2017 International Conference on Emerging Trends & Innovation in ICT (ICEI)
Pune Institute of Computer Technology, Pune, India, Feb 3-5, 2017

Processing geospatial images using GPU

Animesh Sanjay Karnewar
Comp dept., Pune Institute of Computer Technology,
Pune, Maharashtra 411052 INDIA
E-mail: animeshsk3@gmail.com

***Abstract*—The art of computer programming has evolved symbiotically in the world of parallel processors. Thus, a computer program cannot be considered merely as a sequence of step by step instructions as it was earlier. Almost all the serial processors have now been replaced by their parallel counterparts**.

The present paper discussesapplication of CUDA, which is the most used platform for developing GPU softwares,for atypical GIS based system. To exemplify this, a terrestrial image was processed upon by a CUDA based system and the methodology has been presented with a minute consideration of majority of the nuances involved. The paper also attempts a juxtaposing comparison of GPU and CPU architecturesin accordance with the GIS.

Keywords— Geographic information systems (GIS), Parallel programming, Geospatial analysis.

I. INTRODUCTION

NVIDIA brought about a revolution in the field of processors with their GPU (Graphics Processing Unit), intended solely for computer graphics and gaming applications [1]. However, it was later realized that the processing and computational power of the GPUs' could be harnessed for other tasks as well. NVIDIA then came up with their series of GPGPUs' (General Purpose Graphics Processing Unit). These GPGPUs' have shown tremendous potential in fields such as Medicine, Mechanical modeling, Computational finance, Databases, Bioinformatics, AI deep learning and they are still entering in others [2].

GIS has become ubiquitous as it has amalgamated with the most common applications in day to daylife.It isnot limited to theuse of higher academicians and adroit practitioners anymore. The availability of open source data and APIs'(Application Programming Interfaces) has led to the genesis of many proliferant applications which are used even by the lay people. However, this doesn't allude that the elementary usage has beensmothered. The geospatial data processing organizations are flourishing more than ever before[3].

The gist of the modern geospatial computing lies in the use of programming languages, high speed architectures and organized data warehouses to place a complete control over the underlying data and the algorithm which designs its analysis [4,5,6,7]. The building of such automated processes finds its scope in performing customized task-specific operations. Parallel computing platforms such as CUDA leverage formidable adequacy to, multiple geospatial analysis tasks [8,9]. Thus, it becomes acutely crucial to the versed programmers to glean the underlying hardware knowledge of the GPGPUs' andadeptly program them for the functional tasks.

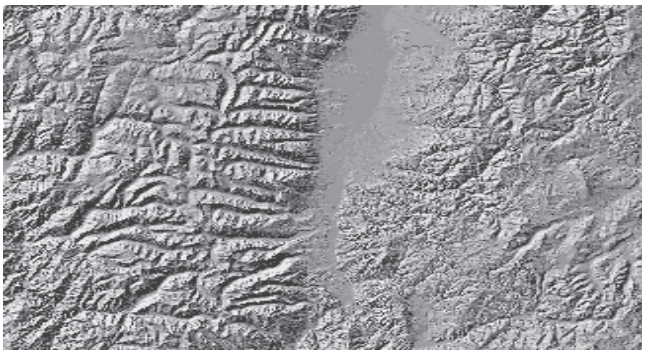

Figure 1: Elevation data from the Montana Natural Resources Information System (NRIS).

GIS includes many tasks; and forms a separate field of study. Some exemplary applicationsinclude,

1. Historical geographic information mapping
2. Road networking
3. Groundwater surveys and development
4. Hydrology
5. Town planning

'Visualization of the data' is a preliminary and commonly required chore in such applications and the best approach to achieve it is with the aid of imaging techniques. Thereby, comprising Image processing and its expertise to accomplish the primary objective.

The data acquired from assorted origins is merged together and mapped into a unit image. For instance, the elevation data can be mapped into 2Dand 3D imagesfor better visualization. But, the original data points being three-dimensional** in nature, it requires a distinct technique from the image processing arsenal; in this case the intensity of the color. The points at a lower elevation would have a low intensitycolor while the higher points display a higher intensity of the same, seeFigure 1.

However, this singular color approach does not provide an insight into the neighboring elevations, for example, slope of the land, proportionate heights of the terrain etc. To resolve this quandary, a better approach would be to synthesize3Dmodels of the data. This elaborate synthesisrequires utilizing an operation called 'Flow Directions Derivation'. This operation is an

exemplar of the data reductions that are derived for the image processing tasks in GIS systems.Thepresent paper discussionis based on the derivation of the flow directions from the given elevation map. The use of CUDA for this application is the initiationfor the creation ofingeniousandcompelling applications dedicated togeospatial modeling,such as Digital Globe, Petaquake, Manifold and BAE systems[10].

Prior to a thorough discussion of the flow matrix derivation using CUDA, the architectureofCUDA and the programming paradigms needed for the GPU are explored.

II. CUDA UNDERLYING ARCHITECTURE

The fundamental objectiveis to succinctly highlight the key distinctions between a CPU and a GPU. The current version of the classic CPU, see Figure2, can be visualized as an encapsulation of a small number of robust cores. The GPU, on the other hand, exemplifies the age old cliché "Unity is strength" by adhering together a very large number of simple cores. The CPU parallelism is also known as coarse grained parallelism while the GPU provides fine grained parallelism.

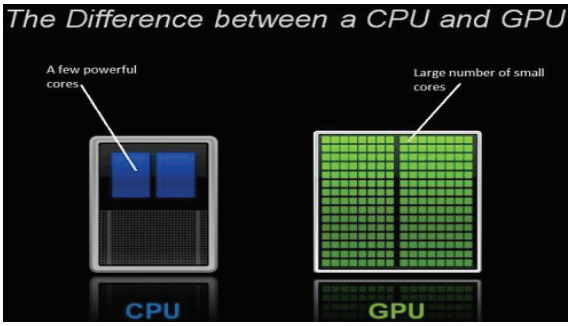

Figure 2: A high level view of CPU and GPU.

It has been observed that this point of distinction between the two processors is the only prevailing belief among numerous students and adept programmers[11]. However, the reality lies in the very fundamental models of instruction execution that these chips implement at the lowest level. The current multicore processors follow the MIMD model while the GPU has the SPMD model [12].

The CPU core fetches multiple instruction streams in its separate core pipelines allowing parallel execution of non-related instructions. This is the primary meansof achieving task based parallelism, i.e. executing disparate tasks inparallel. This model, however, has its performance constraints in relation to the number of memory fetches the core has to carry out for loading the instruction streams. In nutshell, the CPU is good at executing small number of complex independent tasks.

The GPU, conversely, has a distinctnovel architecture, Figure 3. It consists of a few SMs' (Streaming Multiprocessors) which are in turn made up of numerous SPs' (Streaming Processors).The SP is what is referred to as a CUDA core by the authentic GPU programmers. Due to having such a large number of cores on the chip, there is a longing forappropriate number of on-chip registers. The GPU incorporates the novel SPMD model of execution. This model differs from the traditional CPU execution model in two sizeable distinctions. Firstly, a group of NSPs' (Depends on the Warp size of the GPU used)executesthreads in a lock-step basis, i.e. running the same program but on different data.Secondly, because of the huge on-chip register file, switching between threads has practically null overhead.

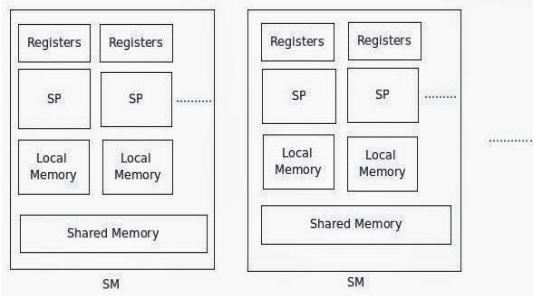

Figure 3: Block diagram of GPU.

Concisely, it can be asserted that a GPU isproficient at executing very large number ofcomparatively simple tasks.

Since, it is evident that a GPU is radically diverse from a conventional processing unit;a detailed review has been covered in this section.

A. GPU Memory model

The GPU is ascribed as 'device',whereas the CPU as 'host' in the parallel programming terminology. The device has a hierarchical memory structure alike the host, see Figure 4.Starting from the smallest unit, each SP has its own set of registers and some local memory depending on the particular architecture.

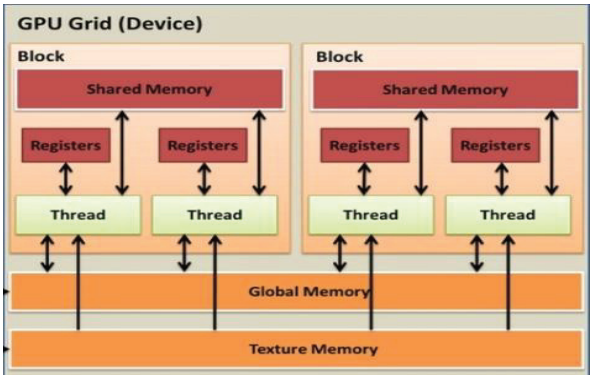

Figure 4: Memory model of GPU.

The local memory is attached to a single SP and acts like its own 'scratchpad memory'. Every SM has a shared memory which is common to all the SPs' within the same SM. The

device also supports an extensive store of memory termed as ‘global memory’. Concurrent fetches and writes to the global memory are comparatively slow and consume around 100s of clock cycles [1]. Another novel concept implemented here is that of the ‘constant’ and the ‘texture’ memory. These are not actual physical memories but views into the global memory that leverage interesting features. The constant memory holds the constant elements and is a read only view. The texture memory is a special view onto the global memory whichis useful when there is interpolation,suchas 2D or 3D rendering.

Furthermore, careful handling of memory is primarily important while programming for the GPU.Unlike a CPU, the caching of theGPU is not hardware implemented and doesn’t incorporate any cache refreshing algorithms. The programmer ought to take care of the memory transactions to achieve utmost performance. Otherwise, the performance of the program can fall as low as 10% of the actual performance [1].

B. GPU Programming model

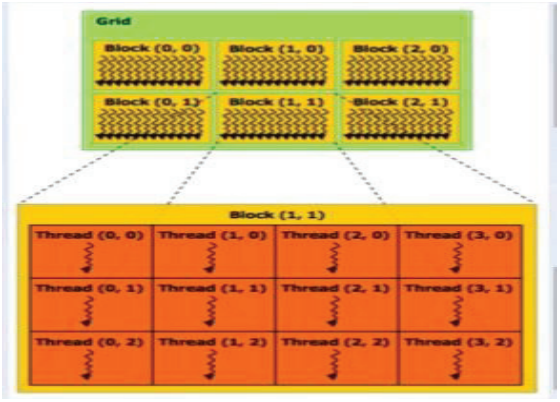

Figure 5: GPU Programming model.

The GPU, as it supports a finer level of parallelism, makes the programming effort for software to be legitimate in terms of leak-proof coding, proper utilization of the resources and achieving optimal performance out of the hardware chip. The crux of the art of programming for a GPU lies in the decomposition of task and realization of the parallel aspects of it. Typical forms of parallel problem decomposition methodologies includeData-based parallelism, Loop-based parallelism, Fork-join pattern, Divide and Conquer and the Tiling Grids apart from the task-based parallelismsupported by the conventional multi-coreCPU.

A thread is the smallest indivisible unit of programming on a device, Figure5. A single thread is what is assigned to an SP for execution. However, the threads don’t execute haphazardly in any order. They are grouped into blocks and a block is assigned to a single SM. The execution of all the threads in a single block is carried on using the same instruction set but on different data elements. This approach is quite akinto the vector processing instructions AVX, MMX and SSE except that the threads contain multiple instructions. A function that is to be executed on the GPU is known as a ‘kernel’in the CUDA terminology. A kernel is invoked on the device in the form of a grid, which consists of multiple blockseach containing the threads. A grid may contain a single dimensional or multi-dimensional arrangement of the blocks to be allocated. The highest dimension that CUDA supports is 3 i.e. one can generate a 3D arrangement of blocks within a grid. To perform this grid modeling, the CUDA API provides a built-in structure known as ‘dim3’. This allows the user to provide the maximum offset in each dimension for the blocks.

The threads allocated to a single block are not accommodated in a randommanner. They are arranged in ‘Warps’. Since, thewarp-size in not programmable, many of the coders perceive that warps are of infinitesimal importance while programming. On the contrary, the truthis that threads allocated keeping the warp-size in mind, perform much better than those ignoring the existence of warps. To achieve this, one doesn’t have to have the knowledge of the chip at hand. CUDA provides another built-in variable called ‘warpSize’. The reason why warps are of such importance is that internally it is not a complete block that is allocated to a single SM, but a number of warps that form a block are allocated for the execution. Typically, a warp contains either 32 or 48 threads. However, it is not true for all GPUs’. Therefore, it is best to use the built-in variable instead of hardcoding the warp-uselogic in the kernel.

Lastly, an upper hand in terms of performance of the GPU can also be attributed toincorporating the thread model for the task, in sync with the scheduling strategy implemented by the hardware [9].These are the axioms that ought to be followed while getting a GPU to work on parallel problems.

III. FLOW DIRECTIONS DERIVATION

This section discussesthe processingthat involves the derivation of flow directions from a given elevation map. The fundamental application of this operation is the synthesis of 3D models from the elevation data. The flow directions oblige as a relative spatial insight into the elevation data. This kind of derivation is an intermediate step required for further spatial analysis.

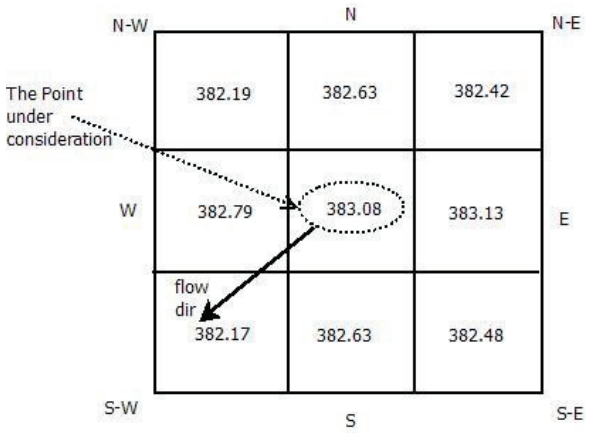

Figure 6: Flow directions derivation.

A meticulous description of the flow directions reveals that,as the data being discrete in nature, the available 360^0 directions are quantized into nine distinct directions for relativerepresentation. Thus, a linear flow is obtained that compresses original data that serves as metadata for the 3D model. The operation involves checking neighboringdata points, Figure 6, in a 3x3 matrix. One way to visualize this operation is to consider a drop of fluid placed at the current position of the derivation [13,14]. The value in that cell corresponds to the elevation data for that particular data point. Thus the flow of that drop takes place in accordance with three distinct conditions:

1. The cell under consideration is at a higher elevation than all other neighboring cells and the difference between them is same for all the points. This causes local proportionate maxima to get created and the flow will not take place. This condition is rather too unlikely to occur at such a macrolevel. However, it is only justifiable to account for it as it can cause vexing inconsistencies in the final 3D model if not considered.
2. The cell under consideration is at the same level as that of the neighboring points or is at a lower elevation compared to others. In suchcase, again no flow will take place.
3. In general, the elevations won't get arranged in such an ingenious manner as discussed in the above two cases but will have limited-range random values. In this case, the drop will flow to that cell which constitutes for the maximum positive difference of elevation.

So, there are in all 8 distinct directions to be stored and a dummy value to denote no flow. This can be achieved in multiple fashions. The simplest strategy here is to denote a direction with a simple unsigned short integer; for instance, 'N-W' is denoted by 1, 'N' by 2 and accordingly the rest. 'No flow-direction' can be denoted by 0. Thus all the data points in the end will have a number between 0-8 associated with them.

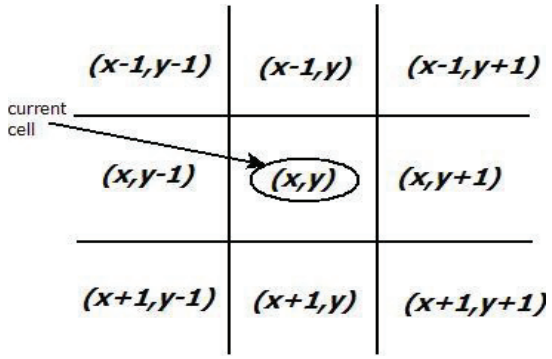

Figure 7: Indices of neighboring cells.

The algorithm for this application canbe implemented in many manners.However, it is to be noted that the code will run on a device instead of the host. One way could be a loop that iterates for every neighboring data cell checking for the elevation difference and assigning the one that constitutes for the maximum difference. But, it has been already verified that a GPU is better off doing a large number computations rather than if-else checking as the execution of the rest of the threads stalls because of branching to the other side of the condition [1]. Thus, minimum number of branching must be induced in the kernel logic. A comparatively lucid way to implement this is to calculate the differences and accumulate them in a [[difference array]]. The indices of the neighboring cellssee Figure 7;can be calculated from the indices of the current cell by simply accounting for the spatial arrangement of the cells. Then a single pass through the array could calculate the max-difference, along with its index into the array, and depending on the value of this max-difference, the flow direction is assigned. The final step in the algorithm however cannot bypass the '*if-else'* checking and thus has to be implemented in that way itself.

The last component of the design is the thread modeling and the parallel decomposition. The operation primarily puts the data at the zenith. Thus it is reasonable to split up the operation in a data-based parallel manner. On a multicore CPU, one would spawn number of threads equaling the number of cores for attune load balancing. Each thread would in turn process a chunk of data. Conversely, on a GPU, a small number of threads would acutely underwhelm it as it is customarily capable of handling thousands of threads. It would be equitable to assert that a disparate thread can be spawned for every data-point. Considering an HD image, which has resolution 1920 x 1080 pixels, as many as 2,073,600 threads would be spawned. This is still not a sprain for the GPU; but what could really be troublesome is creation of so many threads in a complete haphazard manner. Typically, the safest figure for optimal speed on any GPU is 256 threads per block(The number 256 has been derived after making a comparison of the performances of a series of NVIDIA GPUs' by executing different number of threads per block) [1]. Thus for an HD image, it would be 2,073,600/256=8100 blocks. The arrangement of the blocks too

is of prime importance. In spite of using the optimal number of threads per block, one can still jeopardize the performance of the device by invoking the kernel in the most immature way as a single stream of 8100 blocks i.e. partially destroying the 2D spatial arrangement of the data, and in turn containing a stream of 256 threads, thereby, completely destroying the purpose.

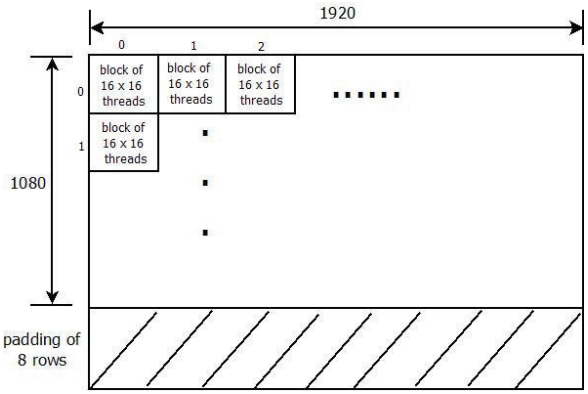

Figure 8: parallel decomposition of the task.

The optimal manner in this case would be to proportionately divide the image into blocks, retaining the base characteristics of the data. These blocks if possible should be aligned in near to square manner. If 256 threads are used, they can be easily structured into a square of (16 x 16). This aids in distributing the load evenly. But as the width (1080) is not an integral multiple of 16, the data should be padded before operating.After padding, the image becomes (1920 x 1088), Figure 8.On the grid view, the blocks can now be easily arranged as a rectangle of (120 x 68).

The padding of data for GPUapplications has received assorted responses. Supporters of this technique stand firm due to the added benefit of not having to model complex arrangements of the threads whilethose who oppose, point out the number of wasted threads that would process dummy data. For this particular application though, the technique works pretty well as the input map originally has dummy data in theregions surrounding the region of interest, see Figure 9. This is customarily done to avoid inconsistencies in calculations for the data points that are outside the region of interest. However, even in other prevailing tasks in GISsystems, mostly the technique of padding the data is applied instead of custom-fitting the programs for the task at hand.

IV. DISCUSSION

For the present paper, to exhibit an exemplary case for the visualization of the derivation of flow directions, the mentioned technique has been applied to a satellite image of the Ramadi Watershed area,Figure 9. The acquired flow directions have been represented in a 2D mapping. This mapping is constructed be applying different colors for distinct directions. The ridges and the variations in the directions of slopes can be clearly identified from the flow direction mapping. This intermediate output serves as the most important stepping stone in various complex geospatial tasks.

While programming for a GPU, the two most prominent APIs' used are CUDA and OpenCL. The latter actually provides a common platform for programming a heterogeneous architecture i.e. irrespective of the manufacturer of the GPU chips, number of devices present, including no device present, and the underlying architecture of the GPU. CUDA on the other hand is specifically designed for the NVIIDA GPU programming. For the present paper, the programming was carried out in CUDA.

Because of the fundamental nature and the foundation upon massive data-based parallelism of such geospatial image processing tasks, they are well suited for the GPUs.Even though a number of GIS based applicationshave beencreated for the GPUs', a major chunk of applications in use are still based on the multicore CPUs. Thus it is of prime importance for the GIS applicationsto evolve and assimilate the power of the GPUs.

V. CONCLUSION

CUDA, having penetrated and excelled in many fields, is now

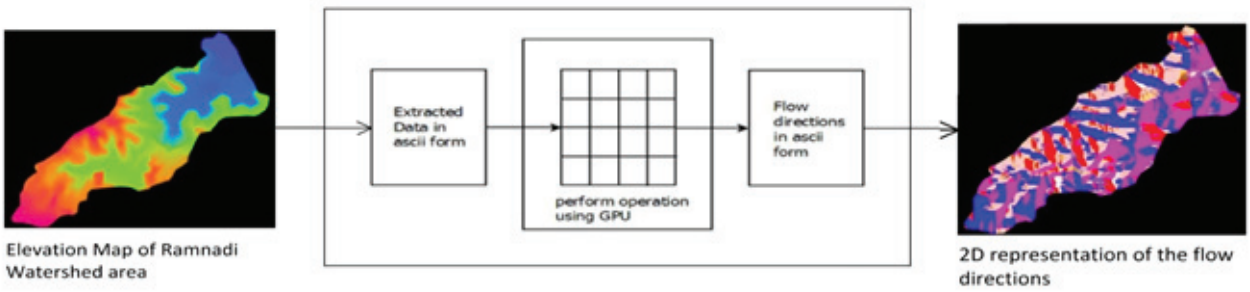

Figure 9: Application of the system to elevation map of Ramnadi watershed area.

ready to conquer the world of GIS. Due to the availability of numerous parallel execution units, large register sets and novel features, GPUs have the potential to tackle the geospatial image processing tasks. Due to the colossal amount of data-based parallelism, which isthe heart of Image processing, the tasks are tailor-made for the GPUs'

Programming in CUDA requires a comprehensive understanding of the GPU architecture. It has been verified from the experiences that even the slightest of ignorant mistakes can cause the performance of GPU to stutter[1].

A refined examination of the already created GIS software for CUDA platform andbetterments in the flow directions derivation algorithm forms the future scope of the paper.

GLOSSARY OF TERMS

MIMD: stands for Multiple Instructions Multiple Data. This model is implemented in the CPU execution design to incorporate task-based parallelism.

SIMD: stands for Single Instruction Multiple Data. This model is mainly used in new generation processors. Typical usage includes vector instructions such as AVX, MMX and SSE.

SPMD:(Single Program Multiple Data) is NVIDIA's novel parallel execution model as an improvement over the SIMD (Single Instruction Multiple Data) model.

ACKNOWLEDGMENT

The author would like to express his sincere thanks to B.C. Maggirwar, Sr. Geologist, GSDA Maharashtra for providing expertise on the topics of spatial analysis and Geospatial Computational Science. Also, the author is grateful towards Dr. B.N. Umrikar, Assistant Professor, Geology Department, Pune University for providing a guidance on Remote Sensing, Geospatial Image processing and the application part of the programming.

REFERENCES

[1] Shane Cook, "CUDA programming, a developers guide to computing with GPUs", 2013, Morgan Kaufmann Publishing, ISBN 978-0-12-415933-4.

[2] [5] David B. Kirk and Wen-mei W. Hwu, "Programming Massively Parallel Processors", 2010, Morgan Kaufmann Publishing, ISBN-10: 0123814723.

[3] SOG (State of GeoInt) Report, 2015, USGIF (United States Geospatial Intelligence Foundation).

[4] Tarboton, D. G., R. L. Bras, and I. Rodriguez–Iturbe. 1991. "On the Extraction of Channel Networks from Digital Elevation Data." Hydrological Processes 5: 81–100.

[5] [2] B. Pierce, "Types and Programming Languages", Cambridge, MA: MIT Press, 2002, ISBN-13: 978-0262162098.

[6] Jenson, S.K. and Domingue, J.O. (1988) Extracting topographic structure from digital elevation data for geographic information system analysis, Photogrammetric Engineering and Remote Sensing, 54(11), 1593-1600.

[7] Lawhead, Joel (2013) Learning Geospatial Analysis with Python. Packt Publishing, ISBN 139781783281138, 364 p.

[8] Grant J. Scott, Georgi A. Angelov, Michael L. Reinig, Eric C. Gaudiello, Matthew R. England "CVTILE: Multilevel parallel geospatial data processing with OpenCV and CUDA", 2015 IEEE International Geoscience and Remote Sensing Symposium (IGARSS), 2015, DOI: 10.1109/IGARSS.2015.7325718.

[9] NatalijaStojanović; DraganStojanović, "Performance improvement of viewshed analysis using GPU", Telecommunication in Modern Satellite, Cable and Broadcasting Services (TELSIKS), 2013, Volume: 02, DOI: 10.1109/TELSKS.2013.6704407.

[10] GIS (Geographic Information Systems) case studies, NVIDIA, HPC (High Performance Computing) documentation.

[11] J. Ghorpade, J. Parande, M. Kulkarni, A. Bawaskar, "GPGPU PROCESSING IN CUDA ARCHITECTURE", (ACIJ), Vol.3, No.1, January 2012.

[12] Fu Yanqing, Liu Fu, GuoDanwei, Liu Xu, LvXiaoling, "The processing of palmprint image based on CUDA", Automatic Control and Artificial Intelligence (ACAI 2012), ISBN: 978-1-84919-537-9.

[13] A. Karnewar, "Designing Python Code for Derivation of Flow Matrix", IJARCSSE Volume 5, Issue 4, ISSN: 2277 128X.

[14] J. O'Callaghan, D. Mark, "The extraction of drainage networks from digital elevation data", 1984, COMPUTER VISION, GRAPHICS, AND IMAGE PROCESSING, Academic Press Inc. 28, 323-344.

[15] Cameron Hughes, Tracey Hughes (2008), "Multicore Programming", Wiley Publications, ISBN 978-81-265-1875-3.