GPU computation acceleration of GRASS GIS modules for predicting radio-propagation

Igor Ozimek, Andrej Hrovat, Andrej Vilhar, Tomaž Javornik

***Abstract*—**GPGPU (General-Purpose Computing on Graphics Processing Units) is a parallel computation technique that has become very popular with the advent of high performance and relatively low priced programmable GPU (Graphics Processing Unit) adapters and of the software tools required for general computing (compilers for computing languages such as OpenCL, etc.). GPU computing can offer massive computation acceleration for algorithms that fulfill certain requirements and map well to the GPU architecture. GPUs were originally developed for displaying and processing raster images, and so can be applied efficiently for fast processing of the rasterized geographic and other maps used in geographic information systems such as GRASS GIS (Geographic Resources Analysis Support System / Geographic Information System). GRASS-RaPlaT (Radio Planning Tool for GRASS) is a GRASS add-on for simulating radio signal propagation in actual geographic environments. In the case of large areas and relatively high resolution, simulations can become computationally demanding, taking a considerable amount of time to accomplish. GPU parallelization of radio propagation modules is therefore presented and the results analyzed, together with the conditions that must be fulfilled to employ GPU computation successfully and achieve considerable computation speedup.

***Keywords*—**Radio propagation simulation, GRASS, RaPlaT, parallel processing, GPU computing.

I. Introduction

RADIO propagation planning is important for setting up a transmitter or for building a network of transmitters and predicting the signal strength at various locations during the normal transmitter/network operation. Numerous professional tools exist for this purpose, their common drawback being high price and inability of the user to add his/her own propagation models in a simple way. Open source solutions have been developed to meet these problems. Usually, their functionalities are limited compared to those of 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 (Radio Planning Tool for GRASS, RaPlaT for short) [1]-[4].

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. For example, a map covering 100 km×100 km with a resolution of 25 m would require 16M raster points; improving resolution to 5 m would require 400M points. The parallel nature of the raster-based algorithm calls for a massive parallelization approach, such as GPGPU (General-Purpose Computing on Graphics Processing Units), that can result in multiple times speedup of the whole computing process.

In the next section, GRASS-RaPlaT is briefly presented. Section 3 gives a short description of possible approaches to parallelization, with special emphasis on the GPU computing. Section 4 presents parallelization of two GRASS-RaPlaT radio signal propagation modules using GPGPU, analyzes the results, and determines the conditions that must be fulfilled for successful employment of GPU processing.

I. Ozimek, A. Hrovat, A. Vilhar, and T. Javornik are with the Department of Communication Systems, Jozef Stefan Institute, Ljubljana, Slovenia (phone: +386-1-477-3105; fax: +386-1-477-3111; e-mail: igor.ozimek@ijs.si).

II. GRASS-RaPlaT

RaPlaT 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 additional user-written modules to be added in a simple way.

RaPlaT consist of a number of GRASS-compatible modules. Fig. 1 shows the RaPlaT functional block diagram with modules presented as white rectangles and input/output files as colored rhomboids. RaPlaT can optionally save the results into a standard data base such as MySQL or PostgreSQL.

RaPlaT core modules perform three distinct tasks that are needed for computation of radio coverage (Fig. 2). 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:

- The free space model.
- The 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.
- The Walfisch-Ikegami model (developed in the

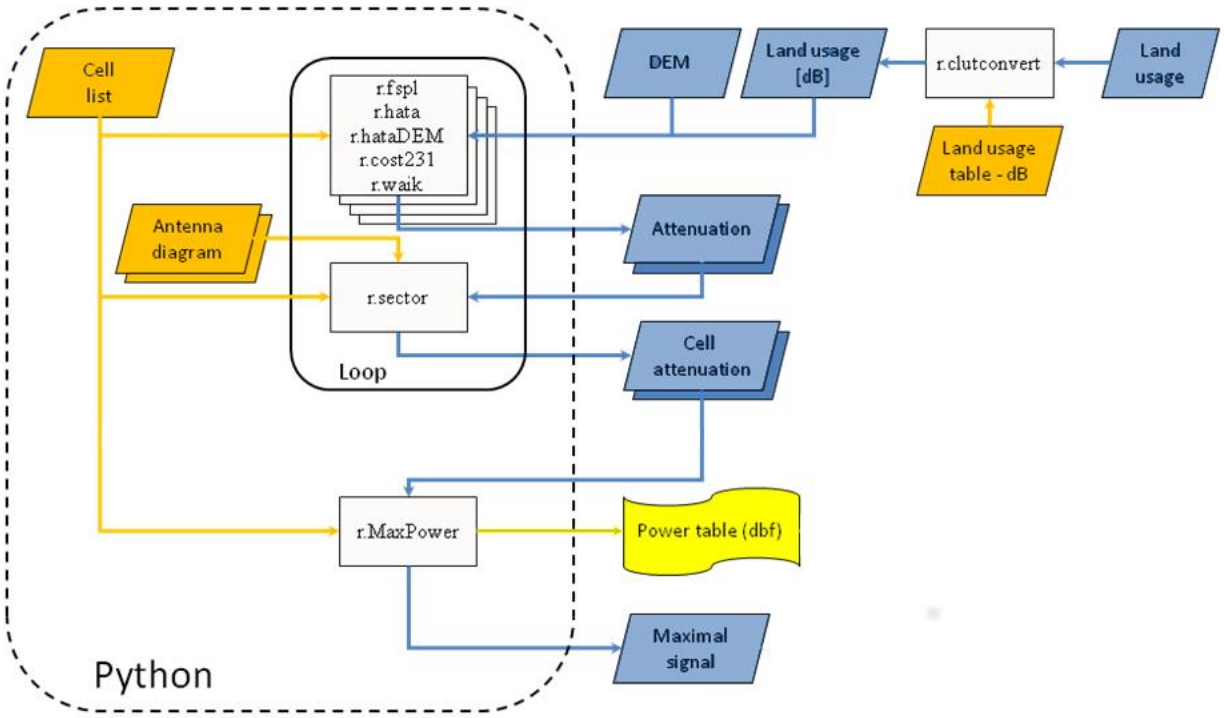

Fig. 1. GRASS-RaPlaT block diagram

framework of the COST231 project).

- The Longley/Rice model (ITM – Irregular Terrain Model).
- The ITU-R Recommendation P.1546-4 model.
- The Urban model.

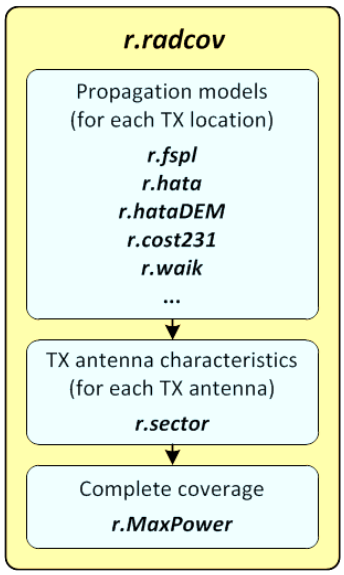

Fig. 2 RaPlaT core modules

An example of a radio signal fading map computed using the simple Okumura-Hata model (the *r.hata* module), together with the underlying geographic map, is shown in Fig. 3.

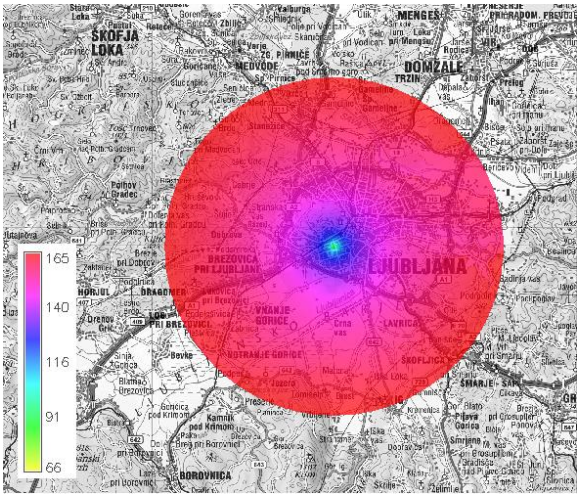

Fig. 3. Isotropic radio signal fading map [dB], Okumura-Hata model (*r.hata*)

The next task, performed by the *r.sector* module, takes into account the actual transmit antenna radiation pattern and modifies the previously computed isotropic path loss map accordingly. The antenna characteristics are specified in an MSI-format file. This is a simple text file defining various properties of an antenna, such as its name, manufacturer, radio frequency, gain, polarisation, electrical tilt and, most importantly, its radiation pattern in the horizontal and vertical planes with one degree resolution. From these data, the 3-D radiation pattern can be synthesized. An example of a fading

map produced by *r.sector* is shown in Fig. 4.

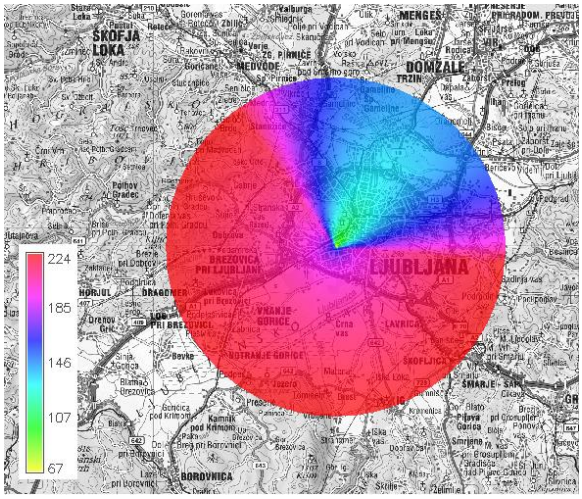

Fig. 4. Directed radio signal fading map [dB], (*r.sector*)

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. Fig. 5 shows an example of the radio signal strength map produced by *r.MaxPower* for the case of three transmitters (antennas) on the same location, corresponding to an actual GSM base station in Ljubljana. The radio strength is given in dBm for a receiver with 0 dB antenna gain.

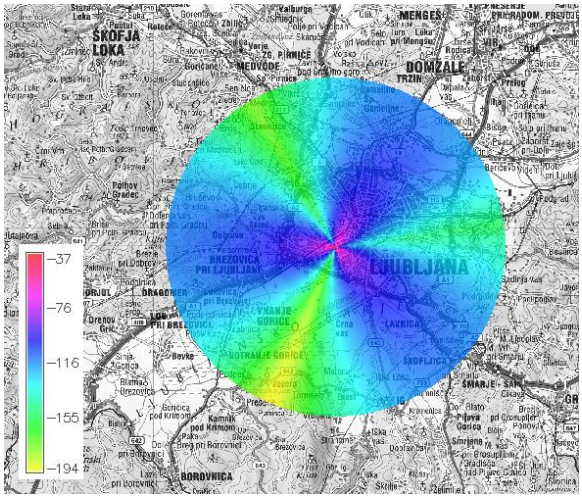

Fig. 5. Received signal strength [dBm] for three transmitters on one location (*r.MaxPower*)

The *r.MaxPower* module can compute a number of results other than the received signal strength. For LTE communication systems, it can compute maps for RSSI (Received Signal Strength Indicator), RSRP (Reference Signal Received Power), RSRQ (Reference Signal Received Quality), CINR (Carrier to Interference + Noise Ratio), maximum spectral efficiency and for maximum throughput and interference.

A RaPlaT user does not need to call all these modules individually. Only the *r.radcov* module (written in Python), which reads a CSV-format file with the description of the transmitter network configuration and calls the previously described modules as necessary, need be called. Each module instance is executed as a sequential single-thread process; however, *r.radcov* can call them concurrently for parallel execution on a multi-core processor.

Besides the classical radio propagation models listed above, an experimental ray-tracing module has also been implemented in RaPlaT [7], [8].

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.

III. Parallel Computing and GPU

The computational speed of a conventional SISD (Single Instruction, Single Data) processor is limited by its clock frequency, the 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. This can be minimized by employing a fast on-chip cache memory.

For many years, SISD processor capabilities have been increased by increasing the clock frequency and the execution unit capabilities, employing techniques such as instruction-level parallelism (instruction pipelining, 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 limit around 2003, when increasing the clock frequency slowed down (largely governed by the processor thermal power dissipation/cooling limitations), and the execution speed could not be improved much more by enlarging the CPU complexity. However, the maximum number of transistors per chip still keeps rising – they are employed to build higher level parallel structures i.e. 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, the latter 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. Multi-core CPU architecture provides high level parallelism of the MIMD (Multiple Instruction, Multiple Data) type, which can concurrently execute multiple independent processes or relatively high level threads of a single process. SIMD extensions, on the other hand, are low level parallel structures that are most suitable for completely identical operations on vector/matrix data. Current implementations usually support only small vector sizes, e.g. four floating point values, suitable for fast media content processing.

In the previous section we have already noted that GRASS-RaPlaT, in its basic form, already supports parallel execution. In the case of a transmitter network consisting of a number of transmitters (antennas), computation of the radio signal fading for each of them (the propagation model and *r.sector* modules) can be executed in parallel on a multi-core processor by *r.radcov*. The speedup achievable is limited by the rather small number of available CPU cores. This approach can be extended by running computations on a computer cluster, instead of on a single computer, as shown in [9]. This can be achieved by modification of GRASS-RaPlaT, but the main drawback is the high cost of the computing cluster hardware compared with that of the cost-effective GPU approach described in the sequel. A further advantage of the GPU solution is that it accelerates the execution of each individual module and can thus be used for a single transmitter, while the multi-core/cluster parallel computing is only applicable to a transmitter network consisting of a number of transmitters.

A. 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 large market for computer games, which enabled their relatively low prices. As 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 based primarily on the C language, although some other languages are often supported. NVIDIA is recognized as the pioneer in the GPGPU field with its CUDA (Compute Unified Device Architecture) parallel computing platform, programming model and software tools (compiler, etc.) [10]. A drawback of CUDA, however, is that it only supports NVIDIA's GPUs. An independent effort by the Khronos Group industry consortium (including NVIDIA among many others) resulted in OpenCL (Open Computing Language) [11], an open standard for parallel programming of heterogeneous systems.

Both CUDA and OpenCL use a similar programming model, called SIMT (Single Instruction, Multiple Threads) by NVIDIA, in which multiple independent threads are executed concurrently, using a single instruction on multiple data. However, this differs from vector processor SIMD and enables conditional branching (*if* statement in C), by which, for each branch, only the corresponding subset of data is processed and the remainder wait for their turn (in an alternative branch or after the branches merge again). Those 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.

The relative efficiency of the programs using CUDA or OpenCL has been the topic of many papers [12-15]. While they differ at some points, often in favor of CUDA – which is not strange given its longer history and support for only NVIDIA GPUs, these differences are, on average, not substantial and the independence of the manufacturer of OpenCL is certainly an advantage.

There is another approach to GPU programming, OpenACC (Open Accelerators) [16]. Here, the procedures for GPU are not coded as separate C-modules. Instead, the program is written completely in the normal C language (or some other supported language). The OpenACC compiler is instructed as to 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 manufacturers of hardware platforms (i.e. GPU processors) is rather limited, with AMD (formerly ATI) being the only other manufacturer of devices with capabilities comparable to those of NVIDIA.

IV. Parallelization, Results and Analysis

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

It must be noted that the existing GRASS-RaPlaT support for parallel execution on multiple-core processors cannot be combined with GPU execution, since this would cause multiple parallel processes to compete for the same GPU resources. It would only make sense if multiple GPU processors/adapters were installed, and this is not yet supported in the current version of the GPU-accelerated RaPlaT modules.

A. Development tools

At first sight, the use of OpenACC is tempting, since it enables GPU execution of C programs just by 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 correctly and efficiently. We therefore took the other approach, and chose OpenCL, on the basis of its hardware manufacturer's independence.

B. Hardware

High performance GPU adapters are produced by NVIDIA and AMD. NVIDIA was chosen, on the grounds of 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, the latter including the Tesla adapter family specifically designed for GPU computing. Professional and consumer-grade cards generally share the same GPU microarchitecture but with different computing capabilities, especially regarding the double precision floating point computation speed.

The GPU implementation and testing presented here were performed using two GPU adapters. The first, GeForce GTX 580, is a high performance consumer-grade adapter. Some basic properties of this adapter are summarized in Table I.

Table I. Some basic properties of GeForce GTX 580

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)	1624.064 / 203.008

The main reason for choosing it was its much lower price than those of professional adapters while having capabilities comparable to those of the much more expensive professional-grade products (with the exception of its 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 but at 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 than those of the single precision operations. Single precision floating point performance is calculated according to (1).

$$GFLOPS = 2 \times f_{SHADER} \times N_{CORES} \quad (1)$$

where f_{SHADER} is the shader clock frequency, and N_{CORES} is the number of cores.

In the meantime, NVIDIA has developed a new, improved GPU microarchitecture named Kepler, the main emphasis being on reduced power consumption and better games support. The 64-bit floating point units are no longer integrated into the computing cores but are separate entities available in appropriate quantities. For consumer-grade adapters, their number is relatively small compared to the number of cores, resulting in double precision floating point computations being typically 24x slower than the speed of single precision ones. Thus, if a consumer-grade graphic adapter is used for general computing, a Fermi-based adapter could still be the better choice.

For comparison, testing was performed with a very basic and low-cost GPU adapter GeForce 210. It uses the Tesla microarchitecture (not to be confused with the Tesla adapter family), which is the predecessor of the Fermi microarchitecture and supports only single precision floating point operations. Some basic properties of this adapter are summarized in Table II.

Table II. Some basic properties of GeForce 210

No. of cores	16
GPU / Shader clock frequency	520 MHz / 1230 MHz
GPU adapter memory size	512 MB
GPU memory bandwidth	4.8 GB/s (32-bit, 1200 MHz)
GPU adapter bus	PCI-E 2.0 x 4 (max. 2 GB/s)
GFLOPS (SP)	39.360

Its floating point performance is calculated according to (1). In terms of single precision GFLOPS, this adapter is 41.3 times slower than GTX 580.

NVIDIA CUDA / OpenCL compilers translate source code to the PTX (Parallel Thread Execution) pseudo-assembly language, which is common to all NVIDIA GPU microarchitectures and computing capabilities, and is forward compatible. PTX is translated at run time into the binary code for a particular GPU processor by the compiler included in the NVIDIA graphic adapter driver. For repetitive running of a program, the binary code is kept in the memory and reused. The process of PTX translation takes time, which can be avoided by precompiling the binary code, but this makes the program runnable only on a specific GPU microarchitecture. In our case, the translation time was less than one second and was excluded from the execution time measurements.

C. OS environment

NVIDIA supports MS Windows, Linux and Mac OS X operating systems. GRASS 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 of keeping the graphic card responsive for its supposedly primary function – displaying the picture on the monitor. Unfortunately, in MS Windows, this limitation is active even if no monitor is connected to the adapter (although this can be changed by tweaking the registry). The much more expensive adapters from the Tesla family use another version of driver without this limitation. In Linux (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. (However, remote work is possible with VNC, which actually runs a local session on the host machine, and transfers the screen image to the remote client.)

D. Parallelized RaPlaT modules

We have modified two RaPlaT propagation models 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.

1) r.hata

The Okumura-Hata model [17] is a rather simple model that ignores 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 level in metres, like the one shown in Fig. 6.

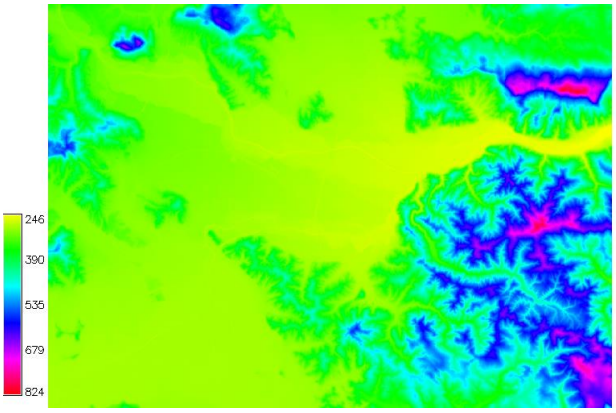

Fig. 6. DEM for the region around the city of Ljubljana

The computed path loss fading map (in dB) is relatively simple, and hence not very accurate for complex environments (Fig. 7).

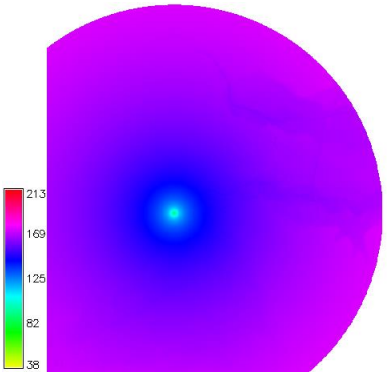

Fig. 7. Path loss map computed by *r.hata*

Some computations are carried out 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 computation defined by the following equation:

$$L_U = L_{Uconst} - 13.82 \cdot \log(h) + \\ + (44.9 - 6.55 \cdot \log(h)) \cdot \log(d) \quad (2)$$

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

According to (2), two base-10 logarithms (log(h) need only be computed once), three multiplications and three additions/subtractions must be computed. The computations have a very regular structure with the calculation for each raster point being performed independently of those for other points. As such, it should be ideal for implementation on a GPU processor. However, these computations are rather simple and can be performed quickly, so that the GPU processing overhead (fetching and storing the data from/to the main graphic adapter memory, etc.) could constitute a bottleneck. Therefore, a sufficient number of raster points must be processed in parallel at every moment, so that GPU can schedule their execution appropriately – computing some points while others are waiting for data to be fetched/stored. Our tests on GTX 580 (with 512 computing cores) have shown that a few 10,000s of computation threads need to be launched in parallel to achieve efficient GPU utilization.

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

2) r.hataDEM

The *r.hataDEM* module implements an extended Okumura-Hata model. For LOS receive locations, it uses the basic Okumura-Hata model. For NLOS receive locations, it computes the path loss according to the edge diffraction effect on the highest obstacle in a direct line between the transmitter and the receive point [18, 19]. Additionally, it takes into account the effects of land-usage (buildings, forests, etc.) at the receive locations. This is given, in dB, by an additional path loss raster map called a clutter map, such as that in Fig. 8.

Compared to the *r.hata* path loss map (Fig. 7), the resulting map, in dB, is much more detailed and accurate, as seen in Fig. 9.

For LOS locations, the computation is basically the same as that for *r.hata* and is described by (2). For NLOS locations, the knife edge diffraction loss is calculated according to (3) and (4):

$$L_{ke} = -20 \cdot \log \frac{1}{\pi \cdot \nu \cdot \sqrt{2}} \quad (3)$$

$$\nu = h \cdot \sqrt{\frac{2 \cdot (d_1 + d_2)}{\lambda \cdot d_1 \cdot d_2}} \quad (4)$$

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, d_1 and d_2 are the distances of the mobile and base stations from the obstacle, and λ is the radio signal wavelength.

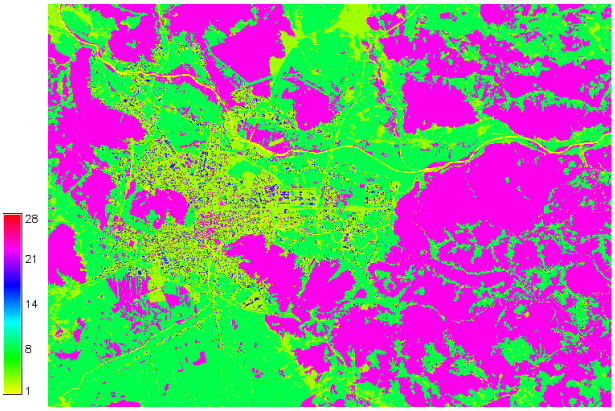

Fig. 8. Clutter map for the region around the city of Ljubljana

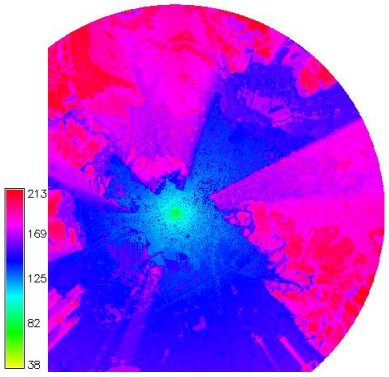

Fig. 9. Path loss map computed by *r.hataDEM*

What is particularly important for computational complexity is the fact that, for each point, a LOS/NLOS check that seeks the highest obstacle along the direct line between the transmitter and the receive point has first to be made. For each receive point, the obstacle searching algorithm performs some initial computations in addition to those described by (3) and (4), but the important computationally demanding part is that of traveling along the transmitter-receiver line, looking for obstacles. Although only some basic computations are performed in this loop (additions, subtractions, multiplications), it increases the order of complexity by 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.

E. Execution performance

The execution performance of the GPU-accelerated version of the *r.hata* and *r.hataDEM* modules and their original CPU-only versions was compared for two geographic regions, the first of which was the whole Slovenia region. The corresponding DEM covered an area of 352 km×224 km with a resolution of 25 m and a 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 smaller, the region around the city of Ljubljana, but with a greater resolution. The corresponding DEM covered an area of 26.9 km×19.2 km with a resolution of 5 m, with a resulting raster size of 5381×3841 = 20,668,421 points. It was used for both *r.hata* and *r.hataDEM* modules.

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

1) r.hata

The performance of the CPU execution (with the usual double floating point precision) and of the GPU execution with single and double floating point precision for the Slovenia region is presented in Tables III and IV. The numerical results computed with single and double precision were not completely identical but the differences were negligible for our use. Table III shows the execution times when only one raster row at a time (14081 points) is computed, while Table IV shows the times when many raster rows at a time are sent from CPU to GPU for computing – all 8961 rows for GeForce GTX 580, 1000 rows for GeForce 210.

The column ‘Calculations’ contains the execution time for the computationally intensive part of the module. In the case of GPU acceleration, this time comprises the time spent by CPU to copy the data from CPU to GPU memory, launch the execution of the GPU part of the program and copy the results back to the CPU memory. Each of these processes is sent to the GPU task queue as a separate task. The waiting (latency) times caused by the GPU task mechanism are included in the GPU execution times.

The column ‘Complete’ contains the times required for complete execution of the module (CPU and GPU). For *r.hata* this time is much larger than the GPU computation time, limiting the overall acceleration to only 2x, which makes GPU acceleration a rather pointless activity. The majority of this time is spent on reading and writing 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 the maps dominates.

Table III. Execution time for *r.hata*, Slovenia region, one row at a time

	Calculations	Complete
CPU (DP)	12.54s	24.00s
GTX 580 DP	2.56s (4.9×)	14.30s (1.7×)
GTX 580 SP	2.08s (6.0×)	13.81s (1.7×)
210 SP	5.75s (2.2×)	17.31s (1.4x)

Table IV. Execution time for *r.hata*, Slovenia region, many rows at a time

	Calculations	Complete
CPU (DP)	12.54s	24.00s
GTX 580 DP	0.86s (14.6×)	12.26s (2.0×)
GTX 580 SP	0.47s (26.5×)	11.92s (2.0×)
210 SP	3.53s (3.6×)	14.88s (1.6×)

The differences in GPU execution times between Tables III and IV are substantial and are due mostly to the latency times of the GPU task queue processing. This can be seen from Tables V and VI, which show only net data copy and GPU execution times without the time lost by GPU task queue processing.

Table V. Net GPU time for *r.hata*, Slovenia region, one row at a time

	Data copy	GPU execution
GTX 580 DP	0.20s	0.60s
GTX 580 SP	0.20s	0.14s
210 SP	1.00s	2.80s

Table VI. Net GPU time for *r.hata*, Slovenia region, many rows at a time

	Data copy	GPU execution
GTX 580 DP	0.20s	0.48s
GTX 580 SP	0.20s	0.083s
210 SP	0.77s	2.61s

GPU execution times in Tables V and VI for GeForce 210 computations and for double precision GTX 580 computations are similar. This is not the case with GTX 580 single precision computations, which means that one row of data (14081 points – computation threads) is not sufficient for efficient utilization of its 512 cores in the case of the relatively simple Okumura-Hata algorithm. It can also be seen that, for a large number of computation threads (Table VI), the ratio of the execution performances of the two adapters (2.61 / 0.083 = 31) is not greatly different from that of their theoretical performances (41.3).

The difference in GPU execution performance between single and double floating point precision (5.8-fold for a large number of computation threads, Table VI) is smaller than the raw difference in the speed of single and double precision floating point operations (8-fold for our consumer-grade graphic adapter). This is so because not all the GPU program instructions actually perform floating point computations.

The results from Tables III and IV are shown in Figs. 10 and 11.

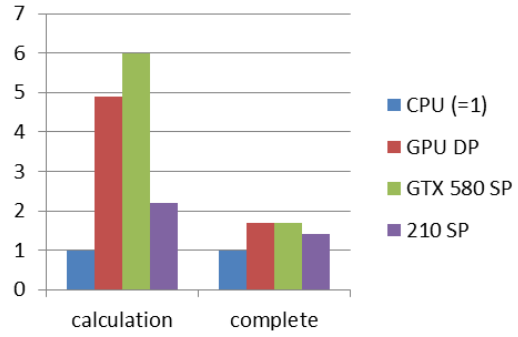

Fig. 10. Relative *r.hata* execution speedup, Slovenia region, one row at a time

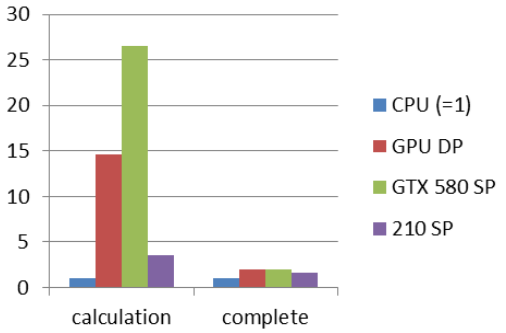

Fig. 11. Relative *r.hata* execution speedup, Slovenia region, many rows at a time

The execution performance of *r.hata* for the Ljubljana region is presented for CPU and double precision GPU execution only in Tables VII and VIII, for single and for all rows at a time, respectively. For computation of many rows at a time (Table VIII), the acceleration is approximately the same as that for the Slovenia region (Table IV) and the execution times are proportional to the number of points (*ca.* 21M in this case), as expected.

Table VII. *r.hata* for Ljubljana region, one row at a time

	Calculations	Complete
CPU (DP)	2.03s	3.83s
GTX 580 DP	0.80s (2.5×)	2.82s (1.4×)

Table VIII. *r.hata* for Ljubljana region, all rows at a time

	Calculations	Complete
CPU (DP)	2.03s	3.83s
GTX 580 DP	0.14s (14.5×)	2.03s (1.9×)

2) r.hataDEM

As 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 that would be very desirable for efficient GPU implementation. Nevertheless, GPU was clearly able to cope well with its partly irregular structure. Our relatively straightforward implementation of the original CPU code on GPU resulted in a highly efficient implementation with exceptionally large acceleration (Table IX), reaching 194x for single precision computation. Even taking into account the reading and writing of the GRASS maps (DEM, clutter map, path loss map), the acceleration was still 87x for the single precision computation, justifying the use of GPU implementation of this module.

Table IX. *r.hataDEM* for Ljubljana region (all rows at one time)

	Calculations	Complete
CPU (DP)	508.43s	511.62s
GTX 580 DP	5.81s (87.5×)	9.05s (56.6×)
GTX 580 SP	2.62s (194.0×)	5.86s (87.3×)
210 (SP)	110.27s (4.6×)	113.51 (4.5×)

The results from Table IX are graphically represented in Fig. 12.

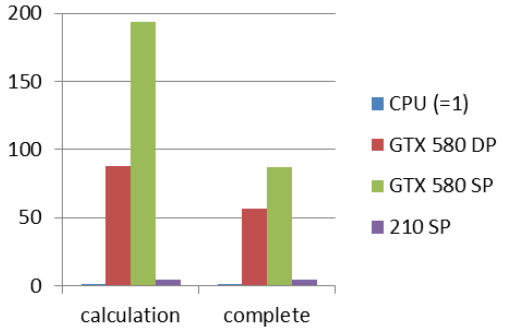

Fig. 12. Relative *r.hataDEM* execution speedup, Ljubljana region (all rows at one time)

V. Conclusion

GPU processing for two GRASS RaPlaT modules, *r.hata* and *r.hataDEM* has been implemented and their achieved accelerations compared. While the overall acceleration for the former was very much lower, due to its lower computational complexity than that of the file reading/writing overhead, the acceleration of the second module was very satisfactory.

Certain conditions must be fulfilled for efficient implementation of GPU processing. First, the number of parallel computations must be large enough to utilize fully the GPU processor. In general, this number must be orders of magnitude greater 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 the present case, in which large raster maps with around 126M and 21M points were processed, this was not a problem.

The GPU computation part must be sufficiently complex to make negligible the time spent for overhead tasks performed by CPU. In the present case, the most time consuming task was reading and writing the GRASS raster maps. For this reason, the *r.hata* module, with its relatively simple GPU-accelerated algorithm, achieved a very weak overall acceleration of only 2-fold. On the other hand, the *r.hataDEM* module, with its much more complex computations, achieved an exceptional 194x acceleration (for single precision computations), fully justifying its GPU implementation.

In general, algorithms suitable for effective GPU acceleration should be as regular as possible, i.e. 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 registers and memory cache. Transferring data to/from the GPU main memory is a rapid process, but with considerable latencies that can slow down the processing unless other waiting threads have data available and can immediately continue their execution. The present *r.hataDEM* module is not ideal in this respect. Nevertheless, the GPU processor was able to cope well with its irregularities and achieved very high acceleration.

References

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

[2] A. Hrovat, I. Ozimek, A. Vilhar, T. Celcer, I. Saje, T. Javornik, "An open-source radio coverage prediction tool", *Latest Trends on Communications*, *14th WSEAS International Conference on Communications*, Corfu Island, Greece, 23-25 July 2010, pp. 135-140.

[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*, Vol. 9, No. 10, 2010, 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] A. Vilhar, A. Hrovat, I. Ozimek, T. Javornik, "Efficient open-source ray-tracing methods for rural environment", *Recent Researches in Communications and Computers*, *16th WSEAS International Conference on Computers*, Kos Island, Greece, 14-17 July 2012, pp. 51-56.

[8] A. Vilhar, A. Hrovat, I. Ozimek, T. Javornik, "Shooting and bouncing ray approach for 4G radio network planning", *International Journal of Communication*, Vol. 6, No. 4, 2012, pp. 166-174, http://www.naun.org/multimedia/NAUN/ communications/16-589.pdf

[9] L. Benedičič, F. A. Cruz, T. Hamada, P. Korošec, "A GRASS GIS parallel module for radio-propagation predictions", *International Journal of Geographical Information Science*, Vol. 28, No. 4, 2014, pp. 799-823.

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

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

[12] P. Du, R. Weber, P. Luszczek, S. Tomov, G. Peterson, J. Dongarra, "From CUDA to OpenCL: towards a performance-portable solution for multi-platform GPU programming", *Parallel Computing*, Vol. 38, No. 8, 2012, pp. 391-407.
[13] K. Karimi, N. G. Dickson, F. Hamze, "A performance comparison of CUDA and OpenCL", Cornell University Library, 16 May 2011, http://arxiv.org/abs/1005.2581
[14] J. Fang, A. L. Varbanescu, H. Sips, "A comprehensive performance comparison of CUDA and OpenCL", *2011 International Conference on Parallel Processing (ICPP)*, Taipei City, 13-16 Sept. 2011, p.p. 216-225.
[15] K. Komatsu, K. Sato, Y. Arai, K. Koyama, H. Takizawa, H. Kobayashi, "Evaluating performance and portability of OpenCL programs", *The Fifth International Workshop on Automatic Performance Tuning iWAPT*, 22nd June 2010, Berkeley, USA, http://vecpar.fe.up.pt/2010/workshops-iWAPT/Komatsu-Sato-Arai-Koyama-Takizawa-Kobayashi.pdf
[16] OpenACC, Directives for Accelerators, http://www.openacc-standard.org/
[17] M. Hata, "Empirical formula for propagation loss in land mobile radio services", *IEEE Transactions on Vehicular Technology*, Vol. 29, No. 3, August 1980.
[18] A. McNamara, C. W. I. Pistorious, J. A. G. Maherbe, *Introduction to the Uniform Geometrical Theory of Diffraction*, Artech House, 1990.
[19] S. R. Saunders, *Antennas and propagation for wireless communication systems*, John Wiley & Sons, 1999.

Igor Ozimek was born in Ljubljana, Slovenia, in 1957. He received the B.Sc., M.Sc., and Ph.D. degrees in Electrical Engineering from the University of Ljubljana, Slovenia, in 1980, 1988 and 1993, respectively.

He spent six months at the University of Westminster, London, UK, as a visiting scientist. He is currently a researcher in the Department of Communication Systems at the Jožef Stefan Institute. His research interests include digital communications, GPU processing, and computer networks.

Andrej Hrovat was born in Novo mesto, Slovenia, in 1979. He received the B.Sc. and M.Sc. degrees in Electrical Engineering from the University of Ljubljana, Slovenia, in 2004 and 2008, respectively. He obtained the Ph.D. degree in Electrical Engineering from the Jožef Stefan International Postgraduate School, Slovenia, in 2011.

He is currently a researcher in the Department of Communication Systems of the Jožef Stefan Institute and assistant at the Jožef Stefan International Postgraduate School. His research interests include radio signal propagation, channel modeling, terrestrial and satellite fixed and mobile wireless communications, radio signal measurements and emergency communications.

Andrej Vilhar was born in Ljubljana, Slovenia, in 1979. He received the B.Sc. and Ph.D. degree in Electrical Engineering from the University of Ljubljana, Slovenia, in 2004 and 2009, respectively.

He spent one year at the French aerospace lab ONERA, Toulouse, as a post-doc researcher. He is currently a researcher at the Department of Communication Systems of the Jožef Stefan Institute. His previous research interests include network modeling and mobility management in IPv6 networks. Currently, he is focusing on radio signal propagation and channel modeling, with the emphasis on terrestrial network planning and on the microwave satellite signal measurements.

Tomaž Javornik was born in Kočevje, Slovenia, in 1962. He received the B.Sc., M.Sc., and Ph.D. degree in Electrical Engineering from the University of Ljubljana, Slovenia, in 1987, 1990 and 1993, respectively.

He spent six months at the University of Westminster, London, UK, as a visiting scientist. He is currently a senior researcher in the Department of Communication Systems at the Jožef Stefan Institute and assistant professor at the Jožef Stefan International Postgraduate School. His research interests include radio signal propagation channel modeling in terrestrial and satellite communication systems, adaptive coding and modulation, adaptive antenna and MIMO systems, wireless optical communication, cooperative communication, adaptive coding and modulation, relay communication and self-organizing networks.