

Comparison of Design and Performance of Snow Cover Computing on GPUs and Multi-core processors

LADISLAV HURAJ¹, VLADIMÍR SILÁDI², JOZEF SILÁČI²

¹Department of Applied Informatics
University of SS. Cyril and Methodius in Trnava
Nám. J. Herdu 2, 917 01 Trnava
SLOVAK REPUBLIC

ladislav.huraj@ucm.sk www.ucm.sk

²Department of Computer Science
University of Matej Bel
Tajovskeho 40
SLOVAK REPUBLIC
{siladi, silaci}@fpv.umb.sk www.umb.sk

Abstract: The aim of this work is the depth of the snow cover computing in the desired point based on the geographical characteristics of a specific geographical point in a modeled area. The measured data are known on few places. These places are coincident with raingauge stations. These data have been collected by many continuous observations and measurements at the specific climatologic raingauging stations of Slovak Hydrometeorological Institute. An interpolation method is necessary to obtain a representation of real situation about whole surface. The main characteristic of the interpolation computing is the fact that it is time-consuming. In paper, we present two cheap approaches of HPC. The first solution is a utilization of graphics processing units (GPUs) where the availability of enormous computational performance of easily programmable GPUs can rapidly decrease time of computing. The second one is a utilization of multi-thread CPUs. In our article we demonstrate how to deploy the CUDA architecture, which utilizes the powerful parallel computation capacity of GPU, to accelerate computational process of snow cover depth using the inverse-distance weighting (IDW) method. The performance of GPU we face with OpenMP implementation of IDW method. We consider variable number of threads per CPU. The outputs are visualized by the GIS Grass tool.

Key-Words: GPGPU, CUDA, Multi-core processor, OpenMP, Snow cover depth, Interpolation

1 Introduction

A big attention in the world is given to the research of prognosis if the depth of snow cover depends on influence of global warming. The high performance computing resources (supercomputers, computer grids) are used for this purpose [1,2,6]. In most cases, the processing relates to a much larger area than continents or countries [2,3,4,5]. Determination of the snow cover depth in a defined territory seems to be problematic because of absence of raingauge stations. Therefore, the computationally time-consuming interpolation methods for determination of the snow cover depth are necessary where the data from nearby raingauge stations are used. A hardware development allows achieving good result and low cost.

Our objective in this paper is to compare two possibilities to obtain computation acceleration. The

aim of method is to develop GPGPU (General-Purpose computing on Graphics Processing Units) and multi-core CPU design that can be used for implementation of snow cover modeling and for acceleration of the computing. The utilization of GPGPU and multi-core CPU will be tested for one method used to use them for the interpolation in geographical information systems (GIS). The method is inverse-distance weighting method (IDW).

GPGPU is the method of using the graphical processing unit (GPU) to process programs that are normally executed by CPUs. This has been possible with the addition of programmable stages to the GPUs and with development toolkits and libraries from the vendors [15].

Graphics Processing Units (GPUs) are widely used among researchers and developers as accelerators for applications outside the domain of

traditional computer graphics. This trend largely results from the great improvements in GPU programmability [7]. CUDA (Compute Unified Device Architecture) is a parallel computing architecture developed by NVIDIA which presents to the programmer a fairly generic abstraction of a manycore architecture supporting fine-grained parallelism. CUDA and the GPU therefore provide massive, general purpose parallel computation resources with the potential for speedups of data processing.

The multi-core processors are ordinary used in the high-performance computing (HPC). Our aim of the experiment has been to utilize the multi-core CPUs, which are used ordinary in personal computers, in the effective and the simplest way. The program executed by single core CPU has been remade into a parallel program with OpenMP library.

In our article the processed data by GPGPU is climatologic data. For the Slovak Republic are available widely relevant data monitored by Slovak Hydrometeorological Institute, which has been recording the trends of snow conditions on the specific hydro-meteorological stations for at least 20 years. Using these data, we have tried to generate computer models of the depth of snow cover for a defined area during the period of time and to visualize the data by geographic information systems.

The rest of the paper is organized as follows. First we review the CUDA and OpenMP programming models briefly in section 2. Section 3 introduces the background of snow cover modeling. Section 4 describes the algorithms and their mapping on the GPU and multi-core CPU with experimental results. The conclusion comes in section 5 with an outlook to further work.

1 CUDA and OpenMP programming models

In paper, we consider three computer architectures according to Flynn's taxonomy. The first one is SISD computer architecture represented with usual processor for PC. The second one is SIMD computer architecture represented with graphic card. And the last one is MIMD architecture represented with the usual multi-core processor for PC. The SIMD and MIMD belong in the part of parallel architectures, but programming style is pretty different. The MIMD architecture has got a variety of implementations. Most often they are multiprocessors and multicomputers. The

programming style is in either event different. The multiprocessors allow using automatic parallelization unlike multicomputers. The automatic parallelization is applicable on multi-core processors so as for the multiprocessors. An utilization of OpenMP libraries in an initial sequential program is easy way to do that.

Graphic cards, especially GPUs, represent quite new possibility for HPC. Parallel computing architecture developed by NVIDIA named CUDA allows programmers to use the graphic cards for parallel programming.

2.1 CUDA

The Compute Unified Device Architecture (CUDA) allows developers to use the C programming language for the development of general-purpose applications using fine-grain parallelism. CUDA is currently supported only on NVIDIA GPUs, but recent work has shown the viability of compiling CUDA programs for performance on multi-core processors [8]. A simple extension to C had invoked that more non-graphics developers port their existing applications to CUDA. CUDA consists of a runtime library and an expanded version of C. CUDA gives developers access to the native instruction set and memory of the parallel computational elements in CUDA GPUs. It includes the CUDA Instruction Set Architecture (ISA) and the parallel compute engine in the GPU [9].

A single source program contains both the host (CPU) code and the device (GPU) code which are automatically separated and compiled by the CUDA compiler tool chain, Figure 1.

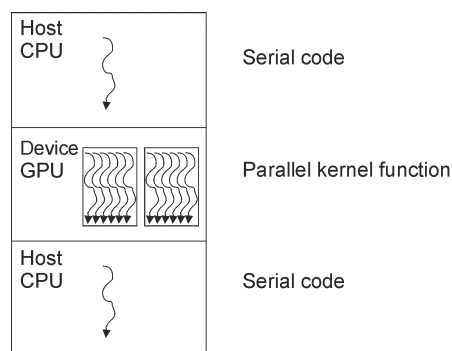


Fig. 1 Heterogeneous Programming in CUDA

CUDA is based on the notion of a kernel function, which is a single routine that is invoked concurrently across many thread instances; a software controlled scratchpad, which CUDA calls the “shared memory”, in a Single Instruction Multiple Data (SIMD) fashion for each SIMD core;

and barrier synchronization. Each GPU thread is a fully independent, scalar, and can execute arbitrary code and access arbitrary addresses. Moreover, each of the GPU threads is given a unique ID that is accessible within the kernel through the built-in threadIdx variable [10,11,13].

CUDA presents a virtual machine consisting of an arbitrary number of streaming multiprocessors (SMs), which appear to be 32-wide SIMD cores with a total of up to 512 thread contexts. Kernels are applied to a 2D grid that is divided into as many as 64K 3D thread blocks. Each thread block is mapped in its entirety and executes to completion on an arbitrary SM. All thread blocks in a kernel run to completion before a subsequent kernel may start, offering a global memory fence. Data in the GPU is stored in special structure. Each group of data is called a texture [10,11,13,14].

2.2 OpenMP

OpenMP (Open Multi-Processing) is an application programming interface (API) that supports shared-memory multiprocessors programming in C, C++ and Fortran. The interface consists of a set of compiler directives, library routines, and environment variables that influence run-time behavior. OpenMP is a portable, scalable model that gives shared-memory parallel programmers a simple and flexible interface for developing parallel applications for platforms ranging from the desktop to the supercomputer [22]. It is useful programming tool to develop parallel programs for desktops equipped with some multi-core processor.

The interface allows dividing a code into two inconsequent parts. The first part of code is executable parallel and the second part contains the part of non-parallelized (sequential) code, Figure 2.

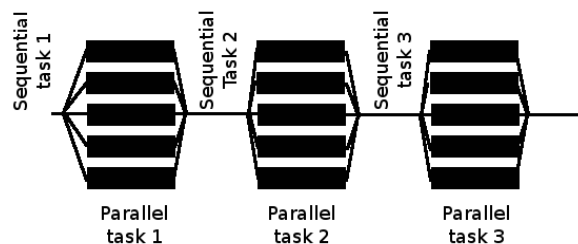


Fig. 2 Master thread forks off a number of threads

The code for parallel execution is enclosed into compiler directives of OpenMP `#pragma omp <rest of pragma>`.

Programmers have to find out the parts, which are suitable for parallelization, place synchronization, locate place of variables in memories (shared, local), set appropriate number of threads, etc. The allocation of threads is not so important task for programmers. Using OpenMP in this way we can achieve task and data parallelism.

3 Snow Cover Modeling and Interpolation

The depth of the snow cover is a very variable meteorological element in the landscape. It depends on many factors, mainly on snow precipitation, altitude, air temperature, profile of the relief, solar power, cloudiness, air temperature inversion, etc. The measurement of the snow cover is taken by meteorological, climatologic and precipitation stations. The total depth of snow is measured and stored, i.e. the depth of snow and the depth of new snow cover. We are able to analyze the depth of the snow cover in detail.

The analysis is based on many continuous observations and measurements at the specific climatologic stations. Geographically we can strictly characterize all these gauging places by the altitude, latitude and longitude, as well as by the detailed characteristics of the relief shape.

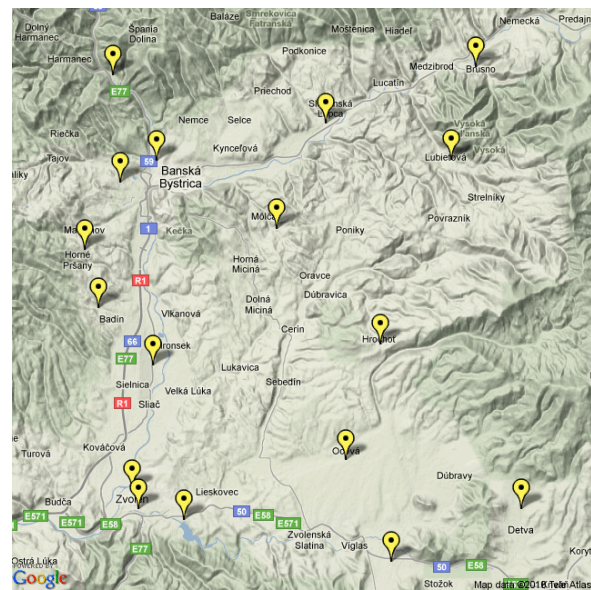


Fig. 3 Location of the 17 meteorological stations [20]

The aim of this work is the depth of the snow cover computing in the arbitrary point based on the geographical characteristics of a specific geographical point in a modeled area. The result is derived from the available data, which have been

obtained from metrological stations, climatologic stations and rain gauge stations from a defined landscape area [17, 18].

As an example of the application of our designed method the geomorphological entity Zvolenská kotlina as a part of Slovak Republic has been chosen, which is exactly defined by its borders. We use the digital terrain model of this entity, which has been done. The 17 meteorological stations of Slovak Hydrometeorological Institute are situated in this defined area and their long-term measurements are available for our research, Figure 3. We decided for the period of years 1990–2009.

The input data are stored in the large matrices. The output values depend on the time-consuming computing process and CUDA and OpenMP were deployed to accelerate these operations.

In addition, the problem of used coordinates system has to be solved, because the coordinates system used in Czech Republic and Slovak Republic was another than the coordinates system used in GIS Grass. Data provided by Slovak Hydrometeorological Institute have coordinates (latitude, longitude) measured in WGS-84 coordinates system and on the other hand data visualized in GIS Grass has been coded in S-JTSK [19]. The S-JTSK coordinates system was implemented and the conversion of coordinates from WGS-84 into S-JTSK was necessarily performed [21]. This data transformation was applied during computing only once to transform WGS-84 coordinates of the 17 raingauge stations into S-JTSK coordinates system. The consequent computations were performed in the S-JTSK coordinates system.

3.1 Interpolation Methods

A couple of interpolation methods are used in GIS to obtain required data from measured data. We experimented with three of them:

1. Triangular linear interpolation method,
2. Kriging interpolation method,
3. Inverse-distance Weighting Method.

In our experiment we have to model 3D space. However, we use 2D data in the interpolation methods. The altitude is omitted. The little change would be made in the distance computation if the altitude was considered. This simplification does not have an essential influence on examined comparison of the computing performance.

The triangular method offers an estimation of the unknown value by linear interpolation. The missing climatologic value in known point, given by the coordinates, is calculated from data of the three

closest points. Point lies within the area bounded by three lines (triangle). Each line passes through the two points, Figure 4. This method is relatively simple. It is suitable for basic geographical tasks. In relation to the time complexity of computing is this method relatively fast and it can be implemented and performed on weaker computers. For the modeling of complex situations is necessary to choose more efficient and accurate method, e.g. kriging.

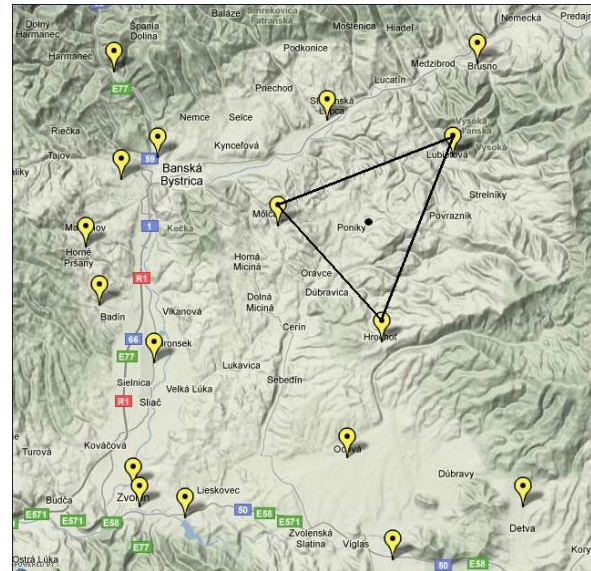


Fig. 4 Linear interpolation method for one point in the location with 17 raingauge stations [20]

Kriging belongs to the family of linear least squares estimation method. This method has been used in geography and in hydrogeography, too. It is one of the frequently used methods to obtain the reliable estimations (local or global) of observed values. Despite of exact results this method uses more complex mathematical apparatus and it is very time consuming.

We decided to use an inverse-distance-weighting (IDW) algorithm to interpolate the snow cover measurements, as a compromise between time complexity and accuracy.

3.2 Inverse-distance Weighting Method

This deterministic model is relatively fast and easy to compute, and straightforward to interpret. The IDW method as a deterministic spatial interpolation model is one of the more popular methods adopted by geoscientists and geographers partly because it has been implemented in many GIS tools [12]. The

general assumption of this method is that the attribute values of any given pair of points are related to each other, but their similarity is inversely related to the distance between the two locations. The general idea of IDW is that the attribute value of an un-sampled point is the weighted average of known values within the neighborhood, and the weights are inversely related to the distances between the location of un-sampled point and the location of its neighbors. The value of inverse-distance weight is modified by a constant power with increasing distance. This dependence can be expressed by the relationship (1).

$$y_0 = \frac{\sum_{i=1}^n \frac{y_i}{d_i^k}}{\sum_{i=1}^n \frac{1}{d_i^k}} \quad (1)$$

Where y_0 is value of un-sampled point; d_i denotes the distance between un-sampled point and sampled location i and y_i is given value at sampled locations i , Figure 5.

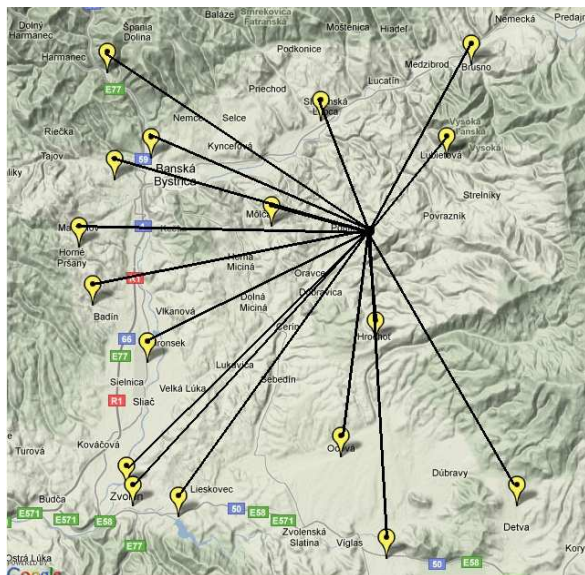


Fig. 5 IDW method for one point in the location with 17 raingauge stations [20]

4 Sequential and Parallel Implementation of IDW Method

To write a CUDA program and an OpenMP program, we designed the sequential version of program. Our implementation was written in language C, because the code was easier modifiable on code for GPU and multi-core CPU. In addition to the changes in the program code, the time

calculating commands had to be changed for each programming model. We used the header file *time.h*:

```
#include <time.h>
...
clock_t start = clock();
...
printf("Elapsed time: %f\n",
((double)clock() -
start) / CLOCKS_PER_SEC);
```

A tendency to use these commands suitable for the sequential program made towards incorrect results in the parallel programs. These commands had to be overridden with related commands from the corresponding OpenMP or CUDA libraries. We used header file *omp.h* in the OpenMP implementation:

```
#include <omp.h>
...
double startomp = omp_get_wtime();
...
double endomp = omp_get_wtime();
printf("Elapsed time openmp:
%.4g\n", endomp - startomp);
```

Similarly, we used header file *omp.h* in the CUDA implementation:

```
...
cudaEvent_t c_start, c_stop;
cudaEventCreate(&c_start);
cudaEventCreate(&c_stop);
cudaThreadSynchronize();
cudaEventRecord(c_start, 0);

//copying data from host to device
//call of kernel
//copying data from device to host
...
cudaEventRecord(c_stop, 0);
cudaThreadSynchronize();
cudaEventElapsedTime(&memsettime,
c_start, c_stop);
cudaEventDestroy(c_start);
cudaEventDestroy(c_stop);
printf("Cas vypoctu raster [CUDA -
cudaEventElapsedTime]: %.4f\n",
memsettime / 1000.0);
```

The change of the sequential program code to the multi-core CPU's program was simpler than the change to the CUDA implementation. In addition,

we had to provide a data transfer from the host (CPU) to the device (GPU – graphic card) and vice-versa in the CUDA implementation.

4.1. Sequential Program

The sequential program consists of several functions. The function IDW has got significant position, because it is the implementation of the IDW method. This function (and whole program) uses common programmer-defined data-type RASTER. This data-type contains all values needed for GIS Grass visualization (location of map segment, number of raster columns, number of raster rows, array of visualized values, etc.). The function IDW computes absent data from measuring data. The measuring data are stored in data-type DATA. This data-type keeps data from rain gauge stations. The code of IDW method in language C has this representation:

```
RASTER* IDW(RASTER* raster, DATA*
data)
{
    int i,j;
    float sum_up;
    float sum_down;
    float distance;
    for(i=0;i<raster->points_count;
i++)
    {
        sum_up = 0;
        sum_down = 0;
        for(j =0; j < data->_data;
j++)
        {
            distance = SQR(raster-
>x[i] - data->x[j]) + SQR(raster-
>y[i] - data->y[j]);
            sum_up += (data-
>value[j]/distance);
            sum_down +=
(1.00/distance);
        }
        raster->value[i] =
sum_up/sum_down;
    }
    return raster;
}
```

4.2. Parallel Program for GPU

To write a CUDA program, we obeyed the following steps from [7] starting from a sequential version and proceeds through:

4. Identify the kernel and package it as a separate function.
5. Specify the GPU threads' grid and partition the computation among these threads.
6. Manage data transfer between the memory of host and GPU memories before and after the kernel invocation.
7. Make memory optimizations in the kernel.
8. Make further optimizations in the kernel for optimizing the single thread performance and the level of parallelism.

The CUDA kernel is written below:

```
__global__ void cudaIDWKernel(float
*raster_x, float *raster_y, float
*raster_values, int points_count,
float *data, float *data_x, float
*data_y, int data_count)
{
    unsigned int inx = blockDim.x *
blockIdx.x + threadIdx.x;
    unsigned int j;

    float sum_up;
    float sum_down;
    float distance;
    if (inx < points_count)
    {
        sum_up = 0;
        sum_down = 0;
        for(j =0; j < data_count;
j++)
        {
            distance =
SQR(raster_x[inx] - data_x[j]) +
SQR(raster_y[inx] - data_y[j]);
            sum_up += (data[j] /
distance);
            sum_down += (1.0f /
distance);
        }
        raster_values[inx] = sum_up /
sum_down;
    }
}
```

4.3. Parallel Program for Multi-core processor

The parallelization of the sequential program is not difficult. OpenMP directives of C compiler are placed into previous sequential program code. A separation of variables (firstprivate, private, shared) is made by programmer, only. The adapted program code is written below:

```

RASTER* IDW_OMP(RASTER* raster,
DATA* data)
{
    int i,j;
    float sum_up;
    float sum_down;
    float distance;
    omp_set_num_threads(n);
    #pragma omp parallel for
    firstprivate(raster,data)
    private(i, j, sum_up, sum_down,
distance) schedule(dynamic, 500)
    for(i=0;i<raster->points_count;
i++)
    {
        sum_up = 0;
        sum_down = 0;
        for(j =0; j < data->_data;
j++)
        {
            distance = SQR(raster-
>x[i] - data->x[j]) + SQR(raster-
>y[i] - data->y[j]);
            sum_up += (data-
>value[j]/distance);
            sum_down +=
(1.00/distance);
        }
        raster->value[i] =
sum_up/sum_down;
    }
    return raster;
}

```

5 Solution and Performance Evaluation

In the experimental evaluation of our computing we focused on the performance improvements from CUDA implementation and multi-core implementation. We used two computers and investigated two likely scenarios of use:

1. CPU
2. GPU.

We made five experiments on CPU. Two of them ran on dual-core CPU and other three ran on quad-core CPU with Hyper-Threading Technology.

The first couple of experiments were executed on dual-core processor Intel Core 2 Duo CPU E7400 @ 2.80GHz (C2D E7400). In this case the OpenMP program was executed as single-thread application and then as two-threads application. The second approach allowed to use full computing power of this CPU.

The second triplet of experiments was executed on quad-core processor Intel Core i7-920 2.66GHz (Core i7-920). In this case the OpenMP program was executed as:

1. single-thread application,
2. four-thread application,
3. eight-thread application.

The third approach allowed to use full computing power of this CPU.

All experiments on CPUs were compared to the second experiment on the GPU. Run times for GPU implementation was measured on the dual-core machine, where the first couple of experiments were executed. This machine was equipped with two a NVIDIA GeForce 9800 GT graphic cards. These cards offer 112 CUDA cores, 512 MB RAM and CUDA compute capability 1.1.

The computers were installed with operating system Linux Ubuntu, version 9.04. The NVIDIA driver of graphic cards had a version 195.36.15. The C code was compiled using GCC, version 4.3.3. The CUDA code was compiled using NVCC, version 0.2.1221, CUDA Toolkit, version 3.0.

The speed-up of computing was detected on ten kinds of raster. The two-dimensional raster differed in pixel density. One-dimensional matrices represented each raster, because of format of GIS Grass input file. The sizes of the matrices had got ten values from 1000×1000 to $10\,000 \times 10\,000$, Table 1.

In case of the OpenMP program we modified the type of processor and then the number of used threads for each type of them. We used OpenMP command `omp_set_num_threads(n)` for that. Because OpenMP is dedicated for numbers of threads, we had to set schedule on 500. It was the best value to reach desired execution time. We used directive of compiler `#pragma omp parallel for firstprivate(raster,data) private(i, j, sum_up, sum_down, distance) schedule(dynamic, 500)`. The value was founded by experimentation.

We obtained fifty results of measured time. The results are figured in columns: C2D E7400, C2D E7400 (2 threads), Core i7-920 (1 thread), Core i7-920 (4 threads), Core i7-920 (8 threads), Table 1.

We made a couple experiments to fine set block size. The first of all we tried to use maximum threads per block, Figure 6. Finally, in experiment with CUDA, we used CUDA Occupancy Calculator to set up two parameters: number of block (grid) and number of threads per block (blocksize). These parameters are important in command, which calls kernel function:

```

...
dim3 dimBlock(blocksize);
dim3 dimGrid(grid);
cudaIDSKernel<<<dimGrid,
dimBlock>>>(d_x, d_y, d_values,
sendBlockSize, d_data, d_data_x,
d_data_y, data->_data);
...

```

Tab. 1 Execution time in seconds.

N	C2D E7400 (1 thread)	C2D E7400 (2 threads)	Core i7-920 (1 thread)	Core i7-920 (4 threads)	Core i7-920 (8 threads)	GeForce 9800 GT512MB
1.00E+06	0.1779	0.0892	0.2079	0.0695	0.0550	0.0500
4.00E+06	0.7117	0.3557	0.8311	0.2771	0.2089	0.1947
9.00E+06	1.6000	0.7996	1.8687	0.6231	0.4687	0.4345
1.60E+07	2.8450	1.4840	3.3215	1.1114	0.8321	0.7738
2.50E+07	4.4430	2.2210	5.1892	1.7305	1.2976	1.2065
3.60E+07	6.3970	3.1960	7.4727	2.4913	1.8692	1.7425
4.90E+07	8.7040	4.3510	10.1710	3.3907	2.5473	2.3637
6.40E+07	11.3700	5.6830	13.2830	4.4284	3.3192	3.0826
8.10E+07	14.3900	7.1920	16.8009	5.6045	4.2036	3.8956
1.00E+08	17.7700	8.8780	20.7525	6.9194	5.1861	4.8208

Method	#Calls	grid size X	block size X	static shared memory per block	registers per thread
1. cudaMemcpy	1	1958	512	76	12
2. cudaMemcpy	1	7821	512	76	12
3. cudaMemcpy	1	17590	512	76	12
4. cudaMemcpy	1	31266	512	76	12
5. cudaMemcpy	1	48848	512	76	12
6. cudaMemcpy	2	35168	512	76	12
7. cudaMemcpy	2	47866	512	76	12
8. cudaMemcpy	2	62516	512	76	12
9. cudaMemcpy	3	52747	512	76	12
10. cudaMemcpy	3	65118	512	76	12

Fig. 6 CUDA Visual Profiler: grid size, block size, shared memory per block and register per threads

According to the results of the CUDA Occupancy Calculator we chose 128 threads per block. The number of block varied according to size of computed matrix. The program used 12 registers per thread and 76 B shared memory per block.

These values yielded optimal results using the graphic card GeForce 9800 GT. These specifications of GPU (compute capability 1.1) enable 83% utilization of the multiprocessors, only.

The size of block has some limits. In case of huge matrices we had to divide them. We could use 2D array, but we preferred one-dimensional array solution.

The results of experiments with GPU are figured in column: GeForce 9800 GT512MB, Table 1. GPU implementation has got one stumbling block. The data have to be moved from a host (CPU) to (GPU). This fact could be have significant computing time overhead. We used the CUDA Visual Profiler to clarify this problem. The profiler showed in all cases, that the overhead had been no more than 30 % of the executing time, Figure 7. This communication overhead plays no significant role in execution in comparison with other observed execution times, Figure 9.

Method	#Calls	GPU time	instruction throughput
1. cudaMemcpy	17	1.52592e+07 82.37	0.0512277
2. cudaMemcpy	102	2.45148e+06 13.23	
3. cudaMemcpy	17	812598 4.38	

Fig. 7 CUDA Visual Profiler: Computing time – data transfer and kernel execution

The summary of the computing times for all 10 experiments on GPU is showed on Figure 8. The bars of the data transfers are severalfold smaller then the bars of kernel executions. The first five triplets of bars described one execution time of matrices computation with size (in stages): from 1000×1000 to 5000×5000 . The next two triples of bars represent execution time of matrix with size 6000×6000 . The matrix was divided into two parts, because of the block size limit mentioned above. The same occurred in case of matrix with size 7000×7000 . The matrices with larger size then 7000×7000 were divided onto three parts and computed consequently.

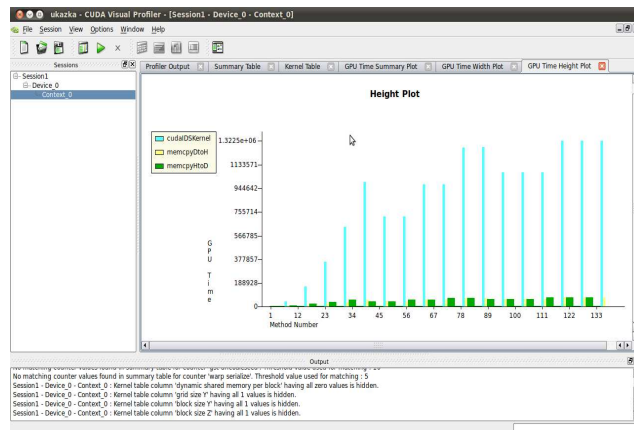


Fig. 8 CUDA Visual Profiler: Computing time plot of 10 experiments on GPU

Results of our experiments are matched in the chart, Figure 9. The executing time sunk in dependence on number of used threads. The dual-core CPU returned better execution time than quad-core CPU in case of one used thread. GPU achieved the best result, however quad-core processor Intel Core i7-920 achieved comparable results, if all 8 threads were used. The determining factor could be a price of hardware. We used run-out model of graphic card (4 years old). Its price is around 90.21 € (current price). The price of Intel Core i7-920 2.66GHz is 272.63 € (current price).

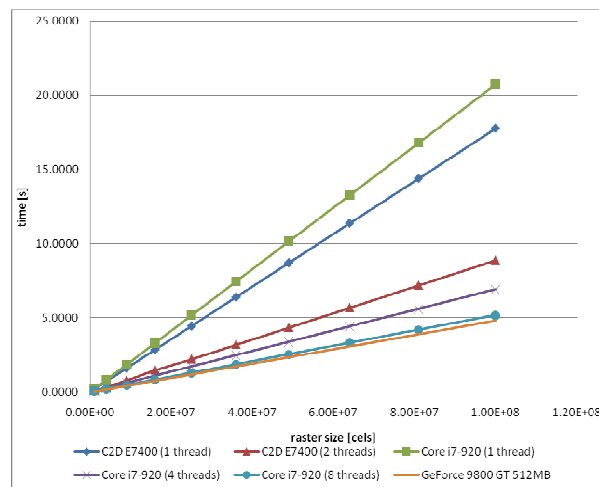


Fig. 9 Chart of execution time depending up problem size (raster size)

For our experiments we chose data from January, the 5th 2000, collected by the 17 gauge stations mentioned above, Figure 4. Outputs of our experiments were visualized by the GIS Grass (Geographic Resources Analysis Support System) version 6.4.0. Figure 10 shows an example of

graphical output of our experiment when the depth of snow cover is visualized in centimeters.

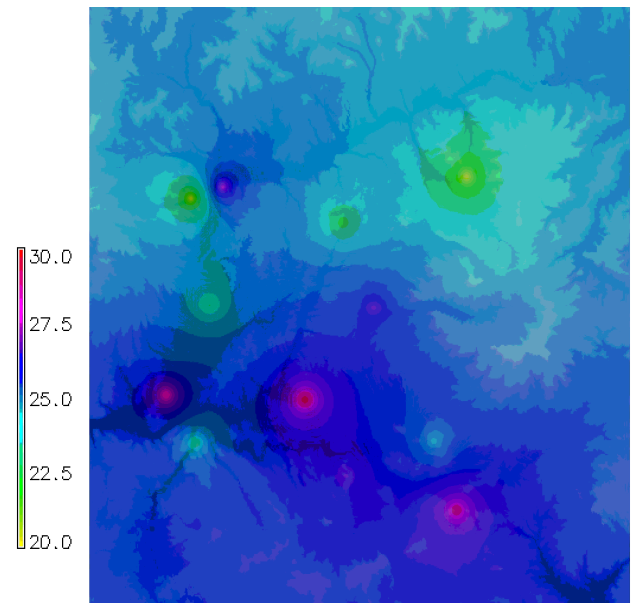


Fig. 10 Visualized depth of snow cover of Zvolenská kotlina

6 Conclusion

In our article we have shown that snow cover computing benefit from using CUDA GPUs, or multi-core CPU. For computing of non-cover points we have used an inverse-distance-weighting algorithm and we process meteorological data of twenty years period. The CUDA implementation provides significant speedups on the C implementation.

Overall, our experiences with CUDA show the power of the GPU as a parallel platform, and help demonstrate how to utilize the GPGPU programming for geographic data processing. The same was showed in case of multi-core processors. Furthermore, the both GPU and multi-core CPU offer comparable computing performance. The determining factor to use one or other method is the price of hardware.

In the future, we are going to implement more efficient algorithms via implementing other interpolation methods where we suggest more rapidly speed-up of the GPGPU programming. Moreover, we also plan to transform the computation from GPGPU programming to Grid Computing.

References:

- [1] Shukinov, A. I., Butenkov, S. A., Zhukov, A. L.: Blanket of Snow State Physical Model For Clustering Calculations Based on Information Granulation. In: *Izvest'ja JUFU. Techničskie nauki*. Vol. 97, No. 8, 2009, pp. 213-223, ISSN 1999-9429.X2. Author, Title of the Book, Publishing House, 200X.
- [2] Šmakín, A. B., Turkov, D. B., Michjlov, A. J.: Model' snežnogo pokrova s učetom sloistoj struktury i eeevoľucii. In: *Kriosfera Zemli*. Vol. XIII, No. 4, pp. 69-79, ISSN 1560-7496.
- [3] Zhiming, L., Chunxiao, Z., Jianxia, S., YE, W.: The Extraction of Snow Cover Information Based on MODIS Data and Spatial Modeler Tool. In: *Proceedings of the 2008 International Workshop on Education Technology and Training & 2008 International Workshop on Geoscience and Remote Sensing*. Vol. 1, 2008, pp. Printed: 29. 4. 2010 9:48:53 VEGA/A- 2 0 1 1 /3 (item - continued) 836-839, ISBN:978-0-7695-3563-0 .
- [4] Takala, M., Pullainen, J.: Detection of Snow Melt Using Different Algorithms in Global Scale. In: *Proceedings of 2008 IEEE International Geoscience & Remote Sensing Symposium*, July 6-11, 2008, Boston, Massachusetts, U.S.A.
- [5] Klaes, K.D.: The EPS/Metop System as a contribution to Operational Meteorology and Earth System Monitoring, In: *2005 WSEAS Int. Conf. on REMOTE SENSING*, Venice, Italy, November 2-4, 2005 (pp.43-48)
- [6] Kirk, D., Hwu, W.: *Programming Massively Parallel Processors. A Hands-on Approach*. Burlington: Morgan Kaufmann Publishers, 2010, ISBN: 978-0-12-381472-2.
- [7] Han, T.D., Abdelrahman, T. S., "hiCUDA: High-Level GPGPU Programming," *IEEE Transactions on Parallel and Distributed Systems*, 31 Mar. 2010, IEEE Computer Society Digital Library. IEEE Computer Society, <<http://doi.ieeeecomputersociety.org/10.1109/T-PDS.2010.62>>.
- [8]] Stratton, J. A., Stone , S. S., and Hwu , W. W., "MCUDA: An efficient implementation of CUDA kernels for multi-core CPUs," in *Proceedings of the 21st International Workshop on Languages and Compilers for Parallel Computing*, Springer-Verlag Berlin Heidelberg, 2008.
- [9] Wang, Y., Feng, Z., Guo, H., Ch. Yang, He, Y., "Scene Recognition Acceleration Using CUDA and OpenMP," pp.1422-1425, *First IEEE International Conference on Information Science and Engineering*, 2009.
- [10] Boyer, M., Tarjan ,D., Acton, S.T., Skadron, K., "Accelerating leukocyte tracking using CUDA: A case study in leveraging manycore coprocessors," pp.1-12, *IEEE International Symposium on Parallel&Distributed Processing*, 2009.
- [11] "NVIDIA CUDA Programming Guide v2.0," Availabe on: http://developer.download.nvidia.com/compute/cuda/2_0/docs/NVIDIA_CUDA_Programming_Guide_2.0.pdf , July 2008.
- [12] Lu, G. Y., Wong, D. W.: An adaptive inverse-distance weighting spatial interpolation technique, In: *Computers & Geosciences*, Vol. 34, No. 9, September 2008, pp. 1044-1055.X1.
- [13] Romero, S., Trenas, M.A., Gutierrez, E., Zapata, E.L.: Locality-Improved FFT Implementation on a Graphics Processor, In: *Proceedings of the 7th WSEAS Int. Conf. on Signal Processing, Computational Geometry & Artificial Vision*, Athens, Greece, August 24–26, 2007
- [14] Vlček, V.V.: Computation of Filtered Back Projection on Graphics Cards, In: *Proceedings of the 5th WSEAS Int. Conf. on SIGNAL, SPEECH and IMAGE PROCESSING*, Corfu, Greece, August 17-19, 2005 (pp 34–39).
- [15] Asavei, V., Moldoveanu, A. D. B., Moldoveanu, F., Morar, A., Egner A.: GPGPU for Cheaper 3D MMO Servers. In: *Proceedings of the 9th WSEAS International Conference on TELECOMMUNICATIONS and INFORMATICS (TELE-INFO '10): New Aspects of Telecommunications and Informatics*, Catania, Sicily, Italy, May 29-31, 2010, pp. 238-243.
- [16] Blišťan, P.: Priestorové modelovanie geologických objektov a javov v prostredí GIS systémov, In: *Acta Montanistica Slovaca*, Vol. 10, No. 3, pp. 296-299, 2005, ISSN 1335-1788.
- [17] Polčák, N.: Možnosti spracovania mezoklímy a miestnej klímy v územiach s chýbajúcou klimatickou databázou na príklade Biosférickej rezervácie Východné Karpaty. In. *Geografický časopis*. 52.GÚ SAV, Bratislava, 2000, pp. 181–191, ISSN 0016-7193.
- [18] Hlásny, T., Polčák, N.: Digitálny model reliéfu a jeho využitie vo fyzickej geografii. In. Baran, V. (edit.): *Geografické štúdie Nr.8. Premeny Slovenska v regionálnom a didaktickom kontexte*. FPV UMB, Banská Bystrica, 2001, pp. 239–245, ISBN 80-8055-583-4.

- [19] Gábor, T., Danišík, M.: Aproximácia Křovákovo zobrazenia Lambertovým konformným kuželovým zobrazením na území Slovenska pre potreby GIS a GPS. In: *Kartografické listy*. Vol. 11, pp. 100–102, Bratislava, 2003, ISBN 80-89060-04-8.
- [20] “Zvolenská kotlina”, 48°34'49.00"N and 19°07'34.00"E. Google Earth. April 15, 2010.
- [21] Hrdina, Z: *Transformace souřadnic ze systému WGS-84 do systému S-JTSK*. Praha: ČVUT, 1997, p. 21.
- [22] “The OpenMP API specification for parallel programming,” Available on: <http://openmp.org/wp/about-openmp/>, April 2008.