Professor Ion LUNGU, PhD
E-mail: ion.lungu@ie.ase.ro
Economic Informatics Department
The Bucharest Academy of Economic Studies
Lecturer Dana-Mihaela PETROŞANU, PhD
E-mail: danap@mathem.pub.ro
Department of Mathematics-Informatics I
The Polytechnic University of Bucharest
Lecturer assistant Alexandru PÎRJAN, PhD
E-mail: alex@pirjan.com
Statistics and Mathematics Department
The Romanian-American University

SOLUTIONS FOR IMPROVING THE PERFORMANCE OF RANDOM NUMBER GENERATORS USING GRAPHICS PROCESSING UNITS

Abstract. *In this paper, we have researched the possibility of using graphics processing units (GPUs) for generating a variety of random numbers. We have compared the performance obtained on the latest GPUs architectures with the one recorded on a high-end central processing unit, we have analyzed the main features of each architecture that influence the performance, we have developed solutions for optimizing the performance of random number generators. Although there is a great interest lately in implementing random number generators on parallel architectures, none of the works so far (to our best knowledge) has developed and studied implementations on Kepler, the latest Compute Unified Device Architecture (CUDA) architecture. We have developed a high performance implementation, harnessing the computational resources of the latest Fermi and Kepler architectures, studying their optimization potential, their impact on the execution time, on the generated samples bandwidth, on the energy consumption and on the execution cost.*

Keywords*: random number generators, graphics processing units, Compute Unified Device Architecture, Kepler architecture, execution threads.*

JEL Classification: C15, C61, C63, C88

1. Introduction

A high performance random number generator is of paramount importance in a wide range of applications, like mathematical systems, signal processing, nuclear physics, cryptography, financial management and simulations. Most of the

random number generators are sequential, facing severe performance limitations. The huge parallel computational power of graphics processing units (GPUs) that incorporate the Compute Unified Device Architecture (CUDA) represents a great opportunity to develop and implement efficient parallel random number generators that overcome the limitations of previously central processing units (CPUs) based architectures that sequentially generated samples of data.

In this paper, we have analyzed the use of GPUs in parallel random number generation. We have implemented the Mersenne Twister random number generator algorithm in the parallel Compute Unified Device Architecture and we have developed optimization solutions that harness the computational power of the GPUs. We have generated a wide range of samples (ranging from 35-size up to 120 million-size samples). We have researched the obtained performance on the most powerful CUDA architectures (Fermi and Kepler) and also on a high-end Central Processing Unit (Sandy Bridge), we have analyzed the technical specifications of each architecture and their influence on the obtained performance level thus obtaining optimization solutions targeting the performance of the Mersenne Twister random number generator on the CUDA architecture.

Lately, there has been a lot of interest in the literature for optimizing random number generators, taking into account their numerous applications, but none of the works so far (to our best knowledge) has studied the optimization potential of the latest CUDA architecture, Kepler, its impact on the performance, on energy consumption due to the generation of random numbers samples, the impact of the generated sample's size on the performance and on the processing costs.

2. The Compute Unified Device Architecture

At the beginning, the graphics processing units' (GPUs') sole purpose was to render graphics. As time passed by, the necessary computational power required for graphics rendering kept increasing, leading to an inherent evolution of the graphics processing unit. In 2006, the Nvidia company announced and launched the Compute Unified Device Architecture (CUDA), a novel parallel programming model that used the huge computational power of the GPUs in order to solve demanding scientific computational problems in a much more efficient manner than by using solely the central processing units. This architecture makes it possible for the GPU to execute and run programs that have been developed in programming languages such as C, C++, Direct Compute, OpenCL, FORTRAN. The computational tasks are executed in parallel by many threads that are grouped in thread blocks, that are further grouped in grids of thread blocks. The graphics processing unit instantiates a special function (kernel) on a grid of thread blocks and each thread of the block executes a kernel's instance, having an unique attached ID, private memory and associated registers [1].

The CUDA processing model offers three levels of abstraction: thread hierarchy, memory hierarchy and barrier synchronization [1]. Using a set of C-

language extensions, the developer has access to these abstractions that offer the possibility to implement fine-grained parallelism for data and threads along with large grained parallelism for data and tasks. The abstractions allow the developer to sub-divide the problems into smaller ones that can be processed in parallel. This offers a high degree of scalability as every thread can cooperate when processing the tasks and so each task can be processed by any of the available CUDA cores. As a consequence it is possible to execute a program compiled in CUDA C on any number of CUDA cores.

The hierarchy of threads is associated to the hierarchy of the CUDA hardware processors: one or more kernel grids are executed by the graphic processing unit; the streaming multiprocessor (SM) launches in execution the thread blocks; within the blocks, the CUDA cores of the streaming multiprocessor run the threads. A streaming multiprocessor can process groups of up to 32 threads. A group of 32 threads is called a warp. Every multiprocessor has a set of 32-bit registry and a shared memory zone, accessible by each core of the multiprocessor but is not visible to other multi-processors (**Figure 1**).

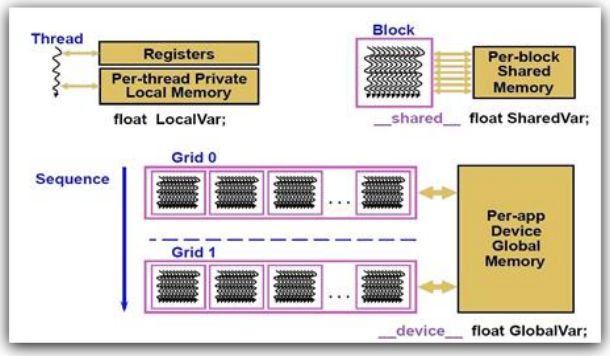

Figure 1. NVIDIA Compute Unified Device Architecture (CUDA)

The size of the shared memory and the number of registry varies from one GPU generation to another. The multiprocessor also contains a read only memory cache for the texture and another one for storing the constants.

When developing algorithms in the CUDA programming model, one must divide the tasks into smaller fragments that can be processed by the desired number of available thread blocks, each of them containing a previously specified number of threads. The most important factor for obtaining a high degree of performance is to take into account how the tasks are allocated to the available thread blocks.

This architecture uses the GPU's computational processing power, offering a new parallel programing model useful in solving high-demanding computational processing tasks in a much more efficient manner than using solely central processing units (CPUs) [2]. The CUDA architecture makes it possible to improve the software performance of a wide range of data processing applications. In this

context, the improvement of many applications that make use of random numbers can be achieved using random number generators, accelerated by many-core graphics processing units.

Once the CUDA-C language was introduced, the application developers were able to benefit from the increased computational power of the latest CUDA-enabled GPUs, through a standard programming language. CUDA-C allows the developers to control how tasks are being parallelized and executed by the parallel threads of the GPU [3] and also offers a high level of control that facilitates the development of resource demanding applications.

In our research we have used two of the most powerful graphics processing cards: the GTX 480 from the Fermi architecture GF100 and the GTX 680 from the Kepler GK104 architecture.

3. The Mersenne Twister random number generator algorithm

An algorithm generating a sample of numbers that approximates the properties of random numbers is a pseudorandom number generator. The algorithm starts with a small set of initial values, that contains a truly random seed and this set of values determines entirely the sequence. Therefore, the algorithms of this type are also called deterministic random bit generators. Due to their efficiency in generating numbers, these pseudorandom number generators are extremely useful in many practical applications (Monte Carlo financial simulations, physical systems, etc). When using a pseudorandom number generator, one must assure, through rigorous mathematical methods, that the generated numbers satisfy a certain degree of randomness that fits to the intended purpose.

Developed by Makoto Matsumoto and Takuji Nishimura [4], the Mersenne Twister pseudorandom number generator algorithm surpasses many of the previous algorithms' limitations. This algorithm is characterized by a high performance, efficient use of memory, long period and good distribution property. The Mersenne Twister algorithm is faster than many other known generators, being equidistributed in up to 623 dimensions for 32-bit sequences, having a huge period: $2^{19937}-1$ iterations. All of these advantages, make the Mersenne Twister a powerful tool in generating random number samples.

In the following, we mention a few notations and describe some mathematical aspects regarding the Mersenne Twister random number generator algorithm, using these notations: i and j are two fixed natural numbers, $1 \le i \le j$ (the parameter j is called the degree of recurrence and i is called the middle term); $(\mathbf{w}_k)_{k\ge 0}$ is a sequence of d-dimensional bit vectors; $0 \le s \le d-1$ a natural number; $(\mathbf{w}_k^l \mid \mathbf{w}_{k+1}^r)$ a d-dimensional bit vector obtained by concatenating the left $d-s$ bits of the $\mathbf{w}_k$ vector and the right s bits of $\mathbf{w}_{k+1}$; $\mathbf{M}$ a quadratic d-dimensional bit matrix (called twister).

The Mersenne Twister algorithm [5], generates d -dimensional bit vectors using the recurrence:

$$\mathbf{w}_{k+j} = \mathbf{w}_{k+i} \oplus (\mathbf{w}_k^l \mid \mathbf{w}_{k+1}^r) \cdot \mathbf{M} \tag{1}$$

where $\oplus$ is the addition of bit values modulo 2 (exclusive OR operation, XOR) and $(\mathbf{w}_k^l \mid \mathbf{w}_{k+1}^r) \cdot \mathbf{M}$ represents the multiplication of the d -dimensional bit vector $(\mathbf{w}_k^l \mid \mathbf{w}_{k+1}^r)$ and the twister $\mathbf{M}$.

All arithmetic operations are modulo 2, and therefore the equation (1) represents a linear reccurence of vectors in the field $\{0,1\}$. The vectors $\mathbf{w}_0, \mathbf{w}_1, ..., \mathbf{w}_{j-1}$ are called the initial seeds and are known. Using the equation (1) and choosing $k = 0$, it is generated the vector $\mathbf{w}_j$. When k takes values $1,2,...$, the generator provides the vectors $\mathbf{w}_{j+1}, \mathbf{w}_{j+2,...}$ The chosing of different values of s provides various particular cases of reccurence: the algorithm for the case $s = 0$ is studied in [6], [7] and is named TGSFR, while in the case $s = 0$ and $\mathbf{M} = \mathbf{I}_d$ the algorithm is named GFSR and is studied in [8]. By choosing a convenient form of the matrix $\mathbf{M}$, the computation of the $(\mathbf{w}_k^l \mid \mathbf{w}_{k+1}^r) \cdot \mathbf{M}$ product from equation (1) becomes simple and fast. In [4] is proposed a form called the companion matrix:

$$\mathbf{M} = \begin{pmatrix} 0 & 1 & 0 & ... & 0 \\ 0 & 0 & 1 & ... & 0 \\ & & ... & & \\ 0 & 0 & 0 & ... & 1 \\ m_{d-1} & m_{d-2} & m_{d-3} & ... & m_0 \end{pmatrix} \tag{2}$$

where $\mathbf{m} = (m_{d-1}, m_{d-2}, ..., m_0)$ is a d -dimensional bit vector. If $\mathbf{w} = (w_{d-1}, w_{d-2}, ..., w_0)$ is a d -dimensional bit vector, the multiplication of the vector $\mathbf{w}$ with the matrix $\mathbf{M}$ gives:

$$\mathbf{w} \cdot \mathbf{M} = \begin{cases} (0, w_{d-1}, w_{d-2}, w_{d-3}..., w_1), \; if \; w_0 = 0 \\ (m_{d-1}, w_{d-1} \oplus m_{d-2}, w_{d-2} \oplus m_{d-3}, ..., w_1 \oplus m_0), \; if \; w_0 = 1 \end{cases} \tag{3}$$

that can be written more concise:

$$\mathbf{w} \cdot \mathbf{M} = \begin{cases} logicalrightshift(\mathbf{w}), \; if \; w_0 = 0 \\ logicalrightshift(\mathbf{w}) \oplus \mathbf{m}, \; if \; w_0 = 1 \end{cases} \tag{4}$$

using the logical right shift operator.

Each generated bit vector w is then multiplied by a invertible quadratic d -dimensional bit matrix $\mathbf{T}$, called tempering matrix [4], [6], [7]. The multiplication with this matrix, that improves the distribution properties and the least significant bits, is done using some successive transformations:

$$\mathbf{y} = \mathbf{w} \oplus (\mathbf{w} >> u) \tag{5}$$

$$\mathbf{y} = \mathbf{y} \oplus ((\mathbf{y} << s) \,\&\, \mathbf{b}) \tag{6}$$

$$\mathbf{y} = \mathbf{y} \oplus ((\mathbf{y} << t) \,\&\, \mathbf{c}) \tag{7}$$

$$\mathbf{z} = \mathbf{y} \oplus (\mathbf{y} >> l) \tag{8}$$

where u, s, t, l are natural numbers, $\mathbf{b}$ and $\mathbf{c}$ are suitable d-size bitmasks, the notation $w >> u$ signifies the logical u-bit right shift and the notation $y >> l$ signifies the logical l-bit left shift. The equations (5)-(8) transform the bit vector $\mathbf{w}$ in $\mathbf{z} = \mathbf{w} \cdot \mathbf{T}$.

In order to use the equation (1), one must previously know j bit vectors of d-dimension. We denote by $\mathbf{w}[0: j-1]$ an array of j bit unsigned integers of d-dimension, p a natural variable and $\mathbf{l,r,m}$ constant unsigned integers of d-size. Using these notations, the algorithm can be resumed in the following steps:

1. The bitmask for the left $d-s$ bits: $\mathbf{l} \leftarrow \underbrace{1\ldots1}_{d-s}\underbrace{0\ldots0}_{s}$

The bitmask for the right s bits: $\mathbf{r} \leftarrow \underbrace{0\ldots0}_{d-s}\underbrace{1\ldots1}_{s}$

The last row of the companion matrix: $\mathbf{m} \leftarrow m_{d-1}m_{d-2}\ldots m_1 m_0$

2. $p \leftarrow 0$

Non zero initial-values: **w[0], w[1],..., w[j − 1]**

3. Concatenating (w_p^l, w_{p+1}^r) : $\mathbf{x} \leftarrow (\mathbf{w[p]} \,\&\mathbf{l}) \,\mathrm{OR}\, (\mathbf{w[(p+1)} \bmod j] \&\mathbf{r})$

4. Multiplying $\mathbf{M}$:

$$\mathbf{w[p]} \leftarrow \mathbf{w[(p+i)} \bmod j] \,\mathrm{XOR}\,(x >> 1)$$

$$\mathrm{XOR}\begin{cases} 0 \text{ if the least significant bit of } \mathbf{x} = 0 \\ a \text{ if the least significant bit of } \mathbf{x} = 1 \end{cases}$$

5. Computing $\mathbf{w[p]} \cdot \mathbf{T}$:

$$\mathbf{x} \leftarrow \mathbf{w[p]}$$
$$\mathbf{x} \leftarrow \mathbf{x} \,\mathrm{XOR}\, (\mathbf{x} >> u)$$
$$\mathbf{x} \leftarrow \mathbf{x} \,\mathrm{XOR}\, ((\mathbf{x} << s) \,\&\, b)$$
$$\mathbf{x} \leftarrow \mathbf{x} \,\mathrm{XOR}\, ((\mathbf{x} << t) \,\&\, c)$$
$$\mathbf{x} \leftarrow \mathbf{x} \,\mathrm{XOR}\, (\mathbf{x} >> l)$$
$$output\mathbf{x}$$

6. $p \leftarrow (p+1) \bmod j$

7. Return to Step **3**.

In the following, we present an efficient method for designing and implementing the Mersenne Twister pseudorandom number generator algorithm in the CUDA architecture.

4. The CUDA implementation of the Mersenne Twister random numbers generator algorithm

As we have depicted above, the Mersenne Twister algorithm uses the following 11 parameters: the dimension d of bit vectors; the degree of recurence j and the middle term i; the separation point s, that appears in the concatenation $(\mathbf{w}_k^l \mid \mathbf{w}_{k+1}^r)$; the d-dimensional bit vector $\mathbf{m} = (m_{d-1}, m_{d-2}, ..., m_0)$, the last row of the companion matrix $\mathbf{M}$; the tempering shift parameters u, s, t, l; the tempering d-size bitmasks, $\mathbf{b}$ and $\mathbf{c}$.

The Mersenne Twister algorithm is suitable to be implemented in the CUDA parallel programming model, as CUDA supports bitwise arithmetic and random writes in memory. One must take into account the fact that the Mersenne Twister algorithm is iterative, just like most of the pseudorandom number generators. Therefore, it is difficult to parallelize one of the state's update steps over multiple threads. The graphic processing unit must employ thousands of execution threads in the grid of thread blocks in order to benefit at maximum from its computational power. In order to overcome this problem and harness the computational potential of the GPU, we have processed in parallel more instances of the pseudorandom number generator. After analysing the experimental results, we have observed that although the initial values have been chosen differently, the generated output sequences were correlated when we have used the same parameters for the random number generator. So, in order to obtain an efficient algorithm's implementation in CUDA, we arrived at the solution of using the DCMT library developed by Makoto Matsumoto and Takuji Nishimura (1998) [9] in order to dynamically create the parameters for the Mersenne Twister algorithm. Thus we have encapsulated the thread's ID into the algorithm's parameters for each thread, so that they can update their own twister matrix independently, assuring in this way that the output contains random values.

We have first used DCMT to compute for each thread the custom parameters that our CUDA implementation uses in the process of generating the random numbers. Although, as we have previously specified at the beginning of this section, the Mersenne Twister algorithm requires the specification of 11 parameters, only the parameters that are bit vectors vary, at the same moment of time and DCMT seed, depending on the chosen thread: the last row of the companion matrix, the d-dimensional bit vector $\mathbf{m} = (m_{d-1}, m_{d-2}, ..., m_0)$; the tempering d-size bitmasks, $\mathbf{b}$ and $\mathbf{c}$.

All the other parameters are being shared by all the threads. During the execution, the CUDA threads store their state in a local memory array. Each thread within a warp accesses the same state index as the parameters i and j are the same for every thread. The read and write operations performed on the state arrays are always coalesced. The random uniform distributed number sequences, generated by the Mersenne Twister algorithm or any other random number generator, can be

transformed in standard normal distribution, using the Box-Muller transformation [10].

In order to obtain an improved version of the Mersenne Twister algorithm's implementation in CUDA, we have developed and implemented a series of solutions for optimizing its performance:

Solution 1. Balancing the computational load, using multiple simultaneous Twisters processed in parallel. In order to efficiently employ the thousands of parallel threads offered by the CUDA architecture, we have processed multiple simultaneous Twisters processed in parallel, thus correctly balancing the computational load. This technique generates a sufficient computational load so that the computational power of all the CUDA cores is entirely harnessed during the execution phases.

Solution 2. Optimizing the distribution of tasks to each of the threads. Generating a single element per thread does not create a sufficient computational load for reducing the memory latency. Each thread incorporates its own private registers, private memory and a state counter. A thread may independently process a part of the source code. At the hardware level, the graphic processing unit manages and executes hundreds of parallel concurrent threads, thus reducing the memory latency and scheduling overloading. The Kepler GK104 architecture has 1536 CUDA cores, while the Fermi GF100 architecture offers 512 processing cores. The graphic cards we have used in our research incorporate 1536 CUDA cores for the GTX 680 and 480 CUDA cores for the GTX 480. In order to completely employ and benefit from the computational power of all these cores, we have distributed the tasks of processing multiple simultaneous Twisters, thus employing hundreds of threads in the process of generating random numbers. For improving the software performance in CUDA, when we have defined the threads, we took in consideration the high latency of global memory. While the CPU's architecture makes use of large dimension cache memory in order to hide memory latencies, the CUDA architecture makes use of thousands of threads to compensate and reduce the memory latency [11]. We have obtained the best results when using thread blocks of 256 threads per block for the GTX 480 and blocks of 512 threads for the GTX 680.

Solution 3. Minimizing the number of used registers. Like we have previously mentioned, the CUDA architecture uses multiple threads in order to reduce memory latency, but their number is often limited by their registry requirements. Therefore, we have decided to minimize the number of used registers in order to maximize the number of available threads. Optimizing the number of used registers is very important in generating random numbers, because many registers are required to store and manage the intermediate results.

Solution 4. Shared memory usage. To obtain an efficient implementation of the random number generator in CUDA, we have used the GPU's shared memory in order to locally store the data that must be processed and to improve the

level of coherence, thus optimizing the overall performance, due to the reduced latency and improved memory bandwidth offered by the shared memory.

Solution 5. Managing shared memory banks conflicts. The CUDA shared memory contains multiple memory modules of equal size, called memory banks [2]. A 32-bit value is stored by each of the memory banks. When multiple data is requested from the same memory bank, a memory bank conflict might occur. These conflicts arise even if the requests come from the same address of memory or from different addresses. When such a conflict happens, the hardware device initiates a serialization process that puts all the threads in a standby mode until all the requests from the memory addresses have been processed. This process can be prevented if the same shared memory address is read by all the threads involved. In this case, a complex distribution mechanism that delivers data to many threads simultaneously is triggered [11].

Solution 6. Using the warp shuffle operation. This is a solution that we have implemented only for the GK104 Kepler architecture (GTX 680 processor), because the warp shuffle operation is only supported by devices having the compute capability 3.x. By using the warp shuffle operation, we were able to exchange data between the threads within a warp directly (without having to use global or shared memory) with an even lower latency than when using shared memory. In this way we were able to save a large amount of shared memory space for other computational tasks.

Solution 7. Managing the synchronization of parallel tasks. The synchronization is usually implemented by defining a synchronization barrier within the application, from which a task cannot continue anymore until another task reaches a certain point. For the Kepler GK104 architecture (GTX 680) we have minimized the number of synchronization operations by using the warp shuffle operation (depicted in **Solution 6**) for exchanging data directly between the threads within a warp. For the GF100 Fermi architecture (that does not support the warp shuffle operation) we had to use the shared memory. We have minimized the number of synchronization operations by processing the data in warps (groups of 32 threads). Within a warp we didn't have to use the synchronization operations in order to share data between threads, as the instructions were being executed in a SIMD (Single Instruction, Multiple Data) model. We have used the synchronization operations only when sharing data between different warps.

Solution 8. Using multiple thread blocks. We have optimized the CUDA implementation of the Mersenne Twister algorithm using multiple thread blocks, leading to a significant improvement of the running time compared to the situation when the algorithm runs on CPU. When implementing the Mersenne Twister random number generator in CUDA, both algorithmic and hardware efficiency have been taken into account. We have also optimized the memory access patterns in order to reduce latency and improve the performance.

Solution 9. Minimizing the number of executed instructions. We have noticed that, as the process of generating numbers progresses, the necessary

number of active threads decreases. When this number is less than or equal to 32, it means that only one active warp remains. As the synchronization operations are not necessary within the same warp, nor the checking of the threads' indexes, we have decided to remove these instructions, thus obtaining an increase in the overall performance.

Solution 10. Minimizing data transfer from/to external memory. Taking into account that the data transfer from/to external memory consumes a large amount of computational resources, we have minimized as much as possible those data transfers, obtaining thus a significant improvement regarding the execution time of our CUDA implementation of the random number generator algorithm.

Solution 11. Using the CPU instead of the GPU to generate low dimension output vectors. Analysing our experimental results, we have noticed that when the output vector's size is relatively small (35-1500 elements), it is not generated an enough computational load in order to fully employ the huge parallel processing capacity of the GPUs. Therefore, we have decided to generate the numbers in this case using the CPU, as its performance exceeds those of the tested GPUs when low dimension output vectors are required.

In the following we benchmark and analyze the obtained experimental results of our pseudorandom number generator's CUDA implementation.

5. Experimental results

In order to prove the efficiency of our solutions for optimizing the Mersenne Twister algorithm's implementation in CUDA, we have developed a series of experimental tests. For evaluating the performance, we have used the following hardware and software configuration:

- the Windows 8 64-bit operating system
- the Intel i7-2600K CPU from the Sandy Bridge architecture clocked at 4.6 GHz
- 8 GB (2x4GB) DDR3 RAM running in dual channel at 1333 MHz
- the NVIDIA graphic cards GeForce GTX 480 and GTX 680
- the CUDA Toolkit 5.0
- the NVIDIA developer driver version 306.94.

After experimenting with different block sizes, we have concluded that the best level of performance is achieved with 256 threads per block in the case of the GTX 480 GPU and a block-size of 512 threads for the GTX 680 GPU. We have designed our random number generator to be implemented in a wide class of GPU applications. As the transfer times between the central processing unit and the graphic processing unit is influenced by the complexity, dimension and specific of each application, we have decided to measure only the random numbers generation's execution time without including the transfer times.

We have measured the GPU's average execution time needed for generating a random number sample, using the CUDA Application Programming

Interface (CUDA API). We have preferred this method instead of those based on operating system timers in order to avoid having latency from other sources, latency that could have appeared taking into account that the execution is asynchronous (the CPU continues its execution, before the GPU would have finished the random generation of samples). We have created two timestamps using the "cudaEvent_t" instruction, we have associated two events to these timestamps and we have marked the beginning and the end of the processing that generates the random numbers using the "cudaEventRecord()". We have used the "cudaEventSynchronize()" method to be sure that all the threads have finished the processing before measuring the execution time using the "cudaEventElapsedTime" instruction. We have used this approach in order to get a reliable and accurate measurement of the execution time in the benchmark suite of our random number generator's implementation in CUDA (**Figure 2**).

```
float TimpulTotal = 0;
float timpul = 0;
cudaEvent_t inceput, sfarsit;
cudaEventCreate(&inceput);
cudaEventCreate(&sfarsit);
cudaEventRecord(inceput, 0);
//.....
GenerareMersenne(obGenerator, (float*) stocareValori, dimEsantion);
//.....
cudaEventRecord(sfarsit, 0);
cudaEventSynchronize(sfarsit);
//calculam timpul dintre evenimentul de inceput si cel de sfarsit
CUDA_SAFE_CALL( cudaEventElapsedTime(&timpul, inceput, sfarsit));
TimpulTotal += timpul;
```

Figure 2. Measuring the execution time of the random number generator's CUDA implementation

The first set of tests evaluates the performance obtained by applying our random number generator's CUDA implementation for generating sequences of float type elements ranging from 35 to 150000000 elements. We wanted to be sure that we obtain accurate results that we can rely upon. Therefore, we have run 10000 iterations and computed their average results when measuring the execution time (in milliseconds) and the bandwidth (in millions of numbers generated per second) for each generated vector on the GTX 480, GTX 680 and on a random number generator on the i7-2600K central processing unit (**Table 1**, **Table 2)**.

Table 1. **The experimental results recorded on the i7-2600K, GTX 480 and GTX 680 – the execution time**

Test no.	Number of elements	Execution time (ms)		
		i7-2600K	**GTX 480**	**GTX 680**
1	35	0.000754651	0.008342884	0.006257163
2	128	0.002263952	0.013071522	0.010195787
3	260	0.003803439	0.01286069	0.009902731
4	512	0.011712179	0.014740216	0.012555162
5	1500	0.020466127	0.032979992	0.024734994
6	30000	0.402470303	0.026143044	0.020391574
7	65589	0.900992584	0.094753468	0.073907705
8	120674	1.638859749	0.050870508	0.039678996
9	262145	3.642608261	0.087344265	0.067255084
10	500238	6.753006744	0.125052893	0.093789670
11	1048576	14.09295044	0.264924192	0.201342386
12	2097152	28.16365356	0.493946886	0.370460165
13	4194334	56.30684204	1.008645535	0.786743517
14	8388600	112.6196045	1.957686234	1.468264675
15	16000569	214.8572021	3.678578949	2.869291580
16	32009634	429.6052246	7.041303253	5.351390472
17	48097624	646.2623535	9.915622711	7.635029488
18	60000123	805.5741211	13.50315247	10.26239587
19	120000000	1612.357715	26.08548584	19.82496924
20	150000000	2017.803516	30.28605347	23.32026117
Total execution time – 10.000 tests (h)		16.5306114454	0.2630598862	0.2012467151
The system's power (kW)		0.1980000000	0.3580000000	0.3070000000
Total energy consumption (kWh)		3.2730610662	0.0941754392	0.0617827415
The GPU's consumption compared to the CPU's			34.75 times lower	52.98 times lower

Table 2. The experimental results recorded on the i7-2600K, GTX 480 and GTX 680 – the bandwidth

Test no.	Number of elements	Bandwidth (millions of numbers generated/s)		
		i7-2600K	GTX 480	GTX 680
1	35	46.3791	4.1952	5.5936
2	128	56.5383	9.7923	12.5542
3	260	68.3592	20.2166	26.2554
4	512	43.7152	34.7349	40.7800
5	1500	73.2918	45.4821	60.6428
6	30000	74.5397	1147.5328	1471.1959
7	65589	72.7964	692.2069	887.4447
8	120674	73.6329	2372.1800	3041.2564
9	262145	71.9663	3001.2846	3897.7722
10	500238	74.0763	4000.2113	5333.6151
11	1048576	74.4043	3958.0228	5207.9248
12	2097152	74.4631	4245.7035	5660.9379
13	4194334	74.4907	4158.3826	5331.2597
14	8388600	74.4861	4284.9563	5713.2751
15	16000569	74.4707	4349.6604	5576.4876
16	32009634	74.5094	4545.9815	5981.5545
17	48097624	74.4243	4850.6912	6299.5990
18	60000123	74.4812	4443.4159	5846.5999
19	120000000	74.4252	4600.2593	6052.9728
20	150000000	74.3383	4952.7747	6432.1750

We have computed the total execution time for the 10000 iterations corresponding to each of the 20 dimensions of the generated sequences. Using an energy consumption meter device (Voltcraft Energy Logger 4000) we have measured the system's power (kW) and the total energy consumption in each of the three analyzed cases (running the tests on the CPU and on the two GPUs). We have noticed that the system consumes 31.17 times less power when the test suite is run on the GTX 480 GPU compared to the i7-2600K CPU. The power consumption is better for the GTX 680 GPU than for the i7-2600K CPU, beeing 52.98 times lower.

We have analyzed the obtained experimental results when running our random number generator's CUDA implementation on the two GPUs and a random number generator on the i7-2600K CPU, when the output sequence has a relatively low dimension (35-1500 elements). We have noticed that the CPU has recorded the lowest execution time (**Figure 3**) and the highest bandwidth (**Figure 4**) because the processing amount, needed to generate the random elements, was

not high enough so that the graphic processing units could employ their huge parallel processing power.

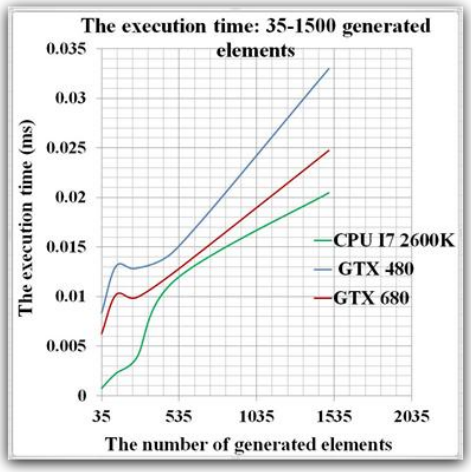

Figure 3. The execution time for 35 - 1500 elements of the output array

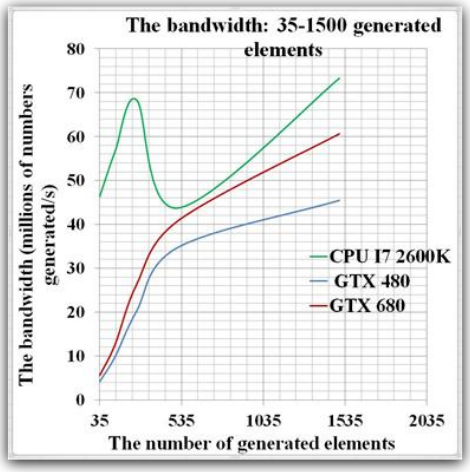

Figure 4. The bandwidth for 35 - 1500 elements of the output array

When we have generated a large dimension output array, ranging from 30000 to 150000000 elements, the GPUs have recorded the lowest execution times and the highest bandwidths. We have obtained the best performance on the GTX 680 GPU and then on the GTX 480, a performance that has overwhelmingly surpassed that of the CPU (**Figure 5**, **Figure 6**). In this case, the graphic processing units benefitted from the high computational amount needed to generate the random elements and fully employed the 512/256 allocated number of threads per block.

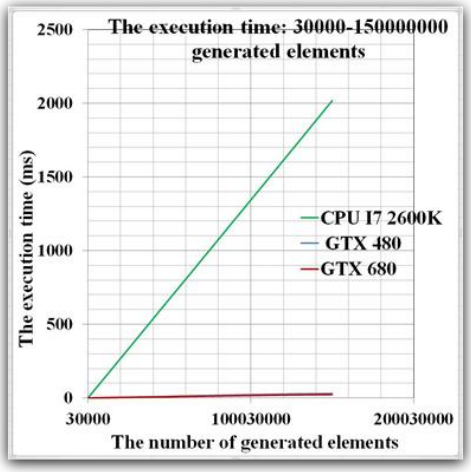

Figure 5. The execution time for 30000-150000000 elements of the output array

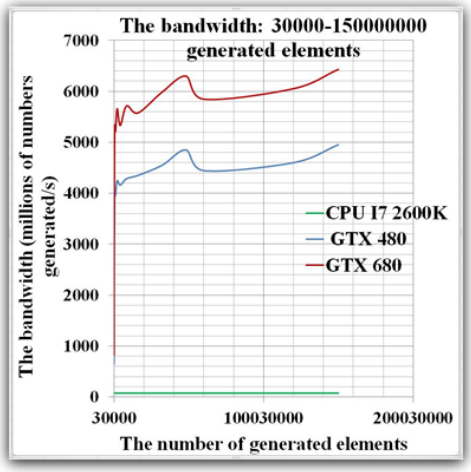

Figure 6. The bandwidth for 30000-150000000 elements of the output array

We have designed our algorithm to generate float or double data types output elements. In the following, we have researched if the generated data type influenced the performance of our random number generator's CUDA implementation. We have analysed the results obtained on the Kepler GK104 architecture (GTX 680), generating an input array ranging from 35 to 150000000 elements. For measuring accurate results, we have computed the average of 10000 iterations.

We have highlighted the execution time (**Figure 7**) and the bandwidth (**Figure 8**) variation depending on the output vector's data types. We have obtained the best results (lower execution time, higher bandwidth) when generating float data type elements, that is justified taking into account the amount of necessary memory to store the analyzed data types. The high level of performance registered in both cases, reflects the efficiency of our random number generator's implementation, using graphics processing units that provide a reduced execution time and a high data processing speed, regardless of the considered data type.

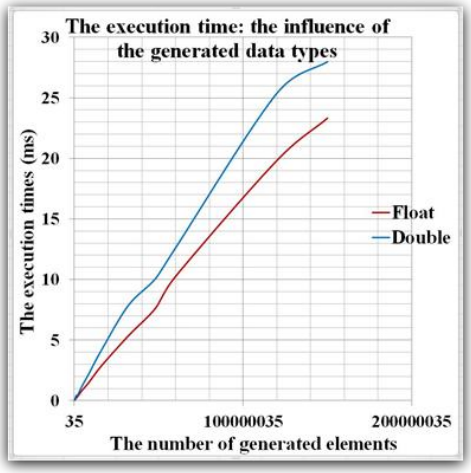

Figure 7. The influence of the generated data types on the execution time

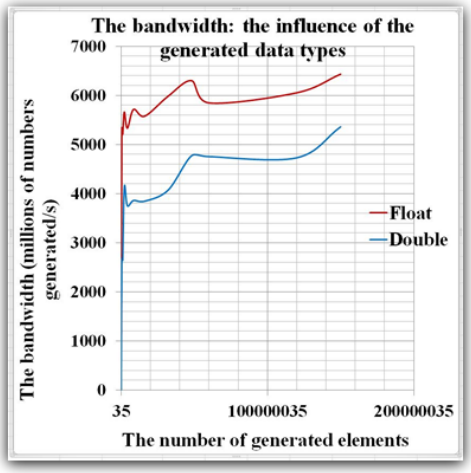

Figure 8. The influence of the generated data types on the bandwidth

Analysing the previous obtained results, we have concluded that our random number generator, implemented in the Compute Unified Device Architecture, provides a high level of performance, covering a broad range of scenarios, being a useful tool in applications that require random number generating.

6. Conclusions

We have focused our research on developing an efficient implementation of the Mersenne Twister random number generator algorithm in CUDA, using different optimization solutions. After analysing the Mersenne Twister's main execution steps, we have developed and implemented the algorithm in the Compute Unified Device Architecture. In this purpose, we have developed and applied a series of optimization solutions for our CUDA implementation (**Table 3**).

Table 3. Solutions for optimizing the Mersenne Twister algorithm's implementation in CUDA

No.	The Optimization Solution
Solution 1	Balancing the computational load, using multiple simultaneous Twisters processed in parallel
Solution 2	Optimizing the distribution of tasks to each of the threads
Solution 3	Minimizing the number of used registers
Solution 4	Shared memory usage
Solution 5	Managing shared memory banks conflicts
Solution 6	Using the warp shuffle operation

Solution 7	Managing the synchronization of parallel tasks
Solution 8	Using multiple thread blocks
Solution 9	Minimizing the number of executed instructions
Solution 10	Minimizing data transfer from/to external memory
Solution 11	Using the CPU instead of the GPU to generate low dimension output vectors

Our target was to fully benefit from the graphic processing units' huge parallel computational power. In order to achieve this, we have gradually improved and optimized the solutions after each experimental test.

We have compared the performance of our Mersenne Twister random number generator algorithm's CUDA implementation with the performance obtained by a CPU implementation of the algorithm. We have used the CUDA API in order to measure with outmost accuracy the GPUs' average execution time and bandwidth. We have analysed the obtained experimental results and remarked the following facts:

- The system consumes 34.75x less energy when generating numbers of float data type on the GTX 480 graphics processor, than it does in the case of the Intel i7-2600K CPU, while the GTX 680 GPU determines a 52.98x lower system power consumption than the CPU.
- When we have generated on the GTX 680 graphics processing unit large dimension float data types arrays, ranging from 30000 to 150000000 elements, we have recorded significantly lower execution times and bandwidth. We have managed to obtain an improvement in performance of up to 86.53x compared to CPU, regarding both the execution time (23.32 ms versus 2017.80 ms) and bandwidth (6432.18 millions versus 74.34 millions of numbers generated/s). The CPU has recorded the lowest execution time and the highest bandwidth when the dimension of generated arrays varies between 35 and 1500 elements. This happened because the processing amount, needed to generate the random elements, was not high enough so that the graphic processing units could employ their huge parallel processing power. This is the reason why we have chosen as an optimization solution to use the CPU instead of the GPU to generate low dimension output vectors and to use the GPU to generate large dimension data volumes.
- We have obtained the best results (lower execution time, higher bandwidth) when generating float data type elements on the GTX 680 processor and we have recorded an improvement of up to 2x in both execution time (0.067255084 ms versus 0.134510168 ms) and bandwidth (5333.62 millions versus 2666.81 millions of numbers generated/s) compared to the generation of double data type elements. By analysing these results, we have found that, regardless of the

generated data type, the efficiency of our CUDA algorithm's implementation is confirmed.

- The results that we have obtained allow us to conclude that our random number generator, implemented in the Compute Unified Device Architecture, provides a high level of performance, covering a broad range of scenarios, being a useful tool in applications that require random number generating.
- Through this research, we were able to develop optimization solutions that apply to a wide class of graphics processing units covering all the major compute capabilities (1.x, 2.x, 3.x). In the same time, we have developed specific optimization solutions targeted towards the Kepler GK104 architecture, covering the devices with compute capability 3.x.

There has been a lot of interest in the literature lately for implementing random number generators on parallel architectures, but as far as we know, there have not been any scientific papers that developed specific optimization solutions that improve the performance of random number generators using CUDA devices of 3.x compute capability. We have noticed that, with a proper set of optimization solutions, the latest Kepler GK104 architecture offers a tremendous performance when generating random numbers.

We must mention that both graphic cards used in our research are targeted towards rendering video games, high performance scientifically computations being of secondary importance for these home targeted graphic cards. The Quadro series, on the other hand, are graphic cards targeted especially towards high performance scientifically computations, but due to their huge price and scarce availability we had to use in our research the GTX 480 and the GTX 680 GPUs. The performance that we managed to obtain far exceeds the one of the Sandy Bridge i7-2600K central processing unit.

We have paid particular attention in obtaining a random number generator solution in CUDA that self-adjusts and self-configures the number of thread blocks, number of threads per block, number of generated elements per block depending on the GPU's architecture, with values that provide an optimum performance. The obtained results confirm that our random number generator implemented in CUDA provides a high level of performance, covering the most important CUDA enabled GPUs architectures up to date: the GF100 Fermi and the GK104 Kepler. These aspects prove the efficiency and usefulness of our optimization solutions that we have specifically developed for the Mersenne Twister random number generator algorithm's CUDA implementation. Therefore, our random number generator algorithm's implementation in CUDA becomes an useful tool for a wide range of applications that make use of random number generation.

Without a doubt, the Compute Unified Device Architecture offers a tremendous potential in overcoming the computational limits of today's central processing units architectures, facilitating the development of optimization

solutions for data processing that determine a significant improvement in the performance and energy efficiency in the entire computing industry.

REFERENCES

[1] **Nvidia Corporation (2012),** ***CUDA C Programming Guide.*** Version 5.0, Nvidia Whitepaper, 14-124;
[2] **Sanders J., Kandrot E. (2010),** ***CUDA by Example: An Introduction to General-Purpose GPU Programming.*** Addison-Wesley Professional, 150-280;
[3] **Hwu W. W. (2011)**, ***GPU Computing Gems Jade Edition*****.** Morgan Kaufmann, 50-120;
[4] **Matsumoto M., Nishimura T. (1998)**, ***Mersenne Twister: a 623-Dimensionally Equidistributed Uniform Pseudo-random Number Generator*****.** ACM Transactions on Modeling and Computer Simulation 8 (1), 3–30;
[5] **Podlozhnyuk V. (2007)**, ***Parallel Mersenne Twister*****.** Nvidia Corporation Tutorial, 1-8;
[6] **Matsumoto M., Kurita Y. (1992),** ***Twisted GFST Generators.*** ACM Trans. Model. Comput. Simul. 2, 179-194;
[7] **Matsumoto M., Kurita Y. (1994),** ***Twisted GFST Generators II.*** ACM Trans. Model. Comput. Simul. 4, 251-266;
[8] **Lewis T.G., Payne W.H. (1973),** ***Generalized Feedback Shift Register Pseudorandom Number Algorithms.*** J. ACM 20, 456-468;
[9] **Matsumoto M., Nishimura T. (1998),** ***Dynamic Creation of Pseudorandom Number Generators***. Proceedings of the Third International Conference on Monte Carlo and Quasi-Monte Carlo Methods in Scientific Computing, 56-69;
[10] **Griebel M., Knapek S. , Zumbusch G. (2010)**, ***Numerical Simulation in Molecular Dynamics: Numerics, Algorithms, Parallelization, Applications***. Springer, 70-110;
[11] **Pirjan A. (2010)*****, Improving Software Performance in the Compute Unified Device Architecture.*** Informatica Economica Journal, 14, 4, 30-47.