U.P.B. Sci. Bull., Series A, Vol. 77, Iss. 1, 2015 ISSN 1223-7027

NOVEL APPROACHES IN GENERATING RANDOM NUMBERS USING GRAPHICS PROCESSING UNIT

Alexandru PÎRJAN[1], Dana PETROŞANU[2]

In this paper, we have researched and developed optimization solutions for implementing the Sobol random number generator in the Compute Unified Device Architecture. We have conducted a thorough analysis, using a series of experimental tests, for studying the solutions' influence on the execution time, on the number of generated samples per second and on the energy consumption. Lately, in the literature, there has been a lot of interest for developing random number generators, but none of the works so far (to our best knowledge) has developed optimization solutions targeted towards the Kepler GK104 architecture, harnessing its novel technical features.

Keywords: Compute Unified Device Architecture, Kepler architecture, Sobol random number generator, warp shuffle instruction, thread blocks.

1. Introduction

The random number generation is an essential component in numerous applications, especially in simulation problems. In particular, a fast parallel number generation decisively influences the performance of the parallel Monte Carlo simulations that are frequently needed in countless scientific fields, like financial management, computational finance, engineering, computational science, telecommunications, applied statistics, physical sciences, medicine and computational biology. Traditionally, random number generators process most of the data in a sequential manner and therefore, their software implementations on central processing units often face significant computational limitations. The Compute Unified Device Architecture (CUDA) offers a potential solution to overcome all of these limitations by harnessing the huge parallel computational power of CUDA-enabled graphic processing units (GPUs).

We have used in our research two of the most powerful and recent graphics processing cards, from the latest CUDA-enabled architectures: the Fermi architecture GF100 (from this architecture we have used the GeForce GTX480 reference graphic card) and the Kepler GK104 architecture (in this case we have used the GeForce GTX680 reference graphic card). We have carefully taken into

[1] Romanian American University, Romania, e-mail: alex@pirjan.com
[2] Mathematics Dept., University POLITEHNICA of Bucharest, Romania, e-mail: danap@mathem.pub.ro

account with utmost importance all of these technical characteristics when developing and implementing our random number generator solution.

2. The Sobol random number generator algorithm

The Russian mathematician I. M. Sobol has introduced in 1967 the notion of Sobol sequences, an example of quasi-random, low-discrepancy sequences, obtained through a standard algorithm for generating uniform numbers in the unit hypercube [1].

Sobol's random number generator algorithm, useful in performing numerical integration in the unit hypercube, constructs a sequence within the hypercube, filling it regularly. In order to compute the integral, one has to approximate it using the average of the function values in these points. If we denote by $U^m = [0.1]^m$ the unit m-dimensional hypercube and $f: U^m \to \mathbb{R}$ an integrable function over U^m, the Sobol sequence $x_s \in U^m$ satisfies the equation:

$$\lim_{n\to\infty} \frac{\sum_{s=1}^{n} f(x_s)}{n} = \int_{U^m} f \tag{1}$$

where the left side limit is finite and the convergence must be as fast as possible.

Broadly, the Sobol random number generator algorithm is based on a set of variables that determines a state, denoted by X_n at the n-th step, linked with the next step's state by an equation of the type:

$$X_{n+1} = f_1(X_n) \tag{2}$$

that allows the determination of each step's state, starting from the initial value X_0. An output process generates a quasi-random (approximately uniformly distributed) sequence:

$$x_n = g(X_n). \tag{3}$$

The parallelization of this random number generator is achievable taking into account that a CUDA thread block generates a block of numbers and therefore it requires the usage of a powerful advanced leaping algorithm that facilitates bouncing at the block's start. In this way, the generator can bounce over a number of points using an algorithm of the type $X_{n+k} = f_k(X_n)$, having the complexity $O(\log k)$. In order to perform the bouncing, we have researched the possibilities and identified three viable solutions:

a) The simple bounce solution: starting from a specified point in the generator sequence, each CUDA thread performs a bounce and then it generates a segment of points. These bounces are executed so that the segments do not overlap and are adjacent.

b) The strided bounce solution: the n-th thread $(n \leq N)$ generates the points $n, n+N, n+2N$ and so on.

c) The hybrid bounce solution: first, at the thread block level, a large bounce is performed and afterwards, within a block, each thread executes the strided bounce.

The Sobol random number generator algorithm discussed in this paper uses an efficient bouncing method. We have implemented the simple bounce technique and it proved to be an efficient approach. However, we have obtained an even more efficient result using the hybrid bounce approach because, when writing the output data, this solution has the considerable advantage of simple memory coalescence.

The initial Sobol's algorithm for obtaining the sequence was later improved [2]. The main results that we have obtained in this paper are based on the usage of Gray code and lead us to the conclusion that if the sequence's points were being permuted, a recurrence relation can be identified. This relation allows the simple generation of the $(j+1)$-th point directly from the previous one, the j-th. Starting from this result, one can develop an efficient C algorithm that generates the Sobol sequences [3]. We have developed our algorithm by parallelizing and implementing the initial sequential C algorithm in the Compute Unified Device Architecture.

In order to generate Sobol sequences in the unit cube [3], one must first notice that a high dimension sequence is composed of more one-dimensional sequences and therefore it is sufficient to study a one dimensional Sobol sequence.

Usually, there are generated up to $2^{32} = 4{,}294{,}967{,}296$ points. In this case, we have defined the Sobol sequence using a set of 32-bit integers $w_r, 1 \leq r \leq 32$, called direction numbers. The sequence $y_1, y_2, \dots$ is defined by:

$$y_n = g_1 w_1 \oplus g_2 w_2 \oplus g_3 w_3 \oplus \dots = y_{n-1} \oplus w_{f(n-1)}, \tag{4}$$

where $y_0 = 0$. In the equation (4) we have used the following notations:

- $\oplus$ is the binary "or" operator (exclusive)
- the bits g_r are given by the binary expansion of the Gray code representation of n, namely $n \otimes \left({}^{n}\!/_{2}\right) = \cdots g_3 g_2 g_1$
- the function $f(n)$ returns the index of the rightmost zero bit in the binary expansion of n.

The Sobol sequence is obtained using the sequence y_n using the following relation:

$$x_n = 2^{-32} y_n \tag{5}$$

When generating multidimensional Sobol sequences, it is recommended to use different direction numbers for each of the dimensions and to choose these numbers carefully in order to maintain the multidimensional uniformity properties of the sequence [4].

After having analysed the equation (4), we have noticed that the first relation offers a method for bouncing to the point y_n as this relation gives the formula for direct computing y_n, while the second relation represents an algorithm for computing y_n starting from the value of y_{n+1}. In this second expression, we first consider the natural nonzero number n fixed and then we increase it with 8. If the bit pattern of n is $n = \cdots b_3 b_2 b_1$and we add 8 at n, these last three bits remain unchanged as $8_2 = 1000$. Computing $f(n+i)$ for each $i, 1 \leq i \leq 8$, one can observe that we obtain the result 1 four times, the result 2 twice, the result 3 once and a result bigger than 3 once. Taking into account the property of the exclusive or:

$$a \oplus b \oplus b = a, \tag{6}$$

we obtain:

$$y_{n+8} = y_n \oplus w_1 \oplus w_1 \oplus w_1 \oplus w_1 \oplus w_2 \oplus w_2 \oplus w_3 \oplus w_{a_n} = y_n \oplus w_3 \oplus w_{a_n} \tag{7}$$

where $a_n > 3$. More general, for any power of two, one can obtain:

$$y_{n+2^t} = y_n \oplus w_t \oplus w_{a_n}, \tag{8}$$

where $a_n > t$, $a_n = f(n|2^t - 1)$ and the"|" denotes the bitwise or operator. Thus, it has been obtained a strided ("leapfrog") bounce generation [5].

In the following, we present an efficient method that we have developed and applied for implementing the Sobol pseudorandom number generator algorithm in the CUDA architecture.

3. The CUDA implementation of the Sobol random numbers generator algorithm

The implementation of the Sobol random numbers generator algorithm in the CUDA parallel programming model is facilitated by the fact that CUDA supports random writes in memory and bitwise arithmetic operations.

In order to develop the CUDA implementation of the above-described algorithm, we have first computed the values of the direction numbers on the host and then we have copied them to the device. In order to obtain a 32-bit Sobol sequence, we needed, for each dimension, at most 32 values for the directions w_r. Taking into account that in a multidimensional Sobol sequence the dimensions are independent, we have computed each dimension's points using one (or more) blocks. When we have used one block per dimension, then for each Sobol dimension a block of threads was launched, having the dimension $2^t, t \geq 6$. The $32 w_r$ values, corresponding to this dimension, are copied in the shared memory.

The k-th thread of the block bounces ahead to the value y_k, according to the first relation from the equation (7). Usually, the bit pattern of $k \otimes \left({}^{k}/_{2}\right)$ contains mostly zero values and therefore, only a few bouncing iterations are needed. As in equation (8), the thread generates the points iteratively: $y_k, y_{k+2^t}, y_{k+2^{t+1}}, \ldots$. At each step there are required the previous value of y and the new value of a_n, while the value of w_t is fixed in this iteration. As successive threads within the same warp generate successive values in the sequence, the write operations into the global memory are coalesced. When writing data in the global memory, we have first stored the numbers corresponding to the first Sobol dimension, then for the second one and the process has continued until we have reached the last dimension.

If we consider a Sobol sequence having the dimension D, a total number N of generated points, x the array that contains the generated points, then for a dimension $0 \leq d \leq D$ and an index $0 \leq k \leq N$, the k-th generated value of the d dimension is located on the position $x[d * N + k]$. If more blocks of threads were available, additional parallel thread blocks could be used per each dimension, considering the fact that the number of blocks must be a power of two. For example, if the number of parallel thread blocks is 2^{β}, then the k-th thread in a block generates the points $y_k, y_{k+2^{t+\beta}}, y_{k+2^{t+\beta+1}}, \ldots$ [5].

We have developed and implemented a suite of optimization solutions for achieving a high level of performance for our Sobol random number generator's CUDA implementation, in a broad range of scenarios and situations that we present in the following.

Solution 1. Using an efficient hybrid bouncing technique. We have implemented an efficient hybrid bouncing solution: a large bounce is performed first at the thread block level and then, within a block, each thread executes a strided bounce, as we have described in the section 2 of this paper. This solution has the advantage of simple memory coalescence when writing the output data.

Solution 2. Scaling the number of used thread blocks according to the number of dimensions and vectors. In order to attain algorithmic and hardware efficiency for the Sobol random number generator's implementation in CUDA, we have used multiple thread blocks, scaled to the number of dimensions and generated vectors. Thus, we have obtained considerable improvements, regarding the execution time and number of generated elements per second, compared to the sequential implementation run on the central processing unit.

Solution 3. Load balancing the parallel computational tasks. As we have mentioned before, one of the main advantages offered by the CUDA architecture is the huge amount of parallelism that can be employed through multiple processing threads [6]. In order to benefit from this advantage, we have processed multiple parallel instances of the Sobol random number generator's implementation, thus optimizing the computational load and benefitting from the huge computational resources of the CUDA architecture.

Solution 4. Optimal management of the task distribution among the available threads. Using a single thread for generating one element causes high memory latency, as it has not been generated an appropriate computational load for the streaming multiprocessors. The GPU reduces the memory latency by executing in parallel the concurrent threads, unlike the CPU that hides memory latency by using extensively cache memory. In order to launch multiple instances of the Sobol random number generator and to reduce memory latency, we have distributed the computational tasks among multiple execution cores, taking into account that the CUDA architecture offers 1536 cores for the GTX 680 and 480 cores for the GTX 480. After analysing the GPUs' features and experimenting with different settings for the resources' allocation, we have decided to use 256 threads per block for the GTX 480 and 512 threads per block for the GTX 680, as with these resources we have obtained the best results.

Solution 5. Using the shared memory for storing data. As the GPU's shared memory offers an improved memory bandwidth and reduced latency, we have decided to use shared memory for storing local data, thus obtaining an improvement in the coherence level and the overall performance of our Sobol random number generator in CUDA.

Solution 6. Avoiding shared memory banks conflicts. The CUDA memory banks are shared memory modules, having the same size, each of them storing a 32-bit value [7]. If the same memory bank receives multiple data requests from the same memory address or from different ones, a memory bank conflict could be triggered. In this case, the hardware device serializes the requests, putting the threads in standby and then processes the memory requests sequentially. We have avoided this process that creates a time penalty, by assuring that all the threads of a half warp read the same memory address, thus triggering a complex distribution mechanism that broadcasts data to many threads simultaneously.

Solution 7. Saving shared memory by using the warp shuffle operation. Taking into account that the warp shuffle operation is specific to the devices that have the compute capability 3.x, we have developed and applied this solution only for the GTX 680 graphic processing unit from the Kepler architecture. By using the warp shuffle operation, we have exchanged data directly between the threads of the same warp, thus managing to save a considerable amount of shared memory, maintaining the memory latency at a minimum.

Solution 8. Minimizing the synchronization operations of the parallel tasks. In order to synchronize the tasks, one must usually define a synchronization point in the application that prevents a task from continuing its execution until other tasks have arrived at a certain point. In the case of the Kepler GK104 architecture, we have used the warp shuffle operation described in **Solution 7**, in order to reduce the number of necessary synchronization operations. In the case of the GF100 architecture, we have used the shared memory and processed data in warps (as the warp shuffle operation is not supported). In this way, we were able to share data between the threads of the same warp, without having to synchronize. The only time when we had to synchronize was when sharing data between the threads of different warps.

Solution 9. Minimizing the usage of register memory. We had to take into account the fact that the threads' number was limited by the register memory requirements because each thread needs its own private and register memory [8]. In addition, this type of memory is very important in our problem as it is being used for storing the partial results. Consequently, by reducing the number of registers, we were able to generate the necessary number of threads.

Solution 10. Using the CPU for generating random numbers until reaching a certain threshold. After we have analysed the obtained experimental results, we have concluded that the CPU offers the best results until a certain threshold is reached. This threshold is influenced by the number of generated vectors and the vector's dimension because, until this threshold has been reached, the computational load is not high enough as to harness the huge processing power of the GPUs.

We have developed and run a series of benchmark tests for analysing the efficiency of the developed optimization solutions for our Sobol random number generator's CUDA implementation. In the following, we depict and analyse the obtained experimental results.

4. Experimental results

After having developed and applied the optimization solutions for the Sobol random number generator's CUDA implementation, we have analysed their efficiency through a series of experimental tests that we have developed using three different processing units, namely: the central processing unit Sandy Bridge Intel i7-2600K, the graphic processing unit GTX 480 from the CUDA-Fermi architecture and the GTX 680 from the CUDA-Kepler architecture.

In order to obtain an objective measurement of the energy consumption, when we have run the tests on the CPU, we have used only the integrated Intel HD Graphics 3000 graphic core from the i7-2600K CPU and no other discrete graphic card was installed in the system. Because neither the CPU nor the integrated graphic core allows the execution of the developed optimized Sobol CUDA implementation, we have run on the CPU a sequential version of the generator. In all the three cases, beside the above mentioned software components we have used the Windows 8 Pro 64-bit operating system, a total amount of 2x4GB of DDR3 random access memory dual channel at 1333MHz, the CUDA Toolkit 5.0 and the NVIDIA developer driver version 306.97 in order to program and access the GPUs.

We have benchmarked different allocation settings regarding the GPUs' thread blocks' sizes (taking into account the features of each graphic processing architecture) and we have finally chosen the optimum resources allocation, that have provided the best level of performance: 256 threads per block for the GTX

480 and 512 threads per block for the GTX 680. As our random number generator is designed to be implemented in various GPU applications, having different computational requirements, complexities and dimensions, we have chosen to measure within our tests only the execution time of the generator, but not the transfer times between the CPU and the GPU, as these times are application-specific.

In our experimental tests, we have successively generated a number of $10, 10^2, 10^3, 10^4, 10^5, 10^6$ and 10^7 vectors of float type elements, ranging from 10 to 10^4 elements. In order to obtain accurate, reliable results, we have run 1000 iterations for each of the tests and then we have computed their average. We have computed the execution time (in milliseconds) and the number of generated samples per second (in millions per second) for each case, when generating the samples on the i7-2600K CPU and on the GTX 480, GTX 680 GPUs.

After analysing the obtained experimental results, we have concluded that for a small number of generated vectors and for low dimension vectors, the best results (the lowest execution time, the highest number of generated samples per second) have been recorded on the CPU, as it has not been generated a sufficient computational load that fully employed the huge parallel computing power of the GPUs. For each number of generated vectors, as the vector's dimension increases, the GTX 680 and then the GTX 480 surpass the CPU and obtain the best performance, starting with a certain threshold that is influenced by the vectors' number and dimension. We have also noticed that, as the number of generated vectors has increased, the above mentioned threshold decreased and from a certain point the threshold does not exist anymore, as the GPU surpasses the CPU in every case, without being influenced by the number of dimensions. Thus, when generating a number of 10^4 vectors or higher, we have noticed that the best performance has been obtained on the GTX 680 GPU, then on the GTX 480 (Table 1).

Table 1

The threshold from which the CPU's performance has been surpassed by the GPUs

The number of vectors	The number of dimensions - threshold
10	2500
10^2	250
10^3	50
$10^4 - 10^7$	-

Another aspect worth mentioning is that, starting from a number of 10^5 generated vectors and a certain number of dimensions (e.g. 10^4 when generating 10^5 vectors, 10^3 when generating 10^6 vectors or 10^2 when generating 10^7 vectors), the necessary memory requirements exceeded our system's available memory and from this point forward, if we had decided to continue, we would have been forced to write the generated data on the disk and generate the rest of the numbers in partitions that do not exceed the system's available memory.

For example, when generating a number of $n = 10^4$ vectors, with a dimension ranging from 10 to 10^4 float type elements, we have obtained in all the analysed situations the lowest execution time on the GTX 680 and then on the GTX 480 (Table 2, Fig. 1).

Table 2

Synthetic experimental results – the execution time for $n = 10^4$ vectors

No.	Dimensions	Execution time (ms)		
		GTX 480	GTX 680	i7-2600K
1	10	0.208197	0.141573	0.988593
2	10^2	0.181088	0.158399	9.814390
3	10^3	0.51001	0.386381	98.743101
4	10^4	3.64851	2.788590	985.239000

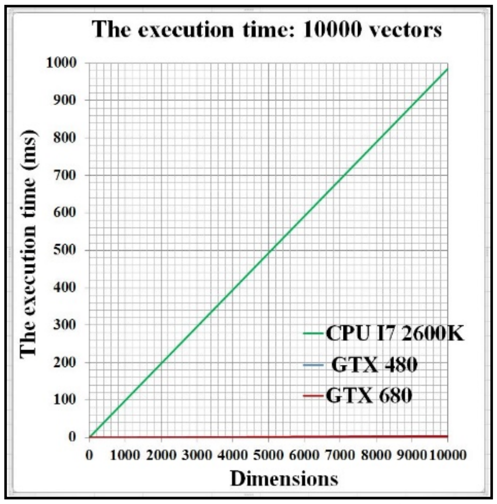

Fig. 1. The execution time for $n = 10^4$ vectors

Analysing the number of generated samples per second, we have noticed that the GTX 680 offers the best performance and is succeeded by the GTX 480, clearly surpassing the CPU's performance (Table 3, Fig. 2).

Table 3

Synthetic experimental results – the number of generated samples per second for $n = 10^4$ vectors

No.	Dimensions	Millions of numbers generated/s		
		GTX 480	GTX 680	i7-2600K
1	10	480.314	706.352	101.154
2	10^2	5522.18	6313.171	101.891
3	10^3	19607.4	25881.201	101.273
4	10^4	27408.5	35860.502	101.498

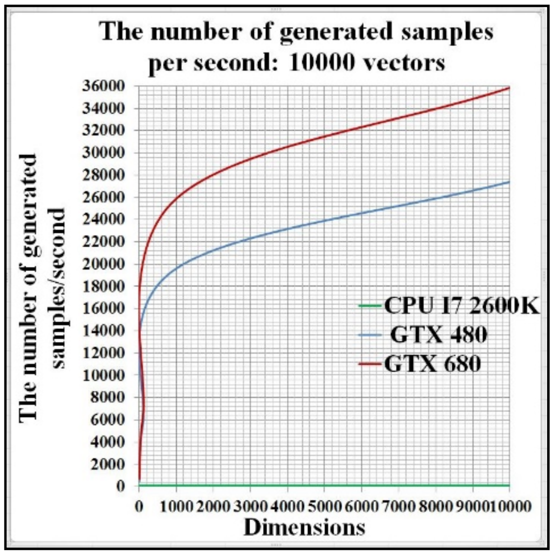

Fig. 2. The number of generated samples per second for $n = 10^4$ vectors

Of particular interest in our research was to highlight the energy efficiency of our Sobol random number generator's CUDA implementation, comparing it to the efficiency of the sequential approach, run on the CPU. Therefore, we have computed the total execution time for all of our analysed situations within the experimental tests (ie for all the number of vectors, for all their dimensions, for all the 1000 iterations and for each of the 3 processing units). In order to determine the system's power (kW) and the total energy consumption in all of the analysed cases, we have used the Voltcraft Energy Logger 4000, an energy consumption meter device (Table 4).

Table 4

The system's power and the total energy consumption

The processing unit	i7-2600K	GTX 480	GTX 680
The total execution time (h)	1.212	0.006	0.004
The system's power (kW)	0.198	0.358	0.307
The total energy consumption (kWh)	0.240	0.002	0.001
The GPU's consumption compared to the CPU's		120 times lower	240 times lower

By analysing the above synthesized results, we have noticed that the total energy consumption is 120 times lower when the Sobol random number generator is run on the GTX 480 GPU than when it is run on the CPU and 240 times lower when the Sobol random number generator is run on the GTX 680 GPU than when it is run on the CPU. Thus, by using the GTX 680 GPU when running the Sobol random number generator, we have obtained 240 times lower execution costs then by using the CPU and 120 times lower when using the GTX 480 GPU.

After analysing the obtained experimental results, we have concluded that the optimization solutions, that we have developed for improving the performance of our Sobol random number generator's CUDA implementation, offer remarkable results in a wide range of situations and scenarios. Thus, the generator proves to be a powerful and useful tool in many applications that require generating random numbers.

5. Conclusions

We have aimed in our research to harness the novel technical features and the huge parallel computing power offered by the latest generations of CUDA-enabled graphic processing units (GPUs) from the Fermi GF100 and the Kepler GK104 architectures. We were able to achieve this by improving continuously and progressively the optimization solutions.

We have obtained very promising results, the developed solutions offering a high level of performance and applicability. As we have generated a large volume of output data on different GPU architectures, using a large number of tests' iterations, we have obtained a detailed analysis of our Sobol random number generator's characteristics.

Lately, in the literature, the interest in implementing random number generators on parallel architectures has continued to grow and a series of works have treated this topic. However, to the best of our knowledge, none of these works has studied the development of specific solutions for improving the performance of random number generators using CUDA-enabled GPUs of compute capability 3.x. By analysing the experimental results, we have noticed that our optimization solutions applied to the Kepler GK104 architecture achieve a huge level of performance when generating Sobol random sequences. The optimization solutions developed within this research prove their effectiveness and usefulness, the CUDA implementation of the Sobol generator proving to be a novel approach in generating random numbers using graphics processing units, a powerful and useful tool in a wide range of applications that require the generation of random numbers.

Undoubtedly, graphic processing units that support the Compute Unified Device Architecture have an enormous potential to overcome the performance limitations of current central processing units' architectures, offering considerable advantages in developing solutions that optimize data processing and lead to huge improvements in energy efficiency and computing performance.

R E F E R E N C E S

[1] *I.M. Sobol*, “Distribution of points in a cube and approximate evaluation of integrals”, U.S.S.R Comput. Maths. Math. Phys., **vol 7**, 1967, pp. 86–112

[2] *A. Antonov, V.M. Saleev*, “An economic method of computing LP sequences”, USSR J. Comput. Math. Math. Phys., **vol. 19**, 1979, pp. 252–256

[3] *P. Bratley, B. Fox,* “Algorithm 659: implementing Sobol’s quasirandom sequence generator”, ACM Trans. Model. Comput. Simul., **vol. 14 (1)**, 1998, pp. 88–100

[4] *M. Matsumoto, T. Nishimura*, “Mersenne Twister: a 623-dimensionally equidistributed uniform pseudo-random number generator”, ACM Transactions on Modeling and Computer Simulation, **vol. 8 (1),** 1998, pp. 3–30

[5] *T. Bradley, J. du Toit, R. Tong, M. Giles, P. Woodhams,* “Parallelization Techniques for Random Number Generators”, in GPU Computing Gems Emerald Edition, 2011

[6] *A. Tăbuşcă,* “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 (1),** 2010, pp. 164-168

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

[8] *G. Garais*, “Web Applications Readability”, Journal of Information Systems & Operations Management, **vol. 5 (1)**, 2011, pp. 114-120