Parallel GPU Processing for Fast Radio Signal Propagation Computation in GRASS-RaPlaT

IGOR OZIMEK, ANDREJ HROVAT, ANDREJ VILHAR, TOMAŽ JAVORNIK
Department of Communication Systems
Jozef Stefan Institute
Jamova 39, 1000 Ljubljana
SLOVENIA
igor.ozimek@ijs.si   http://www-e6.ijs.si

Abstract: - Radio propagation simulation tools are important for prediction and verification of the radio signal coverage by individual transmitters or transmitter networks such as mobile phone cellular networks. In the case of a large geographic area with a relative high resolution, the simulation can become computationally demanding, taking a considerable amount of time to accomplish. Parallel processing can be used to speed up the computation and shorten the response time. We used the GPGPU (General-Purpose Computing on Graphics Processing Units) approach for the open source GRASS-RaPlaT radio propagation tool. By using OpenCL, we modified the existing radio propagation model modules for GPU (Graphics Processing Unit) execution, achieving multiple times processing speedup for computationally intensive modules. In the article, we present our GPU parallelization approach and analyze the results and conditions that must be fulfilled to successfully employ GPU computation and achieve a considerable computation speedup.

Key-Words: - Radio propagation simulation, GRASS, RaPlaT, parallel processing, GPU computing, OpenCL

1 Introduction

Radio propagation planning is important for setting up a transmitter or building a whole transmitter network, as well as later during the normal transmitter/network operation for predicting the signal strength at various locations. Numerous professional tools exist for this purpose, with their common drawback being their high price and inability of the user to add his/her own propagation model in a simple way. For these reasons, open source solutions have been developed. Usually, they have limited functionalities compared to the professional commercial tools; however, they are affordable (free) and allow the user to add additional new propagation models if needed. One such tool is GRASS-RaPlaT (or shortly RaPlaT).

Radio coverage computation in RaPlaT is raster-based. Although the computation for each point is not too complex, the sheer number of raster points can make it quite demanding. E.g., an area of 100×100 km^2 with resolution of 25 m would contain 16M raster points; improving resolution to 5 m would produce 500M points. The parallel nature of the raster-based algorithm calls for a massive parallelization approach, such as GPGPU, which can result in a multiple times speedup of the whole computing process.

In the next chapter, GRASS-RaPlaT is briefly presented. Chapter 3 gives a short description of possible approaches to parallelization with a special emphasis on the GPU computing. Chapter 4 presents the results, analyzes them, and determines the conditions that must be fulfilled for a successful employment of GPU processing. In chapter 5, we summarize the results and draw the conclusions.

2 GRASS-RaPlaT

RaPlaT (Radio Planning Tool for GRASS) [1,2,3,4] is an add-on to the well-known open source GRASS GIS (Geographic Resources Analysis Support System / Geographic Information System) [5,6]. Since radio propagation simulations are always performed within a geographic area with a known terrain profile, a GIS system is a reasonable framework for this task. GRASS is very suitable for this purpose since it is a well-established open source software tool with a modular structure that allows adding of additional user-written modules in a simple way.

RaPlaT consist of a number of GRASS-compatible modules. Its core modules perform three distinct tasks needed for radio coverage computation (Fig. 1).

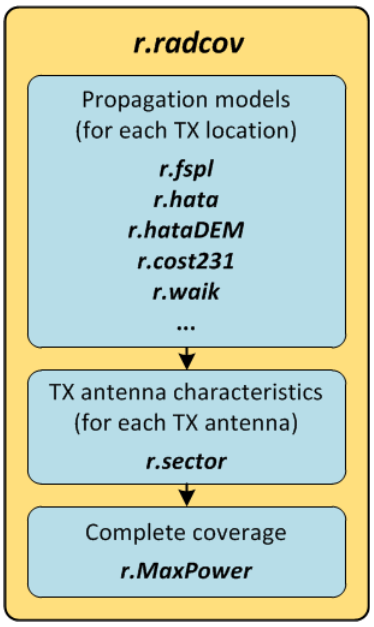
r.radcov
Propagation models (for each TX location)
r.fspl
r.hata
r.hataDEM
r.cost231
r.waik
...
TX antenna characteristics (for each TX antenna)
r.sector
Complete coverage
r.MaxPower

Fig. 1. RaPlaT modular structure

The first task is computation of the undirected (isotropic) signal fading according to a chosen radio signal propagation model. A considerable (and growing) number of different radio propagation models are supported, each implemented as a separate GRASS module written in the C language:

- Free space model.
- Okumura-Hata model (the standard and COST231 versions).
- An extended Okumura-Hata model containing an additional edge diffraction algorithm for NLOS (Non-Line-of-Sight) situations and land-usage related fading.
- Walfisch-Ikegami model (developed in the framework of the COST231 project).
- Longley/Rice model (ITM – Irregular Terrain Model).
- ITU-R Recommendation P.1546-4.
- Urban model.

The next task, performed by the *r.sector* module, is to take into account the actual transmit antenna radiation pattern specified by an MSI-format file and modify the previously computed isotropic path loss map accordingly.

The final task, performed by the *r.MaxPower* module, is to use the actual transmit power value to compute the receive signal strength map, which can be done not only for one transmitter but also for a whole radio transmitter network.

A RaPlaT user does not need to call all these modules individually. He can only call the *r.radcov* module (written in Python), which reads a CSV-format file describing the transmitter network configuration and calls the previously described modules as necessary, possibly concurrently on a multi-core processor machine.

The usability of RaPlaT has been proven by the fact that it has been developed jointly with, and is used by the Slovenian mobile operator Telekom as the core radio coverage computation engine for their own in-house developed radio planning tool.

3 Parallel Computing and GPU

The computational speed of a conventional SISD (Single Instruction Single Data) processor is limited by its clock frequency, number of clock cycles required for execution of each instruction, and the instruction complexity. The other important limiting factor is the time required to access memory for reading and storing instructions and data, which can be minimized by employing a fast on-chip cache memory.

For many years, SISD processor capabilities were being increased by increasing the clock frequency and the execution unit capabilities with techniques such as instruction-level parallelism (instruction pipeling, out-of-order execution, branch prediction, etc.) and hyper-threading (for better utilization of various CPU functional blocks by concurrently running two execution threads). This approach reached its limits around 2003, when increasing of the clock frequency slowed down (largely governed by the processor thermal power dissipation/cooling limitations), and no much execution speed improvement could be achieved by enlarging the CPU complexity. However, the maximum number of transistors per chip still keeps rising and they are employed to build higher level parallel structures – multi-core processors.

Another parallelization technique, known already from the early years of digital computers, uses vector (co)processors – SIMD (Single Instruction Multiple Data). They allow parallel execution of a single instruction (such as an integer or floating point arithmetic operation) on multiple data, which can be used to speed up the processing of vectors or matrices, such as are nowadays most often found in media content (e.g. a picture, consisting of raster points, each of the points having three color components). Due to the importance of the media content processing, SIMD extensions are a standard component of modern processors (e.g. MMX, SSE, and lately AVX), as are the general fast FPU’s (Floating Point Units).

While general processors (CPUs) employ a certain amount of parallelism, it is in no way massive. Multi-core processors can integrate only a very limited number of cores on a chip because of the problems with fast concurrent memory access and cache coherency. This is a high level parallelism of the MIMD type, which can concurrently execute multiple independent processes or relatively high level threads of a single process. SIMD extension, on the other hand are low level parallel structures, but most suitable for completely identical operations on vector/matrix data. The current implementations usually support only small vector sizes, e.g. four floating point values, suitable for fast media content processing.

3.1 GPU

With the advent of capable graphic adapters utilizing programmable GPUs, a new vehicle for massively parallel computing was born. GPUs have been created for performing massive parallel computations on rasterized images. Their development and production was driven by the massive market of computer games, which enabled their relatively low prices. As the GPUs became more and more general-purpose, capable and accessible for programming, they started to be used for general computing (GPGPU), first by using the existing languages for graphic processing (such as OpenGL). Later, dedicated GPGPU languages were developed.

Programming for GPU is primarily based on the C language, although some other languages are often supported. NVIDIA is recognized as the pioneer in the GPGPU field with its CUDA parallel computing platform, programming model and software tools (compiler etc.) [7]. A drawback of CUDA is that it only supports NVIDIA’s GPUs. An independent effort of Khronos Group industry consortium (including NVIDIA among many others) resulted in OpenCL (Open Computing Language) [8], an open standard for parallel programming of heterogeneous systems.

Both CUDA and OpenCL use a similar programming model. NVIDIA calls it SIMT (Single Instruction, Multiple Threads), where multiple independent threads execute concurrently using a single instruction on multiple data. However, this is different form vector processor SIMD and enables execution of conditional branching (an IF statement in C), where for each branch only the corresponding subset of data is processed and the rest of them wait for their turn (in an alternative branch or after the branches merge again). The parts of a program (usually written in C) to be executed on GPU are programmed as separate C-like modules. They define execution of a single thread for a single data element.

There exists another approach to GPU programming, OpenACC (Open Accelerators) [9]. Here, the procedures for GPU are not coded as separate C-modules. Instead, the program is written completely in normal C-language (or some other supported language). The OpenACC compiler is instructed which parts to execute on GPU (usually some parallelizable loops) by additional *#pragma* statements in the code (which would be ignored by a standard compiler).

The number of hardware platforms (i.e. GPU processors) manufacturers are rather limited, with AMD (formerly ATI) being the only other manufacturer of devices with capabilities comparable to those of NVIDIA.

4 Parallelization, results and analysis

In this chapter, our implementation of the GPU acceleration for two GRASS-RaPlaT modules and the achieved performance are presented.

4.1 Development tools

At first glance, the use of OpenACC seems tempting as it enables GPU execution of C programs by only inserting *#pragma* directives to instruct the compiler which parts should be executed on GPU. However, this approach offers much less control over the way a program is executed on GPU, and in reality it is far from trivial to do it right and efficiently. Hence, we took the other approach, and chose OpenCL due to its hardware manufacturer independence.

4.2 Hardware

High performance GPU adapters are produced by NVIDIA and AMD. We decided on NVIDIA, due to its leading role in the GPGPU field. Their graphic adapters can be divided into two groups, one for the massive consumer market, and the other for professional use, which includes also the Tesla family specifically designed for GPU computing. All the cards generally share the same GPU microarchitecture but with different capabilities, especially considering the number of computing cores.

The GPU implementation and testing presented in this article were performed with GTX 580, a high performance consumer-grade graphic adapter. Some basic properties of this adapter are summarized in Table 1. The main reason for choosing it was its much lower price compared to professional adapters while having the capabilities comparable to much more expensive professional-grade products (with the exception of the slower double precision floating point operations). GTX 580 uses the Fermi GPU microarchitecture. 32-bit single precision floating point units are an integral part of its computing cores, and they are also capable of 64-bit double precision floating point processing with half the speed of the single precision operations. However, for the consumer-grade graphic adapters, the double precision computations have been downgraded by a factor of 4, becoming 8x slower compared to the single precision operations.

Table 1. Some basic properties of the GTX 580 adapter used for testing

No. of cores	512
GPU / Shader clock frequency	793 MHz / 1586 MHz
GPU adapter memory size	3 GB
GPU memory bandwidth	192.4 GB/s (384-bit, 4008 MHz)
GPU adapter bus	PCI-E 2.0 x 16 (max. 8 GB/s)
GFLOPS (SP / DP)	1624 / 203

In the meantime, NVIDIA has developed a new improved GPU microarchitecture named Kepler, with the main emphasis on the reduced power consumption and better games support. The 64-bit floating point units are not integrated into the computing cores anymore but are separate entities available in suitable quantities. For consumer-grade adapters, their number is relatively low compared to the number of cores, resulting in the double precision floating point computations being typically 24x slower than the single precision ones. Hence, if a consumer-grade graphic adapter is used for general computing, a Fermi-based adapter might still be a better choice.

4.3 OS environment

NVIDIA supports MS Windows, Linux and Mac OS X operating systems. GRASS also supports all these operating systems; however, until now RaPlaT has only been compiled and tested under Linux (Ubuntu), which was also used for tests presented in this article.

A limitation of GPU computing with consumer-grade adapters is their run time limit of a few seconds on kernels. This is implemented in the graphic adapter driver with the aim to keep the graphic card responsive for its supposedly primary function – displaying picture on the monitor. Unfortunately, in MS Windows this limitation is active even if no monitor is connected to the adapter (this can be changed by tweaking the registry). The much more expensive adapters from the Tesla family can use another version of driver without this limitation. In Unix (Ubuntu in our case), the situation is better and an adapter without a monitor operates without the run time limit on kernels.

Another limitation in MS Windows (but not in Linux) is that GPU computation can only be run from the local physical workstation's console and not from a remote desktop session.

4.4 Parallelized RaPLaT modules

Two RaPlaT propagation models have been modified for execution on GPU using OpenCL tools from NVIDIA: the basic Okumura-Hata model – module *r.hata*, and the extended Okumura Hata model – module *r.hataDEM*.

4.4.1 r.hata

The Okumura-Hata model [10] is a rather simple model, which ignores any obstacles between the transmitter and receiver as well as the land-usage related fading. It needs a DEM (Digital Elevation Map) with terrain heights above sea in [m], like the one represented in Fig. 2 for the region around the city of Ljubljana.

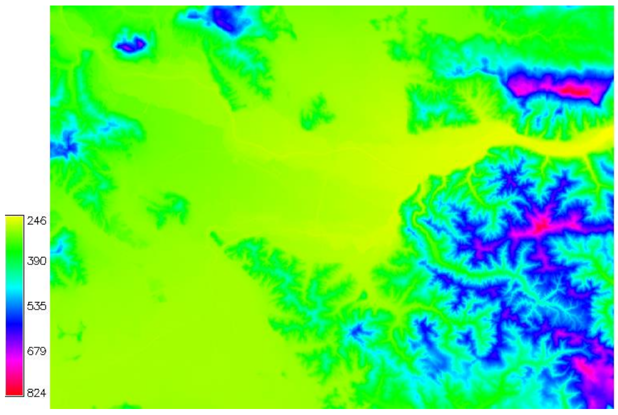

Fig. 2. DEM for the region around the city of Ljubljana

The computed path loss fading map (in [dB]) is relatively simple (and hence not so very accurate for complex environments), as represented in Fig. 3.

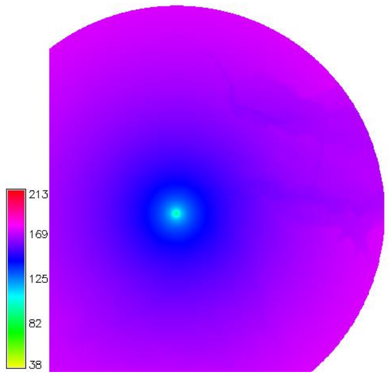

Fig. 3. Path loss map computed by *r.hata*

Some computations are done only once at the beginning of the module execution and do not contribute significantly to the overall computing time. The computations that matter for parallelization are those executed for each raster point. In *r.hata* this is primarily the following equation:

$$L_U = L_{Uconst} - 13.82 \cdot \log(h) + \\ + (44.9 - 6.55 \cdot \log(h)) \cdot \log(d) \tag{1}$$

where L_U is the path loss in [dB], L_{Uconst} is the precomputed constant part of the path loss in [dB], h is the effective transmit antenna height (relative to the receive point height) in [m], and d is the distance between the transmitter and the receive point in [km].

According to Eq. (1), for each raster point two base-10 logarithms must be computed (log(h) need only be computed once), two multiplications and three additions/subtractions. The computations have a very regular structure with calculation for each raster point being performed independently of the other points. As such, it would be ideal for implementation on GPU processor. However, its computations are rather simple and can be performed quickly, so the GPU processing overhead (fetching and storing the data from/to the main graphic adapter memory, etc.) could become a bottleneck. The key to success is that there are a sufficient number of raster points being processed in parallel at every moment so that GPU can suitably schedule their execution (computing some points while others are waiting for data to be fetched/stored). Our tests on GTX 580 (with 512 computing cores) showed that around 10000 computation threads had to be lunched in parallel to achieve efficient GPU utilization.

The *r.hata* computational complexity is proportional to the number of raster points. If we denote the linear map dimension (e.g. x or y, keeping the ratio between them approximately the same) by n, its order of complexity is $O(n^2)$, which is optimal for image raster processing.

4.4.2 r.hataDEM

The *r.hataDEM* is an extended Okumura-Hata model. For LOS locations, it uses the basic Okumura-Hata model. For NLOS locations, it computes the path loss according to the edge diffraction effect on the highest obstacle relative to the direct line between the transmitter and the receive point [11,12]. Additionally, it takes into account the effect of the land-usage (buildings, forests, etc.) at the receive locations. This is given in [dB] by an additional path loss raster map called clutter map, such as the one in Fig. 4 for the region around the city of Ljubljana.

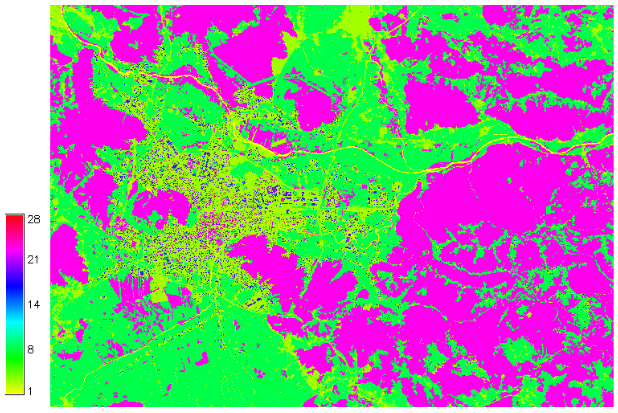

Fig. 4. Clutter map for the region around the city of Ljubljana

Compared to the *r.hata* path loss map, the resulting map (in [dB]) is much more detailed and accurate, as can be seen in Fig. 5.

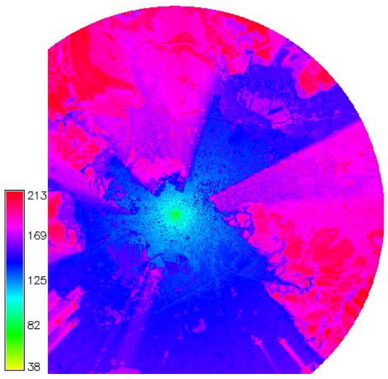

Fig. 5. Path loss map computed by *r.hataDEM*

For LOS locations, the computation is basically the same as for *r.hata*, described by Eq. (1). For

NLOS locations, the knife edge diffraction loss is calculated according to Eq. (2) and (3):

$$L_{ke} = -20 \cdot \log \frac{1}{\pi \cdot \nu \cdot \sqrt{2}} \tag{2}$$

$$\nu = h \cdot \sqrt{\frac{2 \cdot (d_1 + d_2)}{\lambda \cdot d_1 \cdot d_2}} \tag{3}$$

where L_{ke} is the path loss due to the knife edge diffraction effect in [dB], h is the height of the highest obstacle above the direct line between the transmitter and receiver, $d1$ and $d2$ are the distances of the mobile and base stations from the obstacle, and λ is the radio signal wavelength.

What is really important for computational complexity is the fact, that for each point a LOS/NLOS check has to be performed first, which seeks the highest obstacle along the direct line between the transmitter and the receive point. It performs some initial computations that add to those described by Eq. (2) and (3), but this is not really important. The most computationally demanding part is traveling along the transmitter-receiver line, searching for obstacles. Although only some basic computations are performed in this loop (additions, subtractions, multiplications), it rises the order of complexity one step higher, to $O(n^3)$, making *r.hataDEM* much slower than *r.hata* especially for a large number of raster points. It also makes the algorithm irregular in the sense that the computation for each receive point depends not only on the corresponding DEM raster point, but on all points on the line between the transmitter and the receive point.

4.5 Execution performance

The execution performance of the GPU-accelerated version of the *r.hata* and *r.hataDEM* modules and their original CPU-only versions where compared for two geographic regions. The first one was the whole Slovenia region. The corresponding DEM covered an area of 352×224 km^2 with resolution of 25 m, with the resulting raster size of 14081×8961= 126,179,841 points. Due to its large size and the computational complexity of *r.hataDEM*, it was only used for *r.hata*.

The second region was a smaller region around the city of Ljubljana, but with a better resolution. The corresponding DEM covered an area of 26.9×19.2 km^2 with resolution of 5 m, with the resulting raster size of 5381×3841 = 20,668,421 points. It was used for both modules, *r.hata* and *r.hataDEM*.

In both cases, the transmitter was placed in Ljubljana at the IJS location (an actual mobile phone base station location).

4.5.1 r.hata

Execution performance for the Slovenia region is presented in Table 2 for CPU execution (with computations performed with the usual double floating point precision) and GPU execution with single and double floating point precision. The results computed with single and double precision were of course not identical, but the differences were completely negligible for our use.

Table 2. *r.hata* for Slovenia region

	Calculations	Complete
CPU (DP)	12.54s	24.00s
GPU SP	0.47s (26.5×)	11.92s (2.0×)
GPU DP	0.86s (14.6×)	12.26s (2.0×)

The column *Calculations* contains the complete times in [s] spent for the GPU execution, including the time spent by CPU to copy the data form CPU to GPU memory, launch the execution of the GPU part of the program, and to copy the results back to the CPU memory.

The column *Complete* contains the times required for the complete execution of the module (CPU and GPU). We can see that for *r.hata* this time is much larger from the GPU computation time, limiting the overall acceleration to only 2x, which makes GPU acceleration rather pointless. The majority of this time is spent on reading and writing of large GRASS raster maps (DEM, path loss map, ca. 126M points) using GRASS library calls, which is a rather slow process. The computation itself is simple and performed very quickly by GPU, hence the time required for reading and writing of the maps dominates.

The difference in the GPU execution performance with single and double floating point precision is much smaller than one would expect from the raw difference in the speed of single and double precision floating point operations (8x for our consumer-grade graphic adapter). This is so because only some of the GPU program instructions actually perform floating point computations.

The results from Table 2 are graphically represented in Fig. 6.

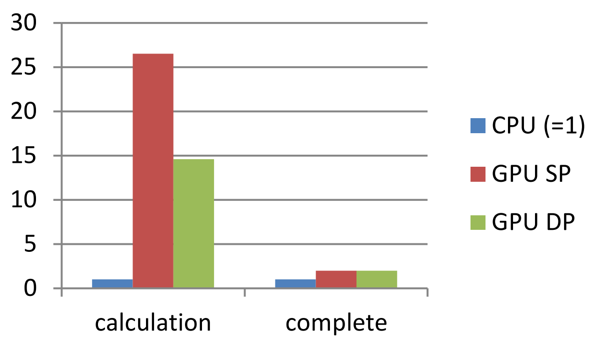

Fig. 6. Relative *r.hata* execution speed for Slovenia region

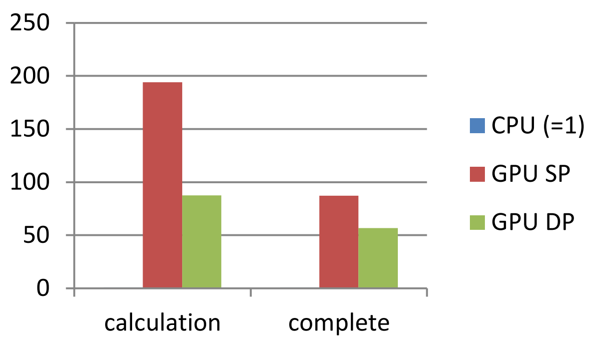

Fig. 7. Relative *r.hataDEM* execution speed for Ljubljana region

Execution performance of *r.hata* for the Ljubljana region is presented in Table 3, this time only for CPU and double precision GPU execution. We can see that the acceleration is the same and the execution times are proportional to the number of points (ca. 21M in this case) as expected.

Table 3. *r.hata* for Ljubljana region

	Calculations	Complete
CPU (DP)	2.026	3.828
GPU DP	0.140 (14.5×)	2.027 (1.9×)

4.5.2 r.hataDEM

As we have already described, *r.hataDEM* is computationally much more complex and – due to its obstacle-searching procedure – lacks the very regular structure of the *r.hata* algorithms, which would be a very desirable property for an efficient GPU implementation. Nevertheless, GPU was obviously able to cope well with it. Our rather straightforward implementation of the original CPU code on GPU resulted in a very efficient solution with an exceptionally large acceleration (Table 4), reaching almost 200x for single precision computation. Even together with reading and writing of the GRASS maps (DEM, clutter map, path loss map), the acceleration was still almost 90x for the single precision computation, justifying the GPU implementation of this module.

Table 4. *r.hataDEM* for Slovenia region

	Calculations	Complete
CPU (DP)	508.43s	511.62s
GPU SP	2.62s (194.0×)	5.86s (87.3×)
GPU DP	5.81s (87.5×)	9.05s (56.6×)

The results from Table 4 are graphically represented in Fig. 7.

5 Conclusion

We implemented GPU processing for two GRASS RaPlaT modules, *r.hata* and *r.hataDEM*, and tested the achieved acceleration. While the overall acceleration for the first module was very low due to its low computational complexity compared to the file reading/writing overhead, the acceleration of the second module gave very good results.

A few conditions must be fulfilled for an efficient implementation of GPU processing. Firstly, the number of parallel computations must be large enough to fully utilize the GPU processor. Generally, this number must be orders of magnitude larger than the number of GPU cores, so that the GPU internal scheduler can always switch execution to those threads with available data instead of waiting for the data to be transferred from/to the main GPU memory (which is a fast process but with rather large latencies). In our case – processing of large raster maps with around 126M and 21M points – this was not a problem.

The GPU computation part must be complex enough to make the time spent for overhead tasks performed by CPU negligible. In our case, the most time consuming task was reading and writing of the GRASS raster maps. Due to this, the *r.hata* module, with its relatively simple GPU-accelerated algorithm, achieved a very weak overall acceleration of only 2x. On the other hand, the *r.hataDEM* module, with its much more complex computations, achieved an exceptional 87x acceleration (for single precision computations), fully justifying its GPU implementation.

Generally, algorithms suitable for an efficient GPU acceleration should be as regular as possible, without much cross connections between parallel threads, and using input/output local data from data blocks that can be kept inside the fast on-chip memory cache. Transferring data to/from the GPU main memory is a fast process but with considerable latencies, which can slow down the processing unless other waiting threads have data available and can immediately continue their execution. It turned out that for our not-so-regular *r.hataDEM* module, the GPU processor was able to cope well with it and achieved high acceleration.

References:

[1] GRASS-RaPlaT web page, http://www-e6.ijs.si/RaPlaT/GRASS-RaPlaT_main_page.htm

[2] I. Ozimek, A. Hrovat, A. Vilhar, T. Javornik, GRASS-RaPlaT Radio Planning Tool for GRASS, User Manual V1.0, IJS DP-11386, 2013.

[3] A. Hrovat, I. Ozimek, A. Vilhar, T. Celcer, I. Saje, T. Javornik, Radio coverage calculations of terrestrial wireless networks using an open-source GRASS system. *WSEAS transactions on communications*, ISSN 1109-2742, 2010, vol. 9, no. 10, pp. 646-657.

[4] A. Hrovat, A. Vilhar, I. Ozimek, T. Javornik, E. Kocan, GRASS-RaPlaT Radio Planning Tool for GRASS GIS system, *Proceedings of 21st International Conference on Applied Electromagnetics and Communications (ICECom 2013)*, 14-16 October 2013, Dubrovnik, Croatia.

[5] GRASS Development Team, 2012. Geographic Resources Analysis Support System (GRASS) Software. Open Source Geospatial Foundation Project. http://grass.osgeo.org

[6] M. Neteler, M. H. Bowman, M. Landa, M. Metz, GRASS GIS: A multi-purpose open source GIS, *Environmental Modelling & Software*, Vol 31, 2012, pp. 124–130.

[7] NVIDIA CUDA, Parallel Programming and Computing Platform, http://www.nvidia.com/object/cuda_home_new.html

[8] OpenCL, The open standard for parallel programming of heterogeneous systems, https://www.khronos.org/opencl/

[9] OpenACC, Directives for Accelerators, http://www.openacc-standard.org/

[10] M. Hata, Empirical formula for propagation loss in Land Mobile radio services, *IEEE Transactions on Vehicular Technology*, Vol. 29, no. 3, august 1980.

[11] A. McNamara, C. W. I. Pistorious in J. A. G. Maherbe, Introduction to the uniform geometric theory of diffraction. Norwood: MA: Artech House, 1990.

[12] S. R. Saunders, Antennas and propagation for wireless communication systems: John Wiley & Sons, 1999.