Optimization solutions for the segmented sum algorithmic function

ALEXANDRU PÎRJAN
Department of Informatics, Statistics and Mathematics
Romanian-American University
1B, Expozitiei Blvd., district 1, code 012101, Bucharest
ROMANIA
alex@pirjan.com

*Abstract:*In this paper, there are depicted optimization solutions for the segmented sum algorithmic function, developed using the Compute Unified Device Architecture (CUDA), a powerful and efficient solution for optimizing a wide range of applications. The parallel-segmented sum is often used in building many data processing algorithms and through its optimization, one can improve the overall performance of these algorithms. In order to evaluate the usefulness of the optimization solutions and the performance of the developed segmented sum algorithmic function, I benchmark this function and analyse the obtained experimental results.

*Keywords:*GPGPU, CUDA, segmented sum, parallel processing, registers, memory, warp.

1Introduction

In contrast to traditional Graphics Processing Units (GPUs), novel GPUs from the latest generations provide an increased computational power and memory bandwidth, in the same time being much more easier to program. These processors have evolved into General Purpose Computation Graphics Processing Units (GPGPU), gaining the attention of software developers and having applications in different scientific fields, such as medicine, telecommunications, financial, image processing etc. The GPGPUs' computational processing power is comprised of hundreds parallel processing cores and surpasses the parallel processing power of Central Processing Units (CPUs) to a large extent. Therefore, these processors are particularly useful when processing huge data workloads, as the execution time is significantly reduced. An important advantage of a GPGPU compared to a CPU consists in the performance per watt and the resulting low processing cost that comes with an increased number of features. Initially, GPUs have been designed to accelerate solely graphics rendering, but afterwards, due to an increase of graphic requirements, they evolved from a one specialized architecture to many multi-purpose architectures, offering a wide range of features, not only video rendering.

The Compute Unified Device Architecture (CUDA) enables a NVIDIA graphics processing unit to execute software code sequences written in different programming languages such as OpenCL, FORTRAN, Direct Compute, C, C++ and others. A Compute Unified Device Architecture program invokes multiple program kernels that launch a set of parallel threads, grouped in parallel thread blocks and grids of thread blocks. Every thread has an unique associated ID, register and private memory [1].

The programming environment of the Compute Unified Device Architecture (CUDA) exposes three levels of abstractions (the thread blocks hierarchy, shared memory and barrier synchronization) that can be employed by the developer using a minimal C-extensions set language [1]. These result in both fine-grained parallelism (for data and threads) and large grained parallelism (for data and tasks). A big problem is divided into many small-sized ones that can be parallelly processed by the available processing cores, thus allowing the threads to cooperate within a thread block.

The Graphics Processing Unit (GPU) instantiates kernel grids, while a streaming multiprocessor (SM) processes the thread blocks. The processing cores within the SM execute the block's threads being able to process them in warps (groups of up to 32 threads). Each SM has a 32-bit register memory and a block-level shared memory thatcan be accessed by the multiprocessor's cores and varies from one GPU generation to another. The SM also contains two read-only memory caches, designed for storing the texture and the constants.

The great amount of parallel processing power of the CUDA-enabled GPUsbecomes available to application developers through the CUDA-C programming language that allows them to define how the tasks are being decomposed in order to be executed by the Graphics Processing Unit [2].

As the CUDA-C language offers a detailed control of the resources' allocation, it helps obtaining a set of basic powerful algorithmic functions that can be used in optimizing a wide range of high resource consuming applications [3]. Among these basic functions is the segmented sum algorithmic function that is developed and analysed in this paper.

By applying the parallel-segmented sum function, the developers have the opportunity to implement efficient data processing algorithms without having to know how resources are allocated or to optimize the algorithmic function. The developer is not compelled to define how tasks are being divided and allocated to the CUDA cores as the parallel-segmented sum function automatically defines all the optimal technical specifications. The parallel-segmented function is a common functional block useful in building many data processing algorithms, thus, through its optimization, one can improve the overall performance of all these algorithms. Another aspect that has been taken into account when developing the parallel-segmented sum function was the modularity, through which the designed function could be easily integrated in designing other software applications. The designed parallel-segmented sum function offers major advantages to the developers, being a useful tool in developing data processing algorithms with significant economic advantages. The parallel-segmented sum function also offers the advantage of reusing its source code when developing different data processing algorithms, thus bringing a significant improvement to their performance.

2Designing an efficient parallel-segmented sum function in CUDA

The parallel-segmented sum function generalizes the parallel prefix sum function [2] by simultaneously summing in parallel arbitrary sized partitions (segments) of the input vector.

Given an associative binary operator $*$ on the set of real numbers and $v$an input segmented$n$-dimensional real sequence$v = [(a_0, \ldots, a_{i_1}), (a_{i_1+1}, \ldots, a_{i_2}), \ldots, (a_{i_{n-1}+1}, \ldots, a_{n-1})]$a parallel-segmented sum function produces an output n-dimensional sequence w_s, where

$$w_s = [(a_0, a_0 * a_1, \ldots, a_0 * \ldots * a_{i_1}), (a_{i_1+1}, a_{i_1+1} * a_{i_1+2}, \ldots, a_{i_1+1} * \ldots * a_{i_2}), \ldots, (a_{i_{n-1}+1}, a_{i_{n-1}+1} * a_{i_{n-1}+2}, \ldots, a_{i_{n-1}+1} * \ldots * a_{n-1})] \quad (1)$$

For example, considering the input vector $v = [(10,3,2), (1,3,4,9), (5,0), (11)]$, if one choses the associative binary summation operator, after computing theparallel-segmented sum function, one obtains the output vector $w_s = [(10,13,15), (1,4,8,17), (5,5), (11)]$.

The parallel-segmented sum offers the same amount of parallelism as the parallel prefixed (unsegmented) sum, but it processes different partitions of the input vector. Thus, this function is extremely useful as itallocates different processing tasks to uniform execution structures, such as the CUDA execution thread blocks. This situation occurs frequently in sorting algorithms as "the quick-sort algorithm" or in sparse matrix multiplication applications. The usual representation of those segmented vectors is realised using a combination of a sequence of values and a vector containing the associated segments' flags that specifies how the input vector is partitioned. For example, I have chosen the representation that stores a "1" for each element from the start of the vector and a "0" for the rest of the partitions' elements. In the literature, this representation is considered to be the most suitable for massively parallel computing systems [2].

In the above depicted case, the sequence of values is $v.val = [10,3,2,1,3,4,9,5,0,11]$ and the vector containing the associated segments' flags is $v.seg = [1,0,0,1,0,0,0,1,0,1]$. In [4] is depicted a way for implementing the parallel-segmented sum function as a parallel prefixed (unsegmented) sum by transforming the associative binary operator $*$ in a new one, denoted by $*^*$, defined on the set of all pairs consisting of the values and their corresponding flags, denoted (d, v):

$$(d_1, v_1) *^* (d_2, v_2) = (d_1 | d_2, if\ d_2 = 1\ then\ one\ choses\ v_2,\ else\ it\ is\ computed\ v_1 * v_2) \quad (2)$$

I have designed the parallel-segmented sum algorithmic function, as to obtain a self-adjustable and self-configurable processing solution (regarding the number of thread blocks, the number of threads within a block and the number of processed elements per thread), depending on the graphic processor's architecture. Within the experimental tests, I have used three different graphic processing units from the CUDA architecture: the Tesla GT200 architecture, launched on 16 Jun 2008; the Fermi GF100 architecture, launched on 26 March 2010 and the KeplerGK104 architecture, released on 22 March 2012. I have designed the parallel-segmented sum algorithmic function as to offer, when executed, a high level of performance, by allocating the appropriate amount of resourcesfor each GPU. I have chosen a block size of 256 threads for the GTX 280 from the GT200 architecture; a block size of 512 threads for the GTX 480 from the Fermi GF100 architecture and a block size of 512 threads for the GTX 680 from the KeplerGK104 architecture. Each execution thread processes 8 elements in the case of the GTX 280 GPU, 16 elements in the case of the GTX 480 GPU and 32 elements in the case of the GTX 680 GPU.

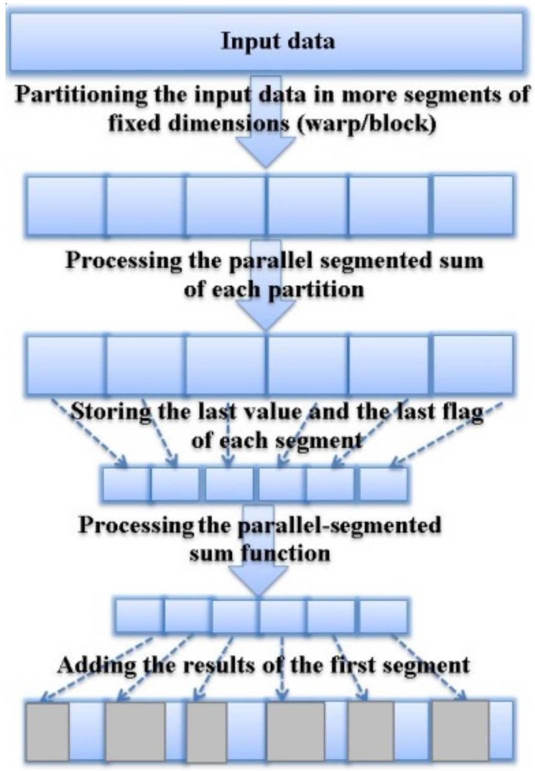
Input data
Partitioning the input data in more segments of fixed dimensions (warp/block)
Processing the parallel segmented sum of each partition
Storing the last value and the last flag of each segment
Processing the parallel-segmented sum function
Adding the results of the first segment

Fig. 1. The parallel-segmented sum algorithmic function at the block and global level

In order to implement an efficient parallel-segmented sum algorithmic function, I have first designed the algorithmic function within a warp, then I have called it within the block level and finally I have designed the parallel-segmented sum algorithmic function at the global level. At the warp level, it is first computed the current thread's index within the warp and depending on this index it is processed the element of the output vector combining the values of the input vector and the vector containing the associated segments' flags that specifies how the input vector is partitioned. The obtained result is further processed at the block level and at the global level (**Fig. 1**).

In the following, I present the main optimization solutions that I have developed and applied for improving the performance of the parallel-segmented sum algorithmic function.

3 Solutions for optimizing the performance of theparallel-segmented sum algorithmic function in CUDA

In order to improve the performance of the parallel-segmented sum algorithmic function in CUDA, I have developed and applied a series of optimization solutions:

Solution 1 - optimizing the allocated tasks of each thread: processing a single element per thread does not generate an enough computational load that reduces the memory latency. This is the reason why I have allocated 8/16/32 input elements to each thread of the GTX 280/480/680 GPU.

Solution 2 – reducing the number of used registers: the available number of execution threads is often limited by their registers requirements. Minimizing the number of used registers is very important in the case of the parallel-segmented sum algorithmic function, as this function requires a large number of registers in order to store and handle the segments' flags. This solution is also useful for optimizing web applications readability[5].

Solution 3 – using the warp shuffle operation in order to save shared memory: this solution can be implemented only for the GTX 680 graphic processing unit from the Kepler architecture, as the warp shuffle operation is supported on devices having the compute capability 3.x. Using this technique, threads within the same warp exchange data between them, without having to use the shared memory, thus reducing the memory latency.

Solution 4–synchronizing the parallel execution tasks: the synchronization of the parallel tasks represents their real time coordination and is often associated with intra-thread communication. I have used this solution for the Tesla GT200 architecture, and the Fermi GF100 architecture, while for the Kepler architecture I have used the **Solution 3**. The warp level function significantly reduces the necessity to synchronize data. The synchronizationis necessary only when sharing data between threads from different warps. In order to process a segmented-sum, the warps write their last element in a shared-memory vector and afterwards a single warp sums all these elements. In the end, every processing thread adds the warp's sum to the first step sum.

Solution 5 – reducing the number of parallel processing steps: although this determined initially an increase of the computational load, the SIMT (Single Instruction Multiple Threads) warp execution model facilitated the processing.

Solution 6 – using multiple execution thread blocks: this leads to a significant improvement of the execution time when processing the parallel-segmented sum algorithmic function in CUDA, compared to its sequential implementations, run on CPU.

Solution 7 – managing shared memory bank conflicts: the shared memory used by the parallel-segmented sum algorithmic function in CUDA, is composed from many memory banks. When multiple data requests originate from the same memory bank, memory bank conflicts occur. In order to eliminate these conflicts, if all the execution threads use the same memory address, a complex triggering mechanism is activated and thus data is simultaneously delivered to multiple execution threads.

Solution 8 – partitioning data in warp-sized fragments and then processing them independently, using one warp per each of them. This kind of solution proves very useful when optimizing security solutions implemented by the use of the multi-layered structural data like the MSDSSA presented in [6].

Solution 9 – the balanced loading of the necessary computational volume. The optimized parallel-segmented sum algorithmic function provides a balanced computational load for each of the input vectors' partitions, even if they have different dimensions (in the next section, in the experimental tests, the input vectors have been randomly segmented). No matter how the input vector has been partitioned, the tasks' parallelization determines the obtaining of an increased level of performance compared to the sequential implementations of the segmented sum algorithmic function.

In the next section, I have developed and implemented a benchmark suite in order to analyse the software performance of the parallel-segmented sum algorithmic function in CUDA, that has been optimized using the **Solutions 1-9**.

4The experimental results and the performance analysis for the parallel-segmented sum algorithmic function

In the following, I analyse the software performance of the parallel-segmented sum algorithmic function described above and compare it with an alternative sequential approach run on the CPU. In the benchmark suite, I have used the following configuration: Intel i7-2600K operating at 3.4 GHz with 8 GB (2x4GB) of 1333 MHz, DDR3 dual channel. I have used the Windows 8 Pro 64-bit operating system. I have analysed how the main technical features of the used graphic cards influence the experimental results. In order to program and access the GPUs, I have used the CUDA Toolkit 5.0 and the NVIDIA developer driver version 306.97. Moreover, in order to reduce the external traffic to the Graphics Processing Unit, all the processes related to the graphical user interface have been disabled. When I ran the tests on the CPU, I have used only the integrated Intel HD Graphics 3000 graphics core from the CPU and in the system, no discrete graphic cardwas installed.

In the first series of experimental tests, I have evaluated the execution times obtained by applying theparallel-segmented sum algorithmic function on various sized vectors, containing float type elements, using the summation as binary operator. The parallel-segmented sum algorithmic function has been run on the three graphic cards and on the CPU mentioned above. The results represent the average of 10,000 test iterations. The vectors' elements and the segments' dimensions have been randomly generated, but the designed parallel-segmented sum algorithmic function allows the specification of the input vector (the elements and their flags). I have highlighted the execution times and the memory bandwidth corresponding to each of the input vectors and and each of the processing units.

After having highlighted the execution timecorresponding to each of the input vectors for each of the processing units, I have computed the Total Execution Time (TET) for the 10,000 iterations corresponding to all the 26 vectors of different dimensions. I have used an energy meter device (Voltcraft Energy Logger 4000) to measure the Power Consumption PC (kW) and afterwards I have computed the Energy Consumption EC (kWh) for each processing unit. When the benchmark suite is run on the GTX 280 GPU, the system consumes 10 times less energy than when using the i7-2600K CPU; on the GTX 480, the system consumes 15 times less energy than on the CPU; when the GTX 680 is used, the system consumes 68 times less energy than when using the CPU [7].

By analysing the obtained results, I have noticed that the parallel-segmented sum algorithmic function developed in CUDA and run on the GPUs offers a high degree of performance (highlighted by the reduced execution times) and considerable economic advantages due to the reduced energy consumption, surpassing the results obtained when using the sequential approach, run on the CPU **(Table 1).**

Table 1. Synthetic experimental results

Experimental results	**The processing unit**			
	CPU	**GTX 280**	**GTX 480**	**GTX 680**
TET (h)	2.426	0.156	0.087	0.024
PC (kW)	0.198	0.306	0.358	0.307
EC (kWh)	0.480	0.048	0.031	0.007
The consumption on the GPUvs on the CPU		10 x lower	15 x lower	68 x lower

By analysing the experimental results obtained for the parallel-segmented sum algorithmic function when the input vector's dimension is 35-1030 elements, I have noticed that the best results (lower execution time, higher bandwidth) are those registered by the CPU. This happens becausethe huge parallel processing capacity of the GPUs has not been used at its entire potential as it has not been generated an enough computational load (**Fig. 2, Fig. 3**).

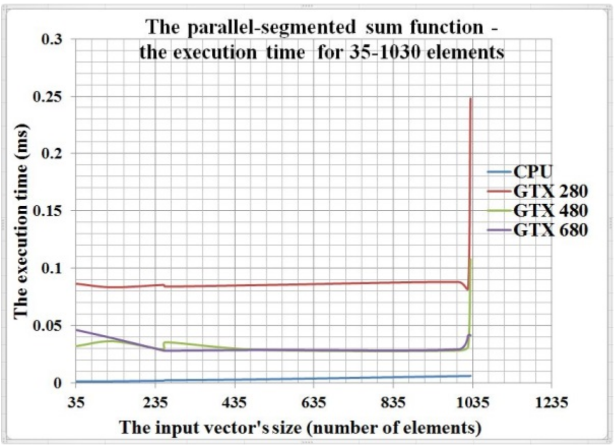

Fig. 2. The execution time for 35-1030 elements of the input vector

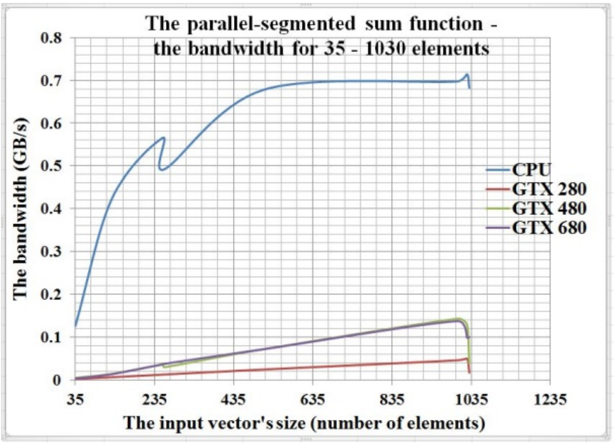

Fig. 3. The memory bandwidth for 35-1030 elements of the input vector

I have analysed the obtained experimental results when processing vectors of32768-60000000 elements and I have noticed that in this case the best results (lower execution time, higher bandwidth) are obtained when running the parallel-segmented sum algorithmic function on the GPUs: the GTX 680, then on the GTX 480 and on the GTX 280(**Fig. 4, Fig. 5**).

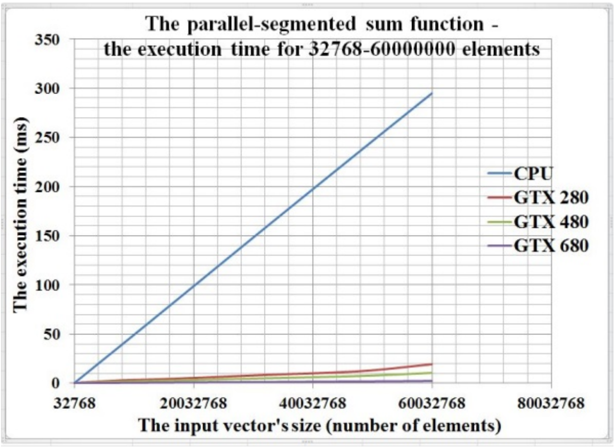

Fig. 4. The execution time for 32768-60000000 elements of the input vector

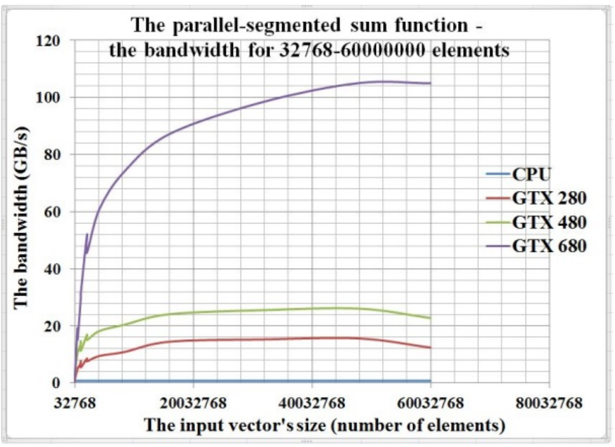

Fig. 5. The memory bandwidth for 32768-60000000 elements of the input vector

This time, the GPUs were able to use their huge parallel processing capacity, because it has been processed a sufficient computational load.

Afterwards, I have analysed the impact of the data type on the parallel-segmented sum algorithmic function's performance. I have designed the function as to allow the specification of the input vector's elements data type, that can be of integer, unsigned integer, float, double, long long or unsigned long long data type. In this case, I have benchmarked the performance on the GTX 680 processor and I have used the same testing methodology as in the previous experimental tests, highlighting the variation of the execution time depending on the input vector's size for different data type elements. I have recorded similar levels of performance when the input data type is integer, unsigned integer or float, the execution time ranging from 0.028399 ms to 2.319234 ms(**Fig. 6**).

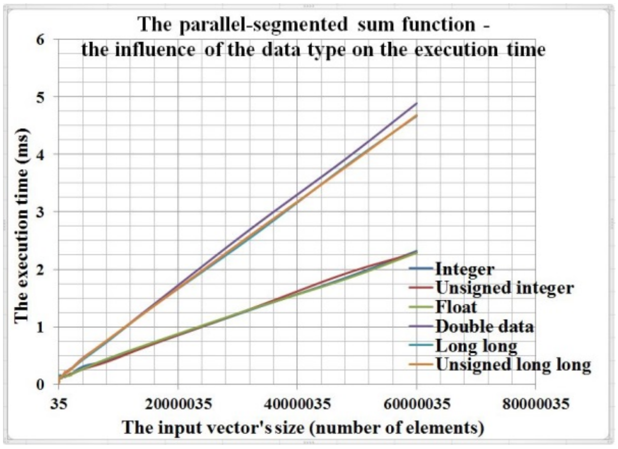

Fig. 6. The impact of the data type– the execution time

I have benchmarked the variation of the bandwidth for different input vector's sizes of different data types (**Fig. 7**). The parallel-segmented sum algorithmic function offers a high level of performance no matter what the data type is (integer, unsigned integer, float, double, long long or unsigned long), the results being comparable in all the 6 analysed cases.

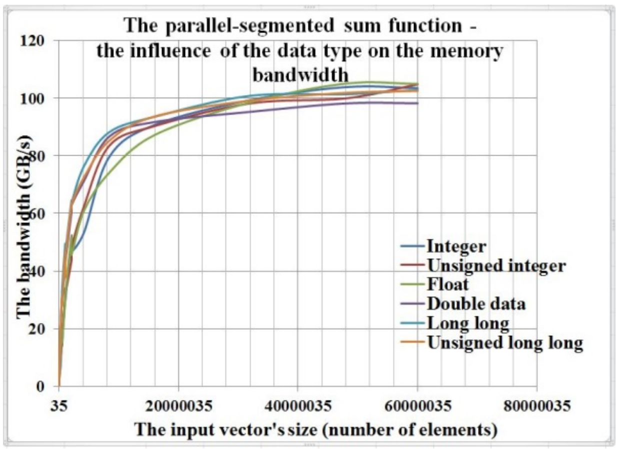

Fig.7. The impact of the data type – the bandwidth

I have also analysed the performance impact of the associative binary operator (as the function allows to select the binary operator that can be summation, maximum, minimum or multiplication). I have concluded that the performance is similar for all the studied binary operators.

5Conclusions

The optimization solutions of the parallel-segmented sum algorithmic function offer a high level of performance on a wide range of CUDA enabled graphics processors, covering different scenarios and applications, without being affected by the processed data type or binary operator. The Compute Unified Device Architecture offers a tremendous potential for optimizing data-parallel problems, overcomingmost of the limitations posed by traditional central processing units.

References

[1] J.Sanders, E. Kandrot, *CUDA by Example: An Introduction to General-Purpose GPU Programming*, Addison-Wesley Professional, 2010.

[2] M. Harris, M. Garland, *Optimizing Parallel Prefix Operations for the Fermi Architecture, GPU Computing Gems Jade Edition,* Morgan Kaufmann, 2011.

[3] I. Lungu, A. Pîrjan, D.M. Petroşanu, Solutions for optimizing the data parallel prefix sum algorithm using the Compute Unified Device Architecture, *Journal of Information Systems & Operations Management*, Vol. 5, No. 2.1, 2011, pp. 465-477.

[4] J. T. Schwartz, Ultracomputers, *ACM Transactions on Programming Languages and Systems*, Vol. 2, No.4, 1980, pp. 484-521.

[5] G. Garais, Web Applications Readability, *Journal of Information Systems & Operations Management*, Vol. 5, No. 1, 2011, pp. 114-120.

[6] A. Tăbu□că, A new security solution implemented by the use of the multilayered structural data sectors switching algorithm (MSDSSA), *Journal of Information Systems & Operations Management*, Vol.4, No.2, 2010, pp. 164-168.

[7] D. M. Petroşanu, A.Pîrjan, Economic considerations regarding the opportunity of optimizing data processing using graphics processing units, *Journal of Information Systems & Operations Management*, Vol. 6, No. 1, 2012, pp. 204-215.