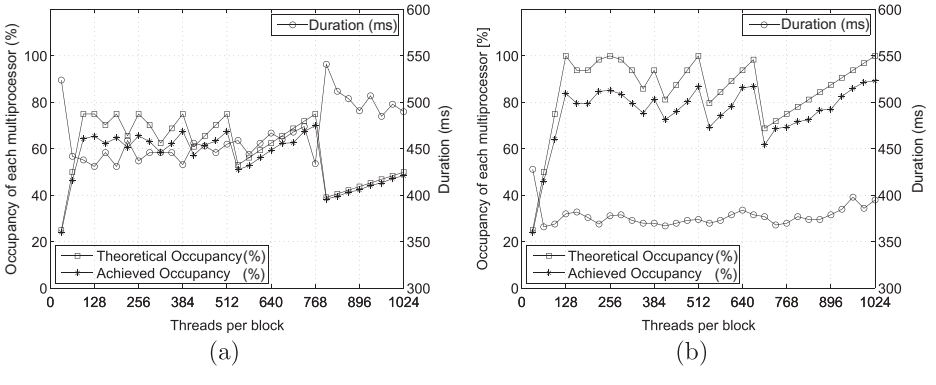


Figure 13. Impact of varying threads per block in a CUDA GPU viewshed kernel for the naive (a) and optimized (b) kernel loop.

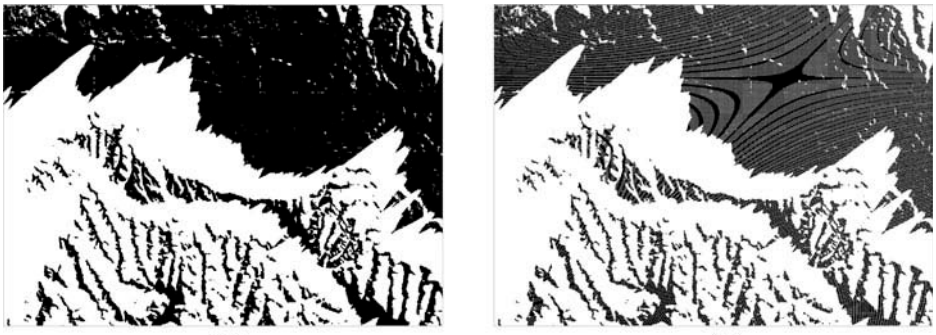


Figure 14. Optimized float implementation (a) versus naive float implementation (b) of the loop in the kernel.

(Occupancy Calculator 2013). The best theoretical occupancy value for the naive implementation of the viewshed kernel was 75%; the practically achieved value was 67.8%, with the threads per block set to the value of 768. The fastest duration of kernel execution was achieved at a value 128 threads per block (see Duration line in Figure 13a), for which we achieved an occupancy value of 65.9%.

There is, of course, the possibility of using float variables for this naive implementation and reaching an occupancy of near 100%, but the results in this case are unreliable, i.e., a Moirre pattern occurs (Figure 14b). The reason for this effect lies in the aggregation of errors through the loop because float (and also double) values have a limited set of possible values (*float* and *double* are not real numbers). In each iteration through the loop, we obtain a small error value that accumulates throughout the loop.

In the optimized implementation of the main loop inside the kernel, a new integer variable, *step*, was introduced. This variable is increased in each iteration only by a value of 1. Variables x and y are calculated from scratch at each iteration (Figure 15). With this procedure, the aggregation of errors through the loop is prevented.

In the optimized implementation, we used the float type for variables (Figure 15), which led to 28 registers used for each thread. A theoretical occupancy of 100% can be reached, and a practical occupancy of 90% was achieved (Figure 13b).

The results indicate that a setting of 512 threads per block in the optimized version is the optimal value for our device. All these parameters were measured using the NVIDIA profiler, which is part of the NVIDIA Nsight development platform (NVIDIA Profiler 2014).

```
float x, y; //variables can be float
float x_step, y_step;
int step;
calculate x_step and y_step
for (step=0; ;step++)       //at each iteration step, the integer value is increased
    x=x_step*step+x_poi;    //x and y are calculated at each iteration
    y=y_step*step+y_poi;
    ...
```

Figure 15. Optimized implementation of the main loop in the kernel.

7. Coalesced global memory access in the main kernel routine

An access to device global memory costs 400–600 clock cycles. GPUs can read 32-bit (for 8-bit data), 64-bit (for 16-bit data), or 128-bit words (for 32, 64, or 128 bit data, which is our case) from global memory into registers in a single instruction. Global memory bandwidth in a GPU is most efficient when simultaneous memory accesses threads in a half-warp (during the execution of a single read or write instruction), which can be coalesced into a single memory transaction (coalesced global memory access).

Coalesced global memory access can be strongly utilized for the north and south triangles from a POI. For the west and east triangles, coalesced memory access is utilized near the POI, where more threads traverse the same grid cell. The closer we get to the edge of the map, the less coalesced access is used. In Figure 16a, these effects are presented in a simplified form. The gray color represents data we intend to transfer from the GPU global memory to the kernel. For the north and south triangles, we can get all the data into the kernel with one read. For the east and west triangles, we need four read operations to get the same amount of data into the kernel.

In Figure 16b, throughput for our GPU, GeForce GTX 680, is measured. The graph represents coalesced global memory access with different offsets. We can see that the reading-data offset has very little impact on throughput. However, the stride size has a huge impact on throughput. This happens in the east/west triangles more often than in the north/south triangles of the map. Closer to the edge of the map, the throughput is smaller.

We measured the time difference between the computation time for the north/south triangles and the east/west triangles. On average, a north/south computation took half of the time of an east/west computation, regardless of the size of the map. However, the overall kernel computation time was reasonable, which can be seen in Figure 18.

8. Hard disk throughput compared to the cudaMemcpy throughput

We exploited the fact that the read/write (R/W) throughput of hard disks is significantly lower than the R/W throughput of CUDA memory copy. This fact helped us design the implementation of the viewshed algorithm, in which the overall computational speed depends almost exclusively on the R/W hard disk throughput.

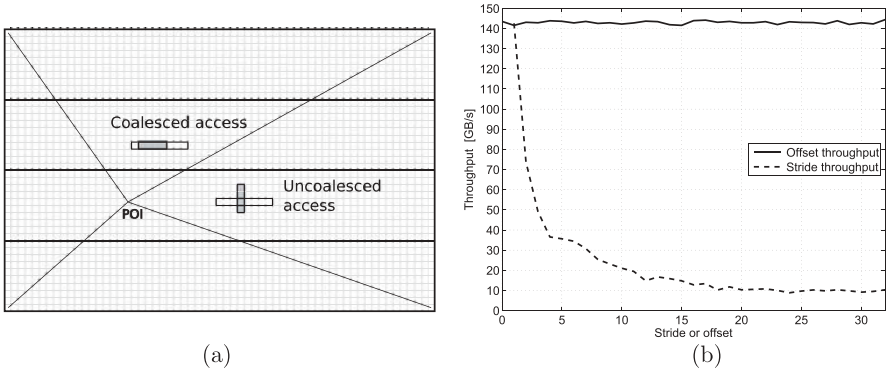


Figure 16. Coalesced memory access for north triangle versus uncoalesced memory access for east triangle (a). Measured throughput for coalesced (offset) and uncoalesced (stride) memory access for GeForce GTX 680 (b).

For our PC computer, we made R/W tests for different file sizes and two hard disks. The file was divided into the segments. Since the segment was relatively large in size (1 GB), the transfer of the whole file behaved like a stream. The average R/W throughput of different file sizes is shown in Figure 17a. From the graph, it can be clearly seen that the disk buffer helped out for file sizes that were less than approximately 100 MB. The influence of a page cache vanished at file sizes of more than approximately 10 GB. The page cache is a transparent cache of disk-backed pages kept in main memory (RAM) by the operating system for quicker access.

The data were transferred between the CUDA device and the CPU RAM with the *cudaMemcpy* function. For our PC computer, we made throughput tests for different segment sizes and for pageable and pinned CPU memory. From the graph in Figure 17b, one can clearly see that pinned memory achieved greater throughput for all segment sizes.

We are interested in file sizes of 1 GB and more. Figure 18 shows the timeline for small viewshed calculation (size of file is 1 G, segment size is 4). Since reading and

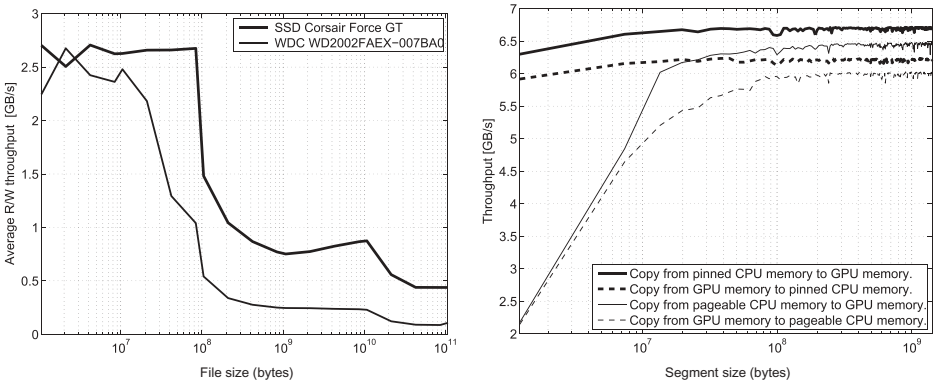


Figure 17. The average stream R/W throughput for two hard disks and different file sizes (a). cudaMemcpy throughput (Geforce GTX 680) for different segment sizes (b).

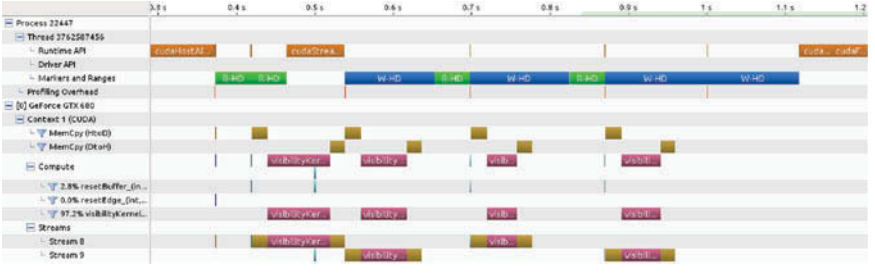


Figure 18. The Timeline View from NVIDIA visual profiler for concrete viewshed analysis for data divided into the four segments. In the **Runtime API** row are *cudaHostAlloc* call (left) and *cudaFreeHost* call (right). In the **Markers and Ranges** row are CPU reading from and writing to HD routines (R-HD and W-HD). In the **Stream 8** and the **Stream 9** rows are GPU *cudaMemCpy* and main viewshed kernel routines.

writing to hard disk is time consuming and the CPU and GPU can work simultaneously, we exploited this fact and let the GPU and CPU overlap. For this, we needed to consider a few rules. The main viewshed kernels cannot overlap due to the segmentation order computation (see Section 4). MemCpy(H to D), kernel and MemCpy(D to H) can start after the data have been read to RAM. Writing to hard disk can start after the calculated data has been copied to RAM.

To successfully perform overlapping, we used a double buffer and two CUDA streams for all calculation steps (double buffer for input data in the CPU and GPU and also double buffer for output data for the CPU and GPU were used). The first segment was calculated in the first buffer set and the first CUDA stream (Stream 8), the second segment was calculated in the second buffer set and the second CUDA stream (Stream 9), the third segment was calculated in the first buffer set and the first CUDA stream (Stream 8), and so on until the last segment.

For this particular case, we measured the following average throughputs: reading from hard disk was 2.5 GB/s, copy data to GPU was 5.7 GB/s, kernel throughput was 2.1 GB/s, copy data from GPU was 6.2 GB/s, and writing to hard disk was 0.9 GB/s. For different hardware configurations, different throughputs should be measured. Modern CUDA units can perform memory copy and kernel execution simultaneously. Also, two kernels can be run simultaneously – what we actually used in our case (simultaneous execution of the run visibility and reset buffer kernels). Based on these results, the scheduler should provide the right timing for routines to perform the optimal calculation timeline.

9. Impact of the number of segments on program execution time

In Figure 19, we illustrate the impact of varying the number of segments on the computation time. The test with a 1.89 GB data size was conducted with different segment quantities. The minimum number of segments was 3, and the test was conducted for up to 50. For each segment set, we ran the computation 50 times. We recorded the average, maximum, and minimum duration times for each segmentation. Figure 19a shows the impact of the number of segments on the calculation time for all kernels. Figure 19b

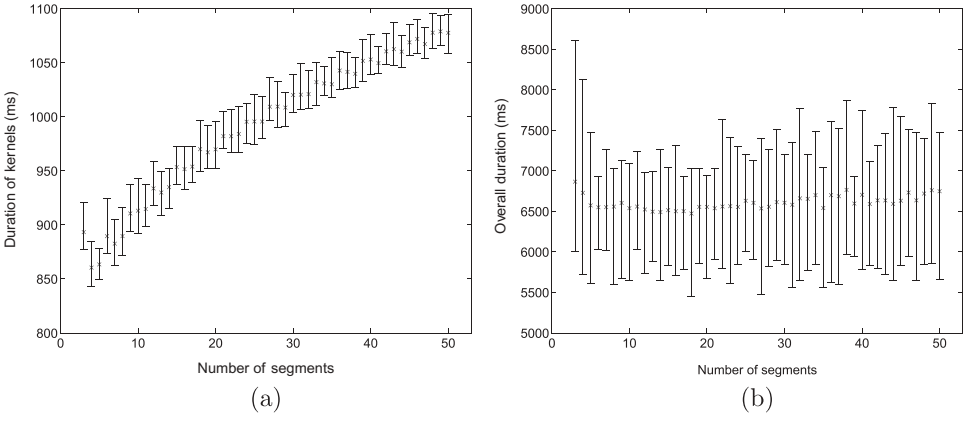


Figure 19. Impact of varying the number of segments on the computation time for the kernel duration time (a) and for the overall duration time (b).

shows the impact of the number of segments on the overall computation time. We observed that the number of segments had very little impact on the execution time for all kernels and almost no impact on the overall computation time. The overall computation time increased noticeably when the number of segments exceeded 10,000 (in this case, the overall computation time rises by approximately 10%). The reason for this result is the increased number of kernels starts.

The number of segments depends on the size of the maps and the amount of free memory. The minimum number of segments can be calculated using Equation (1). As a general rule, the appropriate number of segments should be kept as low as possible. However, care must be taken to leave enough free memory for other applications, which could run simultaneously with the viewshed calculation.

10. Algorithm comparison

Four different algorithms were tested and compared. The R3 algorithm (Shapira 1990), implemented in the GRASS GIS package, was chosen as a reference. The proposed parallel algorithm, r.cuda.visibility, and the sequential version of the proposed algorithm, r.cpu.visibility, were compared with the reference algorithm. In addition, we conducted the same tests with the TiledVS viewshed implementation (Ferreira *et al.* 2012).

The Earth-curvature correction was not integrated in the reference algorithm R3. For the purposes of comparison, this correction was also disabled in the other algorithms (r.cpu.visibility, r.cuda.visibility, and TiledVS) during the tests.

10.1. Experimental results

The experimental results for different data sizes and different algorithms are shown in Table 2. From the test results, it can be observed that the computation time of the R3 algorithm is strongly dependent on the number of visible grid cells. Therefore, the R3 algorithm was tested and compared using two different parameter values (a pit and a peak POI). In contrast, the computation time of the proposed algorithm is independent of the number of visible grid cells (independent of the POI position: pit or peak).

Table 2. Comparison of computation time for four implementations of viewshed algorithms.

		R3	TiledVS	r.cpu.visibility		r.cuda.visibility		
Data set	Size (GB)	all (pit/peak)[a]	all[a]	all[a]	calc.[a]	all[a]	kernel[a]	segments[b]
1	0.48	310/2776	28	13	11	1.8	0.2	1
2	1.89	1426/21,045	115	52	46	7	0.7	3
3	2.94	–	196	84	77	10	1.3	4
4	7.52	–	453	229	216	21	3.5	9
5	11.75	–	670	375	352	31	5.5	13
6	18.36	–	1014	626	588	52	9.5	21
7	23.98	–	1317	892	831	91	13	27
8	47	–	2635	2266	2106	265	29	54
9	120	–	13,498	9645	9252	1051	112	140
10	186	–	–	20,474	19,865	1922	235	218

Notes: [a]All values of duration are in seconds.
[b]The indicated number is the number of segments in the calculation with our hardware.

For the sequential version of the proposed algorithm (r.cpu.visibility), the total calculation time (all) and calculation viewshed time (calc.) are presented. For the parallel version of the proposed algorithm (r.cuda.visibility), two values are presented: the total computation time (all) and the kernel computation time (kernel). The total time (all) represents the overall duration of the operation (kernel computation time and all data transferred from hard disk to GPU and back). The number of segments (for our computer environment) into which the map has been divided is also presented.

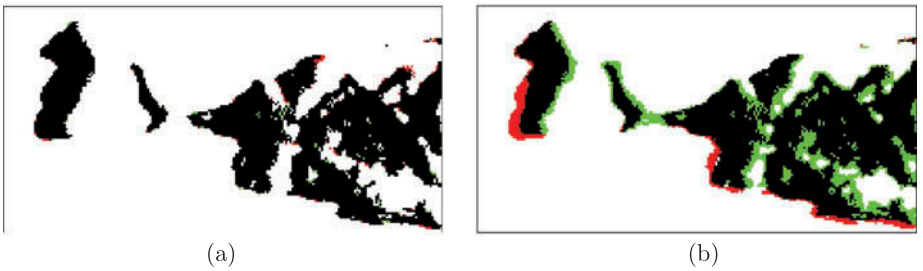
We also conducted tests with the sequential TilesVS package (Ferreira *et al.* 2012), which is based on subdividing the terrain into blocks that are stored in a special data structure managed as a cache memory. Our approach results in a better computation time than TilesVS, even with the sequential r.cpu.viewshed version.

From the time results of r.cpu.visibility, we can conclude that the calculation part of r.cpu.visibility took more than 90% of the duration time. Memory reading and writing took less than 10% of the time (strictly sequential routines). Using Amdahl's law, we can make a significant speed improvement if we improve the calculation time with parallelism (CUDA). The drawback of using CUDA is the additional memory required to copy to and from the device. The speed-up achieved with r.cuda.visibility improves r.cpu.visibility by a factor of 10.

10.2. Accuracy comparison of algorithms

A detailed view of the viewshed analysis using the same input data set with the r.los (R3), the r.cuda.visibility (R2) and the TiledVS implementations is shown in Figure 20. Testing the accuracy of the r.cpu.visibility was implicitly done, because the r.cpu.visibility produces exactly the same result as the r.cuda.visibility implementation.

A test with 10 representative POI locations on a 1.89 GB data size (28,161 cols × 17,921 rows, each grid cell is 12.5 × 12.5 m^2) was conducted. Five of them are located on hilltops, from which the view extends to all four sides of the sky; and five of them are located on hill slopes with a relatively good view. The maximum distance was set to 25 km. For each test case, we counted the number of grid cells showing different results from the compared algorithms. In Figure 20, a fragment of 2625 × 1375 m^2 is shown.



(a) (b)

Figure 20. Panel (a) shows the comparison between the r.los (R3) and the r.cuda.visibility (R2) algorithms. Panel (b) shows the comparison between the r.los (R3) and the TiledVS. The black grid cells represent the area of visibility for both algorithms. The green grid cells represent those visible only for the r.cuda.visibility (a) or the TiledVS (b). The red grid cells represent those visible only for the r.los.

From Figure 20a, we can see that the r.los and the r.cuda.visibility achieved almost the same result. Both algorithms agreed on the visibility of 99.89% of grid cells in the best case and of 99.52% in the worst case. This is expected because R2 is an approximate algorithm, whereas R3 is an exact one. Similar accuracy differences between R2 and R3 were reported by Franklin and Ray (1994). We exploited this minimal accuracy difference to achieve the remarkable execution-time speed-ups presented in the previous section.

In Figure 20b, we show the results of the comparison between r.los and TiledVS. After testing with the same 10 representative POIs, the algorithms agreed on the visibility of 97.52% of grid cells in the best case and of 93.22% in the worst case.

11. Related work

Several approaches have been proposed for a parallel implementation of the viewshed calculation. In Xia *et al.* (2011), the simulation results of two algorithms were presented: Matrix Traversal and Ray Traversal. Both algorithms were implemented in sequential and parallel modes, resulting in four different implementation types. If the grid cell is traversed by multiple rays, the logical 'OR' function was used between the rays to determine the visibility criteria, which generates overly optimistic results. The presented results are calculated on a map size of 4996 × 3088 grid cells, which is smaller than our first data set (see Table 1).

In Zhao *et al.* (2013), two parallel algorithms for the viewshed analysis of grid-based terrains were introduced. These algorithms, namely, the GPU-fine and GPU-coarse, were based on the well-known R3 sequential algorithm (Shapira 1990). The computational efficiency of these algorithms has been demonstrated on various DEM sizes, from 1.49 to 20.54 GB, and compared to the original R3. The computation time of these algorithms depends on the number of visible points. However, the approach taken by these researchers is fundamentally different from our approach because they worked with the R3 algorithm.

In Gao *et al.* (2011), a linear-interpolation approach between neighboring grid cells was described. The authors present two DEM processing optimization solutions on a GPU. The first solution assigned data to different memory locations on the GPU. The second solution used instruction optimization for SM. The GPU texture memory was used for the input map data set, resulting in an increased processing speed and hardware interpolation between the grid-cell values (limited interpolation methods by the device hardware). The maximum size of the map was restricted to the amount of 2D texture memory available. The presented report was only for two map sizes, 366 × 372 and 2847 × 3717 grid cells. In contrast to our approach, the authors did not provide a solution for large data maps.

In Strnad (2011), the Earth-curvature consideration was presented. For the input map data set, texture memory was used, leading to limitations in map size and interpolation. The computation time of relatively small maps was reported (maximum of 8192 × 8192 grid cells, which is smaller than our first data set; Table 1).

In the research report Osterman (2012), a basic approach was presented, suggesting further investigation and leading to the optimizations presented in the present article. With the additional improvements in memory-handling efficiency (Section 4.1), near-cell interpolation, the nearest-ray strategy (Section 4.5), and optimization (NVIDIA Corporation 2013), a reduction in computation time and improved accuracy and flexibility of the algorithm were achieved.

In Amanatides *et al.* (1987), the authors proposed a fast voxel traversal algorithm for ray tracing, which searches objects in voxels. If the object is not present in the voxel, the computation for that voxel can be abandoned and the computation time is faster. In R3, the longest line-of-sight (LOS) functions are calculated first. Along LOS, traversing the voxels, obstacles are searched. If no obstacles are found in the voxels, those voxels are skipped from further LOS computation and computation speed is accelerated. In R2, when the LOS functions are calculated, the computation of viewshed is finished. Therefore, the fast voxel traversal algorithm does not bring any benefit to R2 computation.

In Izraelevitz (2003), the authors proposed a bridge between the direct method based on LOS analysis and the fast, but less accurate, XDraw method. Their implementation was sequential. The best method for implementing their algorithm in a parallel manner is an open challenge.

12. Conclusion, discussion, and future work

In this article, we introduced a modification of the well-known R2 algorithm in a parallel implementation, with the possibility of processing large maps in a computer with a relatively small memory size. The focus is on flexibility, IO efficiency, and the best possible computation time. The essential reason for the improved computation times of our algorithm, as compared to other approaches, is the use of coalesced memory access to global memory on the device (Davidson and Jinturkar 1994) during terrain segmentation. Therefore, IO efficiency involves a single reading and writing of the map in and out of memory, coalesced global memory access, and double buffering, with overlapping of CPU and GPU routines.

The type of interpolation (e.g., linear) can have a major effect on visibility. In addition, the influence of the type of the interpolation depends on the type of terrain. In this work, we used only linear interpolation. We leave experimentation with other methods for future work.

In the mobile telecommunications industry, as a starting point, our research allows for real-time terrain analysis for the complex task of mobile network planning. It is possible to increase the network planning efficiency at lower costs by using real-time analysis tools. This algorithm allows for the optimization of the locations of mobile network base stations, with computations of numerous scenarios and effective planning of long-haul microwave links, replacing less accurate, traditional methods (floating balloons or mirror targeting).

A variety of analysis algorithms could be based on the computationally efficient kernel that forms the core of the presented algorithm. The presented algorithm, r.cuda.visibility, is currently part of the GRASS-RaPlaT project (Hrovat *et al.* 2010, Benedičič *et al.* 2014). RaPlaT is an open-source radio planning tool for GSM, UMTS, and LTE mobile systems. With the increasing demand placed on the analysis capabilities of modern mobile networks, our parallelization approach can be used to further improve RAPlaT tools.

References

Amanatides, J. and Woo, A., 1987. A fast voxel traversal algorithm for ray tracing. *In*: G. Maréchal, ed. *Proceedings of eurographics,* 24–28 August, Amsterdam. Elsevier Science Publish Company, 3–10.

Benedičič, L., *et al*., 2014. A GRASS GIS parallel module for radio-propagation predictions. *International Journal of Geographical Information Science*, 28 (4), 799–823.

Davidson, J.W. and Jinturkar, S., 1994. Memory access coalescing: a technique for eliminating redundant memory accesses. *ACM SIGPLAN Notices*, 29 (6), 186–195. doi:10.1145/773473.178259.

Dodd, H.M., 2001. *The validity of using a geographic information system's viewshed function as a predictor for the reception of line-of-sight radio waves*. Thesis (PhD). Virginia Polytechnic Institute and State University.

Ferreira, C.R., *et al.*, 2012. More efficient terrain viewshed computation on massive datasets using external memory. *In*: *Proceedings of the 20th international conference on advances in geographic information systems*, 6–9 November, Redondo Beach, CA. New York: ACM, 494–497.

Franklin, W.R. and Ray, C., 1994. Higher isn't necessarily better: visibility algorithms and experiments. *In*: T. C. Waugh and R. G. Healey, eds. *Advances in GIS research: sixth international symposium on spatial data handling*, [sponsored by] International Geographical Union Commission on GIS, Association for Geographic Information, 5–9 September, Edinburgh. London: Taylor & Francis, 751–770.

Franklin, W.R. and Vogt, C., 2006. Tradeoffs when multiple observer siting on large terrain cells. *In*: A. Riedl, W. Kainz, and G.A. Elmes, eds. *Progress in spatial data handling*. Berlin: Springer, 845–861.

Gao, Y., *et al.*, 2011. Optimization for viewshed analysis on GPU. *In*: *19th international conference on geoinformatics*, 24–26 June, Shanghai. IEEE, 1–5.

GRASS GIS, 2013. GRASS (Geographic Resources Analysis Support System) GIS (Geographic Information System), *The world's leading Free GIS software* [online]. Available from: http://grass.osgeo.org [Accessed March 2013].

Hrovat, A., *et al.*, 2010. An open-source radio coverage prediction tool. *In*: *Proceedings of the 14th WSEAS international conference on communications*, 23–25 July, Corfu Island. Athens: WSEAS Press, 135–140.

Izraelevitz, D., 2003. A fast algorithm for approximate viewshed computation. *Photogrammetric Engineering and Remote Sensing*, 69 (7), 767–774. doi:10.14358/PERS.69.7.767.

Kukushkin, A., 2004. *Radio wave propagation in the marine boundary layer*. Weinheim: WILEYVCH Verlag GmbH & Co. KGaA.

Kvarfordt, K.L., 2010. *Planning for closure of the Logan City/Cache County landfill and surrounding landscape*. Master's thesis. Utah State University.

Lake, M.W. and Woodman, P.E., 2003. Visibility studies in archaeology: a review and case study. *Environment and Planning B: Planning and Design*, 30 (5), 689–707. doi:10.1068/b29122.

Lake, M.W., Woodman, P.E., and Mithen, S.J., 1998. Tailoring GIS software for archaeological applications: an example concerning viewshed analysis. *Journal of Archaeological Science*, 25 (1), 27–38. doi:10.1006/jasc.1997.0197.

Magalhães Salles, V.G., Andrade Marcus, V.A., and Franklin, W.R., 2011. Multiple observer siting in huge terrains stored in external memory. *International Journal of Computer Information Systems and Industrial Management (IJCISIM)*, 3, 143–149.

NVIDIA Corporation, 2013. *CUDA C best practices guide* [online]. Available from: http://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html [Accessed March 2013].

NVIDIA Profiler, 2014. *NVIDIA visual profiler – NVIDIA developer zone* [online]. Available from: https://developer.nvidia.com/nvidia-visual-profiler [Accessed April 2014].

Occupancy Calculator, 2013. *CUDA GPU occupancy calculator* [online]. Available from: Developer.download.nvidia.com/compute/cuda/CUDA_Occupancy_calculator.xls [Accessed August 2013].

Osterman, A., 2012. Implementation of the r. CUDA. los module in the open source GRASS GIS by using parallel computation on the NVIDIA CUDA graphic cards. *Elektrotehniški Vestnik*, 79 (1–2), 19–24.

Pinned Memory, 2013. *Choosing between pinned and non-pinned memory* [online]. Available from: http://www.cs.virginia.edu/~mwb7w/cuda_support/pinned_tradeoff.html [Accessed March 2013].

Shapira, A., 1990. *Visibility and terrain labeling*. Thesis (PhD). Department of Electrical, Computer, and System Engineering, Rensselaer Polytechnic Institute, Troy, NY.

Strnad, D., 2011. Parallel terrain visibility calculation on the graphics processing unit. *Concurrency and Computation: Practice and Experience*, 23 (18), 2452–2462. doi:10.1002/cpe.1808.

Tabik, S., Zapata, E.L., and Romero, L.F., 2013. Simultaneous computation of total viewshed on large high resolution grids. *International Journal of Geographical Information Science*, 27 (4), 804–814. doi:10.1080/13658816.2012.677538.

Toma, L., 2012. Viewsheds on terrains in external memory. *SIGSPATIAL Special*, 4 (2), 13–17. doi:10.1145/2367574.2367577.

van Kreveld, M., 1996. Variations on sweep algorithms: efficient computation of extended viewsheds and class intervals. UU-CS, (1996-22).

Xia, Y.J., Kuang, L., and Li, X.M., 2011. Accelerating geospatial analysis on gpus using CUDA. *Journal of Zhejiang University-Science C*, 12 (12), 990–999. doi:10.1631/jzus.C1100051.

Zhao, Y., Padmanabhan, A., and Wang, S., 2013. A parallel computing approach to viewshed analysis of large terrain data using graphics processing units. *International Journal of Geographical Information Science*, 27 (2), 363–384. doi:10.1080/13658816.2012.692372.