A MIXED APPROACH TOWARDS IMPROVING SOFTWARE PERFORMANCE OF COMPUTE UNIFIED DEVICE ARCHITECTURE APPLICATIONS

Alexandru Pîrjan [1*]

ABSTRACT

One of the most important aspects when developing applications that leverage the powerful parallel processing power of graphics processing units (GPUs) that offer support for the Compute Unified Device Architecture (CUDA) is to divide the tasks that are to be processed appropriately into sections that can be processed either serially or in parallel. The paper analyses this important developing issue by proposing a mixed approach to programming efficient software applications. The article makes an in depth analysis of the key aspects that carry significant weight when deciding to parallelize a certain part of an application: the analysis phase of the application that is about to be parallelized; the amount of time involved to achieve the implementation; the feasibility of parallelizing the source code; situations when one should aim for central processing units optimization techniques that yield better performance on sequential source code rather than parallelizing the whole algorithm.

KEYWORDS: *mixed approach, software performance, CUDA, Parallel Nsight, CUDA Profiler.*

1. INTRODUCTION

In recent years, there has been a great interest in the literature for developing optimization solutions using the parallel processing power of graphics processing units [1], [2], [3]. The optimization of data processing is a field of great interest due to its numerous applications: in developing intelligent systems [4], [5], in cryptography [6], in developing office applications [7], in electronic payment systems [8], [9], in developing web applications [10], [11].

The Compute Unified Device Architecture, implemented in the latest graphics processing units, offers developers the opportunity to use an extension to the traditional C language that makes it possible to write the source code in the familiar C programming language. When developing a CUDA application, the programmer must write source code for both the central processing unit (host) and for the graphics processing unit (device).

The central processing unit (CPU) calls a large number of kernel functions on the device that will use its internal task scheduler to establish the most suitable strategy of allocating the functions (kernels). As long as there is a sufficient amount of parallelism in the task that is being processed, the processing time of the program should improve as the number of graphics processing unit's Streaming Multiprocessors (SMs) increases.

[1]* corresponding author, Lecturer PhD, Faculty of Computer Science for Business Management, Romanian-American University, 1B, Expozitiei Blvd., district 1, code 012101, Bucharest, Romania, alex@pirjan.com

The performance improvement of a CUDA software application should take into consideration a series of optimization aspects, such as: dividing the tasks appropriately into sections that can be afterwards processed either serially or in parallel; an appropriate managing of the graphics processing units' memory and their associated data transfers between the device (GPU) and the host (CPU); understanding in detail the CUDA threading mechanism and the hardware's computational capability, that determines the GPU's processing features; having an insight of the ready-optimized algorithms; identifying the weak points that hold down the performance of the application, their causes and solutions for minimizing their impact; having in mind to always develop reusable and scalable code, that can be easily maintained and upgraded to support future versions of the GPUs.

The first of the above mentioned optimization aspects, dividing the tasks appropriately into sections that can be afterwards processed either serially or in parallel, is essential and its analysis as a software performance improvement method represents the purpose of this paper. Therefore, in the following it is analyzed a mixed approach in order to optimize software applications by means of parallelization in the Compute Unified Device Architecture.

In this paper it has been made an in depth analysis of the key aspects that carry significant weight when deciding to parallelize a certain part of an application: the analysis phase of the application that is about to be parallelized; the amount of time involved to achieve the implementation; the feasibility of parallelizing the source code; situations when one should aim for central processing units optimization techniques that yield better performance on sequential source code rather than parallelizing the whole algorithm.

2. DIVIDING THE DATA AND TASKS IN VIEW OF PARALLEL PROCESSING IN CUDA

The first thing one should have in mind when improving the software performance of an application by means of parallelization in CUDA, is how much of the code is suitable to be processed in parallel. The quantity of source code that can be processed solely sequentially limits the improvement in software performance that can be achieved by parallelization.

For that reason, the first thing that the developer must do, is to take into consideration whether there is a sufficiently great number of tasks that can benefit from the parallelization process, so he must be certain that parallelization is indeed the best solution. Of central importance for the developer is to identify if the problem he wants to solve can be divided into more parts that can be processed in parallel.

If there is no way to expose concurrency by dividing the task into several ones, the developer should instead focus on central processing units optimization techniques that yield better performance on sequential source code rather than parallelizing the whole algorithm. There are many optimization techniques that, after having been applied, make the sequential code perform in a satisfactory way. The graphics processing units can invoke thousands of threads, so if one wants to benefit the most from the huge parallel processing power of the CUDA GPUs, he should be able to divide the task into thousands of simultaneously ones.

One should always begin to analyze initially the data and afterwards the tasks that must be processed. One should always have in mind the final obtained data set and reflect whether there exist one or more transformation rules that represent the output data in a certain point through the input one, in the same point. In the case in which such transformation is possible, it is feasible to parallelize the problem, using the CUDA architecture.

The programmer must correctly identify the approach that is most suitable for optimizing his application: a whole sequential approach, an exclusive parallel approach or a mix between the sequential and the parallel approaches. For most of the problems, a parallelization solution that produces the best results can be identified and applied. The problem actually lies in the fact that many developers are accustomed to single-threaded applications and they fail to seek and identify the possibility of parallelizing the source code.

There are certain situations in which the sequential approach is entirely unpractical, like the H264 video encoding [12]. Regarding this problem, it comprises a number of stages and each of them produces an output data stream varying in length. Every destination pixel depends on n of the source pixels. A various number of scientific problems exert the same particularities. There are some situations when the input data set is huge and a pre-filtering might be applied to it. As a result, those data points whose impact on the output data is small, are removed from the data set. This process will indeed lead to a small increase of error, but in certain cases the advantages outweigh the disadvantages, as long as the final error stays below a threshold.

Up till now, the optimization process focused solely on the operations that were being applied to the data. Nowadays, the focus is on the data, as the processing power of GPUs has grown dramatically in contrast with the memory bandwidth. In order to efficiently use the memory bandwidth of the graphics processing unit that is up to 10 times the one of the central processing unit, one must properly divide the problem, as to make use of the increased bandwidth.

Particular attention must be payed when dividing the data to more graphics processing units linked together, as the traffic between the different graphics processing units will have a negative impact on the computation cycles therefore affecting the total processing time of the application. Consequently, this penalty must be minimized and compensated through the computational volume and processing power.

3. SOLVING THE DEPENDENCY PROBLEM

There are situations when the current computation needs the output of a previous one, thus arising a dependency problem when a certain variable cannot be computed without knowing beforehand the value of another variable, or a value from a previous step. In some cases, for computing the value of one variable one needs the output resulting from computing one or several variables, while in other cases the result of a computation from a previous step is needed in order to perform the current step. This dependency problem impedes the possibility of parallelizing the application or a certain step of an algorithm. In this situation, it is only the current step that can be the parallelized.

```
void example_kernel(void)
{
  x=3*z + 1024;
  y=7*z - 512;
  t=2*x + 5*y;
}
```

Figure 1. An example of a dependency situation occurring within a kernel

In **Figure 1** is presented an example of dependency situations that may occur within a kernel function. Thus, in this case, the value of the x and y variables cannot be computed without knowing the value of the z variable, so one can say that x and y depend on the variable z. After having computed the value of the variable z, one can parallelize the computation of x and y.

One can also remark that the variable t depends on the variables x and y. Therefore, the value of the variable t cannot be computed without knowing the values of the before mentioned variables. The computation of this variable cannot be parallelized with the computation of x and y, it must be executed in a subsequent step.

In the above mentioned example, parallelism cannot be employed efficiently and both the central processing unit and the graphics processing unit use multiple threads to compensate for this drawback.

In what concerns the central processing units that support the hyperthreading feature, there can be employed other virtual central processing unit cores to compensate for the idle time. The central processing unit must manage precisely the allocation of the instructions to different threads. The graphics processing unit makes use of multiple threads that are switched alternatively, at different moments of time, as to compensate most or even the whole latency that has been induced by the needed computations.

One solution to facilitate the parallel execution of the above mentioned kernel code is to use an overlapping independent instructions technique [12]. In this case, one can insert new variables and several independent instructions between the computations, thus granting the computations an additional time to compute before obtaining the output, but this additional time will be compensated by being able to process in parallel the independent instructions that contain intermediary results of the variable.

Practically, this technique compensates the latency resulting from the arithmetic computations, by means of instructions level parallelism. This mechanism makes it possible for the compiler or for the processing unit not only to overlap the processing of a set of computation instructions, but also to modify the instructions' execution order.

Another method used for solving dependency issues is the loop fusion technique. As one can see in **Figure 2**, the number of steps necessary to compute the kernel function *compute_1* exceeds the number of steps necessary for the kernel function *compute_2*. If we analyze the kernels closely we can notice that a considerable part of the first loop's iterations from the kernel function *compute_1* overlaps the iterations of the second loop of the same kernel.

```
extern int x, y, z, t;
void compute_1(void)
{
   unsigned int i,j;

   x = 0;
   for (i=0; i<150; i++)
   {
     x+= 3*z + i;
   }

   y = 0;
   for (j=0; j<300; j++)
   {
     y+= 5*t + j;
   }
}
```

```
extern int x, y, z, t;
void compute_2 (void)
{
   unsigned int i;

   x = 0;
   y = 0;
   for (i=0; i<150; i++)
   {
     x += 3*z + i;
     y+= 5*t + j;
   }
   for (i=150; i<300; i++)
   {
     y+= 5*t + j;
   }
}
```

Figure 2. An example of a loop fusion technique that inserts independent instructions and obtains a decrease in the number of instructions that have to be processed

As a consequence, one can allocate the part of iterations that overlap to the first loop of the kernel, thus obtaining the optimized version *compute_2* kernel (**Figure 2)**, therefore introducing independent instructions while obtaining a decrease in the overall number of loop iterations, consequently cutting down the number of instructions that have to be computed.

When performing the computation on a last generation graphics processing unit, one can obtain a higher increase of performance by unrolling the two loops and allocating the individual computations to different threads thus using only a kernel. A possible solution consists in allocating two blocks of threads, the first block containing a number of execution threads equal with the number of iterations of the first loop while the second block contains a number of threads equal with the number of iterations of the second loop.

However, this approach can sometimes pose problems due to a decrease in the total amount of parallelism exposed to the scheduling process of blocks and threads. If the level of parallelism in an application is minor, the execution time will suffer a penalty due to this approach. Moreover, the kernels spend more registry memory when they are fused, therefore the number of possible blocks that can be invoked will be reduced.

One must take into account practical situations that arise when developing production applications in which more passes are needed to solve the problem. In order to achieve more passes, one needs to implement several sequential calls to the same kernel. If after every iteration the kernel needs to access the global data for storing and retrieving the data, the whole software performance of the application will suffer due to the high latency of this type of memory.

One solution consists in refitting the data allocated to each of the kernel functions, reducing the amount of processed data to an extent where it is possible to use shared and registry memory types that offer considerable faster access times that will improve substantially the overall software application performance.

4. A COMPARISON BETWEEN THE TECHNICAL OPTIONS OF MANAGING THE DATASET ON THE HOST AND THE DEVICE

When developing a software application, one must take into account the available memory resources, the different types of memory with their associated characteristics and the dimension of the data that the application must process. According to [12], the requirements of a central processing unit are different from those of a graphics processing unit as it is depicted in **Table 1**.

Table 1. A comparison between the amount of data that can be stored in different types of memory on a CPU and a GPU

How much data can be stored within a certain type of memory	Central Processing Unit	Graphics Processing Unit
L1 cache	16 - 32 KB	16 - 64 KB
L2 cache	256 KB -1 MB	512 KB - 4096 MB
L3 cache	512 KB - 16 MB	-
GPU memory	-	512 KB - 16 GB
host memory on one machine	1 - 128 GB	1 - 128 GB
host-persistent storage	500 GB - 20 TB	500 GB - 20 TB
distributed among many machines	more than 20 TB	more than 20 TB

If the data set has a small dimension, one can increase the processing speed on the central processing unit by using a CPU that has more cores. In this case, the increase of the processing speed overpasses a linear trend due to the fact that every core of the processor has a reduced amount of tasks to process. If the new amount of data that is smaller in dimension is processed in the L2 cache memory instead of the L3 cache memory, one can obtain a dramatic increase in performance because of the increased bandwidth that this type of memory offers. If one used the L1 cache instead of the L2 cache memory, the impact on performance would be even greater.

In the GPUs case, the most important aspect relies in the quantity of data that can be stored on a single card. Copying data between the host and the device consumes a lot of resources and affects the overall performance of the application. There are some graphic processing units that allow to copy the data to and from the device simultaneously, but in order to achieve this one must make use of the pinned memory feature. This type of memory cannot be substituted by the system's virtual memory, so one has to use the Dynamic Random-Access Memory of the system.

Nowadays most of the motherboards support at least two CUDA-enabled graphics processing units and a maximum of four cards, using the Scalable Link Interface (SLI) mode. If the requirements of a problem that must be solved involves a huge dataset, that cannot be stored in a single computer, one has to use the internode communication concept. Unfortunately, this implies a huge time penalty and advanced knowledge of different application programming interfaces.

5. DETECTING AND ISOLATING THE DEFICIENCIES IN DATA PROCESSING

When analyzing parallel architectures, a lot of the scientific papers refer to the Amdahl's law that is useful to forecast, from the theoretical point of view, the speedup that might be obtained if one uses more processing units.

If one denotes by S the above mentioned speedup of processing a whole task, from the theoretical point of view; by s the speedup factor that improves the execution performance of a certain part of a task, resulting from the enhancement of the system's available resources; by p the percentage from the processing time allocated to the part of the task that improved after the upgrade, measured before this enhancement, the Amdahl's law stipulates that:

$$S(s) = \frac{1}{1-p+p/s} \quad (1)$$

From the equation (1) one can easily deduce that $S(s) \leq \frac{1}{1-p}$ (as $p/s \geq 0$). Furthermore, $\lim_{s\to\infty} S(s) = \frac{1}{1-p}$. Analyzing these results, one can conclude that the improvement in the execution performance of the entire task, from the theoretical point of view, raises along with the enhancement of the system's available resources, but no matter how much the resources are enhanced, the improvement is in every situation impeded by the part of the task that has not improved after the upgrade.

In particular, in parallel computing, it is considered a case in which the execution of a software application requires 48 hours on a central processing unit that incorporates only one processing core. It is presumed that a certain part of the entire task (whose execution takes 8 hours) cannot benefit from a potential enhancement of the system's available resources (it is impossible to be executed in parallel), while for the remaining $p = 83\%$ of the entire task (whose execution takes initially the remaining 40 hours), the execution can be parallelized, being improved along with the system's available resources.

Applying the above mentioned theoretical results, one can state that no matter how much the resources are enhanced, the execution time cannot go beyond the threshold of 8 hours. The speedup of processing a whole task, from the theoretical point of view, $S(s)$ is always lower than or equal to $\frac{1}{1-p} = 5.88$, meaning that the maximum improvement in the execution time that can be achieved due to the parallelization of the algorithm is 5.88 times. Therefore, improving the software performance only by means of parallelization has certain limits and a mixed approach should be employed that targets both the parallel and the sequential aspects of optimization.

In order to acknowledge the deficiency points of an application, one must make use of the profiling process. Without applying this process, developers may find themselves in the situation that, after having spent a tremendous amount of time to fix a certain drawback they had thought was crucial in the application, they do not achieve the desired results. The development of a complex application usually implies more teams of professional programmers that are working together on different parts of the application. It is entirely possible that one programmer could think that he has identified a serious drawback in the application only to realize later that it was not the case.

Therefore, it is very important to use specific profiling tools like those offered by the NVIDIA Company: the CUDA Profiler and the Parallel Nsight that provide a detailed and comprehensible profiling analysis of a developed application. These tools are identifying and analyzing the hardware counters, the deficiency points where processing time is wasted and determine the total utilization of the graphics processing unit's resources. Using the CUDA Memcheck tool, one can efficiently check if the memory bandwidth is being used efficiently.

The CUDA Toolkit version 7.5 brings a very useful new feature to the NVIDIA Visual Profiler, namely the ILP (Instruction-Level Profiling) support. This option can be used on graphics processing units from the Maxwell and Pascal architectures and makes it easier to identify drawbacks in the source code, to identify performance related issues. One can even isolate certain lines from the source code along with assembly instructions that can affect the performance negatively. The latest version of Visual Profiler can also generate a pie chart that depicts the warp state distribution (**Figure 3**).

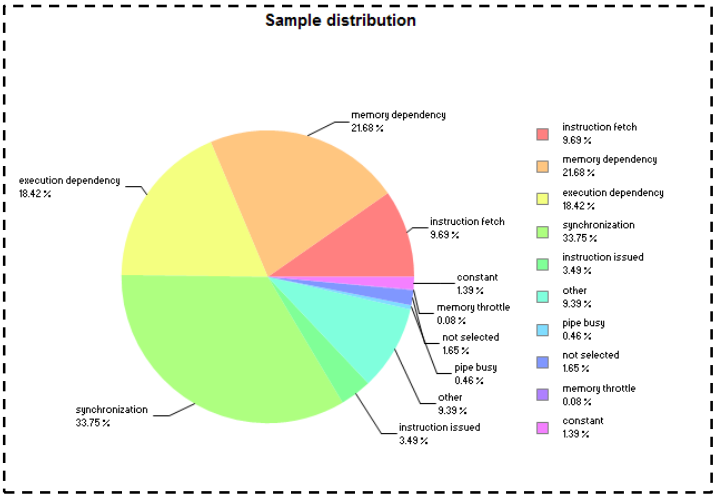

Figure 3. A pie chart generated using Visual Profiler depicting the warp state distribution [1]

When profiling an application one should pay particular attention at the part where the source code consumes most of the total processing time. After all the problems signaled by the profiling process have been resolved, dramatic improvements in the performance of the application cannot be achieved unless the whole application is redesigned from scratch.

Regarding this aspect, the Parallel Nsight tool from NVIDIA offers the possibility to run several testing scenarios, in order to identify important information in the source code, like: identifying the divergent threads or the code branches; important statistics regarding the use of the different types of memory, like the cache memory and the obtained

[1] The pie chart has been downloaded from the official NVIDIA documentation site http://devblogs.nvidia.com/parallelforall/wp-content/uploads/2015/09/Figure5_samplingPieChartRawCode.png , accessed on 09.27.2016, at 22:00

bandwidth in different types of memory; statistics regarding the main reasons for stalling; the efficiency in issuing instructions; the attained FLoating-point Operations Per Second (FLOPS)[1].

6. PREPARING AND ALLOCATING THE TASKS TO THE CENTRAL AND TO THE GRAPHICS PROCESSING UNITS

The applications that offer the best performance per Watt are the ones that are developed based on a mixed approach: benefitting from the strong points of both the central processing unit and the graphics processing one, allocating the data appropriately. The central processing unit must not be ignored even if one transfers most of the computational burden to the graphics processing unit.

When developing production applications, other restrictions might come into play, such as the network, the input/output bandwidth of the system. Nowadays, computers have a lot of memory available, thus most of the input/output operations are cached. As a consequence, these operations target to a great extent the movement of data in the memory rather than its movement between different devices. It is also possible to instantiate a different graphics processing unit context using idle central processing units' resources. The newly created kernel functions are inserted in the waiting queue of the graphics processing unit and are about to be executed when the necessary resources are available.

One must take into account that there is a great amount of idle time both on the central processing unit and on the graphics processing one. According to [12], the idle time of the CPU is usually cheaper, while the idle time of the GPU is up to 10 times more valuable compared to the one of the CPU. The Parallel Nsight tool from NVIDIA has the possibility to generate a special timeline graphic that will depict the amount of idle time and which kernel functions have generated it. When programmers invoke more kernels on a distinct graphics processing unit, the kernels will occupy the idle resources. As a consequence, the initial kernels that were invoked have a slightly increased latency than the others, but the global performance of the software application increases significantly.

There are many software applications in which a margin of 25%-30% idle time is completely acceptable. Usually, a software application starts by loading the data from a data source that operates or is designated to operate at low speeds. Once loaded, the data is transferred to the graphics processing unit waiting to be processed. After it has been processed by the CUDA cores, the results are copied back to the central processing unit, which usually stores it to the initial low speed data source and loads the following data block that is to be processed, the process thus continues until all the data blocks have been processed.

By using CUDA streams, the graphics processing unit can be programmed to load the following block of data from the low speed data source while the current one is being processed. The whole process is achieved by instantiating several processes: the low

[1] https://developer.nvidia.com/parallel-nsight-21-new-features, accessed on 09.27.2016, at 23:00.

speed data source halts the second process while loading the data from the first one. While the first process is working on the first block of data, the second one accesses the low speed data source and loads the second block of data, waiting afterwards idle to be invoked after the first process has finalized the processing. While the first process copies the results to the host, the second process begins executing the kernel function. This technique of using CUDA streams and multiple processes makes it possible to efficiently extend over the processing times of the host and the device in a manner that covers partly some of the idle times.

Another method for accomplishing a similar outcome consists in the use of CPU threads in order to make available a certain portion of data jointly with other processes, thus achieving the synchronization process at a higher speed. A technique called processor affinity (central processing unit pinning) allows a certain thread belonging to a core to be linked only to that particular core. This technique can frequently increase the processing performance as it gives the opportunity to reuse more efficiently the available cache memory of the respective core. The decision to put into practice this approach must be weighed against the required level of synchronization that a programmer wants to achieve on the central processing unit.

When tailoring the mixed approach, the aspect of paramount importance is given by the manner in which the tasks are divided between the host and the device. If the data set is thinly dispersed or scattered or it has a very reduced size, the central processing unit offers the best solution as such data sets types are best processed sequentially. Nonetheless, as the graphics processing unit offers a huge amount of processing power, the developer must pay particular attention not to keep the graphics processing unit idle when programming the application, waiting for the CPU to finish its processing.

A common method is to make use of the central processing unit only in the transfer part of the tasks. This approach however can sometimes have the drawback of oversaturating a single central processing unit's core, so the programmer must profile the application carefully and identify how much time the graphics processing unit really needs to perform its tasks [13], [14]. A successful technique that provides very good results consists in using the central processing unit in the last steps of an algorithm's computation when the level of parallelism has decreased dramatically and the workload is not sufficient for the GPU to harness its huge parallel processing power.

7. CONCLUSIONS

Dividing the tasks that are to be processed appropriately into segments that are suitable for parallel processing and segments that are better processed serially is the key to obtaining a high level of performance when developing CUDA Applications. This paper proposes the use of a mixed approach that does not neglect the role of the central processing unit nor the one of the graphics processing one. The mixed approach is fundamental if a programmer is to obtain a powerful software application from the performance and efficiency point of view.

When implementing the mixed approach, the developer must carefully divide the data and tasks in view of parallel processing on the Compute Unified Device Architecture enabled

graphics processing units and in view of sequential processing on the central processing ones. The programmer must be aware of any dependency problems that may arise within the source code and solve them promptly without neglecting the total amount of time that is needed in order to comply with the project's timeframe.

The developing of the application must take into account the available technical options of managing appropriately the dataset's dimension on both the host and the device and the identification of certain deficiencies in data processing using specialized profiling tools like the CUDA Profiler and the Parallel Nsight. After having taken into account thoroughly the analyzed technical aspects, the mixed approach can be successfully implemented and will lead to significant improvements to both the software performance of the application and its economic efficiency.

ACKNOWLEDGEMENTS

This paper presents results of the research project: Intelligent system for predicting, analyzing and monitoring performance indicators and business processes in the field of renewable energies (SIPAMER), research project, PNII – Collaborative Projects, PCCA 2013, code 0996, no. 49/2014 funded by NASR.

REFERENCES

[1] Lungu Ion, Pîrjan Alexandru, Petroşanu Dana-Mihaela, Optimizing the Computation of Eigenvalues Using Graphics Processing Units, University Politehnica of Bucharest, Scientific Bulletin, Series A, Applied Mathematics and Physics, Vol. 74, Number 3/2012, pp.21-36, ISSN 1223-7027.

[2] Pîrjan Alexandru, Optimization Techniques for Data Sorting Algorithms, The 22nd International DAAAM Symposium, Annals of DAAAM for 2011 & Proceedings of the 22nd International DAAAM Symposium, Vienna, 2011, pp. 1065-1066, ISSN 1726-9679, ISBN 978-3-901509-73-5.

[3] Petroşanu Dana-Mihaela, Pîrjan Alexandru, Economic considerations regarding the opportunity of optimizing data processing using graphics processing units, JISOM, Vol. 6, Nr. 1/2012, pp. 204-215, ISSN 1843-4711.

[4] Lungu Ion, Căruțașu George, Pîrjan Alexandru, Oprea Simona-Vasilica, Bâra Adela, A Two-step Forecasting Solution and Upscaling Technique for Small Size Wind Farms located in Hilly Areas of Romania, Studies in Informatics and Control, Vol. 25, No. 1/2016, pp. 77-86, ISSN 1220-1766.

[5] Lungu Ion, Bâra Adela, Căruțașu George, Pîrjan Alexandru, Oprea Simona-Vasilica, Prediction intelligent system in the field of renewable energies through neural networks, Journal of Economic Computation and Economic Cybernetics Studies and Research, Vol. 50, No. 1/2016, pp. 85-102, ISSN online 1842– 3264, ISSN print 0424 – 267X.

[6] Tăbușcă Alexandru, A new security solution implemented by the use of the multilayered structural data sectors switching algorithm (MSDSSA), JISOM, Vol.4, No.2 – December 2010, ISSN 1843-4711, Universitary Publishing House, pages 164-168.

[7] Pîrjan Alexandru, Petroşanu Dana-Mihaela, Solutions for developing and extending rich graphical user interfaces for Office applications, JISOM, Vol. 9, Nr. 1/2015, pp. 157-167, ISSN 1843-4711.

[8] Pîrjan Alexandru, Petroşanu Dana-Mihaela, Dematerialized Monies – New Means of Payment, Romanian Economic and Business Review, Vol. 3 Nr. 2/2008, pp. 37-48, ISSN 1842 – 2497.

[9] Pîrjan Alexandru, Petroşanu Dana-Mihaela, A Comparison of the Most Popular Electronic Micropayment Systems, Romanian Economic and Business Review, Vol. 3, Nr. 4/2008, pp. 97-110, ISSN 1842–2497.

[10] Tăbușcă Alexandru, HTML5 - A new hope and a dream, JISOM; May2013, Vol. 7 Issue 1, p49, ISSN 1843-4711.

[11] Garais Gabriel Eugen, Security measures for open source website platforms, JISOM. May2016, Vol. 10 Issue 1, pages 175-185

[12] Cook Shane, CUDA Programming, 1st Edition, A Developer's Guide to Parallel Computing with GPUs, Morgan Kaufmann, 2012, ISBN :9780124159334.

[13] Lungu Ion, Petroşanu Dana-Mihaela, Pîrjan Alexandru, Optimization Solutions for Improving the Performance of the Parallel Reduction Algorithm Using Graphics Processing Units, Informatica Economică, Vol. 16 Nr. 3/2012, pp. 72-86, ISSN 1453-1305.

[14] Lungu Ion, Pîrjan Alexandru, Petroşanu Dana-Mihaela, Solutions for Optimizing The Data Parallel Prefix Sum Algorithm Using The Compute Unified Device Architecture, JISOM, Vol. 5, Nr. 2.1/2011, pp. 465-477, ISSN 1843-4711.