# Impact of the Random Number Generator Quality on Particle Swarm Optimization Algorithm Running on Graphic Processor Units

C. J. A. Bastos-Filho, M. A. C. Oliveira Junior, D. N. O. Nascimento
*Polytechnic School of Pernambuco, University of Pernambuco*
*Recife, Brazil*
*carmelofilho@dsc.upe.br*

A. D. Ramos
*Department of statystics*
*Federal University of Pernambuco*
*Recife, Brazil*

***Abstract*—Particle swarm optimization (PSO) is a bio-inspired technique widely used to solve real optimization problems. In the recent years, the use of Graphics Processing Units (GPU) has been proposed for some general purpose computing applications. Some PSO implementations on GPU were already proposed. The major benefit to implement the PSO for GPU is the possibility to reduce the execution time. It occurs due to the higher computing power presented nowadays on GPUs plataform. A study on the impact of the quality of Random Number generator has been made but it only covered some variations of the algorithm on a sequential plataform. In this paper, we present an analysis of the performance of the random number generator on GPU based PSOs in terms of the RNG statistical quality. We showed that the Xorshift random number generator for GPU presents enough quality to be used by the PSO algorithm.**

***Keywords*-Particle swarm optimization; Random number generator; GPU computing.**

## I. Introduction

Particle swarm optimization (PSO) is a stochastic, population-based global optimization technique proposed by Kennedy and Eberhart. It is inspired by the social behavior of bird flocking [1]. Each particle in the swarm represents a solution in a high-dimensional space and adjusts its position in the search space based on the best position reached by itself and on the best position reached by its neighborhood during the search process.

Some random variables included in the PSO are used to initiate the positions and the velocities of the particles and to evaluate the velocity update equation as well. These random variables are generated by a random number generator (RNG) and help to guide a more effective exploration during the search process.

In recent years, Graphic Processing Units (GPU) have appeared as a possibility to speed up general-purpose computing applications. Because of its parallel computing mechanism and fast float-point operation, GPU were applied successfully to many applications. Some examples of GPU applications are physics simulations, financial engineering, and video and audio processing.

PSO is inherently parallel and naturally suitable for a parallel implementation. However, most of the classic random number generators can not be executed concurrently, since they need the definition of a current state and a seed to run. Moreover, these RNGs also need to generate sequentially a long sequence of numbers to avoid poor statistical properties.

Most of GPU-based PSO approaches already proposed pre-generate all random numbers needed to run the algorithm. Some of them generate the numbers on the CPU, increasing the overall time to run the algorithm. And even when the numbers are generated directly on the GPU, they all need to be stored on the device memory.

A recent study published by Bastos-Filho *et al.* [2] showed that the RNG used in the PSO algorithm needs a minimum quality to guarantee that the PSO algorithm will perform well. However, the results presented in this paper only covered PSO algorithms running on a sequential architecture. Since the methods to efficiently generate the random numbers on GPU are quite different, it is relevant to analyze the statistical properties of fast RNGs for GPUs and compare them to classic RNGs. Furthermore, it is important to carry out some PSO simulations on GPU plataforms and measure the impact of the quality of these RNG in the PSO algorithm performance. The results of these analysis are presented in this paper.

This paper is organized as follows. Section II shows an overview of the related works. The implementation used in this paper to generate random numbers on GPUs is presented in the section III. The simulation setup and the results obtained are shown in the section IV and V. The analysis of the RNG statistical quality is presented in the section IV while the analysis on the performance of the PSO running in GPUs is presented in the section V. In the last section, we give our conclusions and suggest some future works.

## II. Background

In the follow subsections, we present the background for our research. In the first subsection, the basic particle swarm optimization concepts are detailed. Some concepts and examples of random number generators are explained in the second subsection. An overview about GPU Computing is presented in the third subsection. In the last subsection, some GPU-based PSO implementations are reviewed.

## A. Particle Swarm Optimization

The original PSO algorithm was first described in 1995 by James Kennedy and Russel C. Eberhart [1]. The algorithm is suitable for searches and is inspired by the behavior of flocks of birds looking for food. Each particle in the swarm represents a point in the fitness function domain. Each particle $i$ contains four vectors: its current position in the $D$-dimensional space $\vec{x_i} = (x_{i1}, x_{i2}, ..., x_{iD})$, its best position found so far $\vec{p_i} = (p_{i1}, p_{i2}, ..., p_{id})$, the best position found by its neighborhood so far $\vec{n_i} = (n_{i1}, n_{i2}, ..., n_{id})$ and its velocity $\vec{v_i} = (v_{i1}, v_{i2}, ..., v_{id})$ [3]. The position and the velocity of every particle are updated iteratively according to its current best position $p_i$ and the best position of its neighborhood $n_i$ by applying the following update equation for each particle:

$$v_{id} = v_{id} + c_1 \cdot r_1 \cdot (p_{id} - x_{id}) + c_2 \cdot r_2 \cdot (n_{id} - x_{id}), \quad (1)$$

$$x_{id} = x_{id} + v_{id}, \quad (2)$$

where in the equation (1) the learning factors $c_1$ and $c_2$ are non negative constants. They are the cognitive and the social acceleration constants, respectively. They are used to weight the contribution of the cognitive and the social components. The values $r_1$ and $r_2$ are two different random variables generated at each iteration for each dimension from a uniform distribution in the interval [0,1].

Particles velocities can be clamped to a maximum value in order to prevent an explosion state. Their positions are also bounded by search space limits in order to avoid inutile exploration of space.

A parameter $\chi$, known as the constriction factor, was proposed by Clerc [4]. It was designed to adjust the influence of the previous particle velocities on the optimization process. It helps to switch the search mode of the swarm from exploration to exploitation during the search process. The constriction factor $\chi$ is defined according to the following equation:

$$\chi = \frac{2}{|2 - \varphi - \sqrt{\varphi^2 - 4\varphi}|}, \ \varphi = c_1 + c_2. \quad (3)$$

This constriction factor is applied to the entire velocity update equation as shown in the following equation:

$$v_{id} = \varphi \cdot [v_{id} + c_1 \cdot r_1 \cdot (p_{id} - x_{id}) + c_2 \cdot r_2 \cdot (n_{id} - x_{id})]. \quad (4)$$

At each iteration, the algorithm updates the velocity and the position of each particle by using the equations above. Originally, the $\vec{n_i}$ is obtained after all particles have been totally updated. In contrast, another approach, known as asynchronous PSO, updates the $\vec{n_i}$ based on the current status of the swarm. It means that a particle can adjust its $\vec{n_i}$ before some other particles $j$ update their $\vec{n_j}$. An iteration of this approach is faster than the synchronous one due its absence of a synchronization barrier to guarantee that the current best used by the swarm is actually the best one.

## B. Random Number Generators

Random number generators (RNGs) are systems with the ability to generate sequences of random numbers according to a probability density function. A finite sequence of number is a random number sequence if there is no description of the sequence shorter than the sequence itself [5].

Some algorithms can be used as pseudo-random numbers generators (PRNGs) when implemented in a computational device [2]. A PRNG generates a sequence of numbers by using a deterministic algorithm and uses it as a random number sequence. Despite this deterministic construction, numbers generated by mathematical algorithms are good enough for the majority of application because of their good statistical properties.

Two well known implementations of pseudo-random numbers generators are presented in the next two subsections: the Linear Congruential Generator (LCG) and the George Marsaglia's generator.

*1) Linear Congruential Generator:* Linear Congruential Generator (LCG) is a classical random number generator that is widely used and implemented in many mathematical standard libraries of programming languages, such as C and Java. Let $a$, $c$, $m$ be integer constants and $X_0$ the initial number of the sequence known as seed, the next number of the LCG sequence is defined by:

$$X_N = (a \cdot X_{N-1} + c) \ mod \ m. \quad (5)$$

The LCG is very sensitive to the $a$ (multiplier), $c$ (increment) and $m$ (modulus) parameters. The $m$ value defines the period of the generator.

*2) George Marsaglia's Generator:* The algorithm is based on two binary operation described by equations (7) and (8).

$$x \bullet y = \text{if } x \geq y \text{ then } (x - y), \text{ else } (x - y + 1); \quad (6)$$

$$c \circ d = \text{if } c \geq d \text{ then } (c - d), \text{ else } (c - d + r). \quad (7)$$

The original method requires 97 initial seed values $x_1$, $x_2$, ..., $x_{97}$ and a parameter $c_1$ to generate the next seeds according to equations (9) and (10). A sequence of random numbers ($U_1$, $U_2$, $U_3$, ...) can be obtained by using equation (11).

$$x_n = x_{n-97} \bullet x_{n-33}, \quad (8)$$

$$c_n = x_{n-1} \circ r, \quad (9)$$

$$U_n = x_n \bullet c_n. \quad (10)$$

## C. GPU Computing

GPU parallel computing follows an architecture called SIMT (Single Instruction – Multiple Threads) [6]. In the SIMT paradigm, the same instruction set is executed on different data processors at the same time. In our case, the data processors are on the GPUs. The GPU SIMT architecture presents less overhead in parallel computing, which is suitable to intensive and repetitive computation as found in parallel PSO.

GPU were traditionally designed for image and graphic processing on computers and is a device capable to execute a very high number of threads in parallel. The use of GPUs became popular in recent years through the CUDA (Compute Unified Device Architecture), which is a technology that enables programmers and developers to write software to solve complex computational problems by automatically distributing these thread to many-core processors on the GPUs.

CUDA's parallel programming model allows the programmer to divide the main problem in many sub-problems that can be executed independently in parallel. Each sub-problem, can be decomposed in many modules that can be executed cooperatively in parallel. In CUDA's terminology, each sub-problem corresponds to a block of threads, where each thread is a module. The function that is performed to solve the sub-problem is called kernel. When a kernel is invoked by the CPU, it runs on each thread within the corresponding block.

Each of the threads that execute a kernel is identified by its thread ID that is accessible within the kernel through the built-in *threadIdx* variable.

Threads within a block can cooperate among themselves by sharing data through some shared memory and synchronizing their execution to coordinate memory accesses. Each thread has a private local memory. Each block of threads has a shared memory visible to all threads of the block and with the same lifetime of the block. In the recent CUDA architectures, all the threads can access the same global memory, though this type of access is more time consuming. In this paper, we mainly use the shared memory and global memory for our implementation.

### D. GPU-based Particle Swarm Optimization

Few parallel PSO implementations have been developed for GPUs. All of them showed a great speed-up performance when compared to the CPU version. Despite of this, none of these studies presented any analysis on the dependence of the PSO performance with their random number generators.

Some of these implementations pre-generate all random numbers needed to run all iterations of the algorithm on the host [7], [8]. This approach increases the overall run time to transfer the random numbers from the host (CPU) to the GPU device.

A GPU version of the Mersenne Twister generator [9], [10] is used in some PSO implementations [11], [12]. GPU-based random numbers generators are discussed on [13]. Although they run on the device, they are not CPU-free since they need to generate the initial seeds on the host.

Nevertheless, all approaches analyzed need to store the random numbers generated on the device memory. Therefore, the algorithm is not free to run for many iterations, since the numbers are not generated on demand and the memory space on the GPU device is limited.

## III. Our GPU Random Number Generator Implementation

Our random number generator implementation is based on the Xorshift RNG [14], but it does not have a state as the original version. Consequently, it does not need any initial seeds from the host and it is completely CPU free. Our implementation on CUDA is presented on Code 1.

**Code 1** The random number generator on CUDA.

```
__device__ int randomNumber(int id) {
    int n;
    n = clock() + clock()*id;
    n = n^(n << 21);
    n = n^(n >> 35);
    n = n^(n << 4);
    if (n < 0)
        n = -n;
    return n;
}
```

The operators $<<$ and $>>$ are shift operators which shift their first operand left ($<<$) or right ($>>$) by the number of positions specified by the second operand. The operator ˆ is the bitwise-exclusive-OR operator. And the function $clock()$ returns the current value of a per-multiprocessor counter on the GPU [6].

Each thread calls $randomNumber()$ with its thread index as a parameter. Due to the fact that the random numbers generators are running in each thread independently and they do not store any state on the device, the numbers can be generated on demand by each particle.

## IV. Random Number Generators Analysis

As we are proposing to use Xorshift RNG for GPU based PSO, we need to evaluate its statistical properties. We compared the random number generator on CUDA with five other generators. As in [2], we used the following RNGs:

1. Marsaglia – Marsaglia's Generator using four seeds. It is the best random number generator used in this paper.

2. JAVA – Generator used by JAVA based on Linear Congruential Generator (LCG).

3. LCG Excellent – A high quality Linear Congruential Generator using $m$ = $2^{31} - 1$, $a$ = 163490618 and $c$ = 0.

4. LCG Good – A medium quality Linear Congruential Generator using $m$ = $2^{32}$, $a$ = 398347535 and $c$ = 0. It is not as good as the LCG Excelent because it uses a square power $m$ value.

5. LCG Bad – A poor quality Linear Congruential Generator using $m$ = $2^{32}$, $a$ = 1 and $c$ = 1. It generates a sequential series with regular intervals.

For each RNG we will perform our analysis based on 100 sequences where each sequence has 10240 random numbers. Each sequence is initialized with a different seed.

For each one of these sequences, we evaluated the mean, the standard deviation and the variance after we evaluated the mean of each one of these statistical measures. These results are shown on the Table I. One can note similarities between the CUDA generator and the others using these criteria. LCG Bad presented a slightly higher difference between the mean value and the expected value $(0.5)$ and a slightly lower difference between the obtained variance (the square of the standard deviation) and the expected value for the variance $(1/12)$.

Table I
MEAN, STANDARD DEVIATION, AND VARIANCE OF THE SEQUENCES FOR 5 DIFFERENT RNGS

| **RNG** | Mean | Std Deviation | Variance |
|---|---|---|---|
| CUDA | 0.4995743 | 0.288633 | 0.332882 |
| JAVA | 0.4996777 | 0.288728 | 0.333040 |
| LCG excellent | 0.4999714 | 0.288593 | 0.333257 |
| LCG good | 0.5001353 | 0.288593 | 0.333257 |
| LCG bad | 0.5064017 | 0.079705 | 0.336146 |
| Marsaglia | 0.5004785 | 0.288747 | 0.333854 |

However, Mean, Standard Deviation, and Variance are not sufficient to determine the randomness of a RNG. Therefore, we propose to use the correlation between the sequences of numbers generated by the RNGs. The correlation of two variables represents the correspondence between them. It is important to have this value to show if different sequences generated by the same generators are independent. The correlation value is in the interval [-1,1] and the more the value is near to zero the more the sequences are independent. Once they have correlation near to one or near to minus one, the particles on PSO probably will be initialized or updated by the same way on different runs.

Since the sequences generated by the LCG Bad are arithmetic progressions with the seed as the initial term. And due to the fact that every sequence is generated using a different seed, the correlations between them tend to be zero. Thus, the LCG Bad are not included on this analysis.

We evaluated the correlations of the 4950 combinations of the 100 sequences of each generator. The mean, standard deviation, variance and kurtosis are presented on the Table II. The histograms of the correlation achieved for each RNG are quite similar. Fig. 1 and Fig. 2 present the histogram of the correlation of the numbers sequences generated by the RNG on CUDA and JAVA, respectively. These results suggest that the histogram show a great concentration near to zero.

These results show that CUDA random number generator have a similar behavior when compared to the other classical RNGs. All the RNGs have correlations near to zero. It means that the number sequences generated by the RNGs does not depend with one another. Thus, the PSO algorithm will be updated and initialized differently for each run.

Table II
MEAN, STANDARD DEVIATION, VARIANCE AND KURTOSIS OF CORRELATIONS BETWEEN THE 100 SEQUENCES GENERATED BY THE EACH RANDOM NUMBER GENERATORS

| **RNG** | Mean | Variance | Kurtosis |
|---|---|---|---|
| CUDA | $-5.21872E-5$ (0.00988) | $9.75709E-5$ | $-9.08668$ |
| JAVA | $-2.5561E-5$ (0.00999) | $9.98481E-5$ | $-7.62273$ |
| LCG excellent | $-2.65154E-5$ (0.00988) | $9.75869E-5$ | $-5.69151$ |
| LCG good | $2.50877E-5$ (0.00989) | $9.78822E-5$ | $-1.06406$ |
| Marsaglia | $1.44787E-4$ (0.00989) | $9.77466E-5$ | $-7.42308$ |

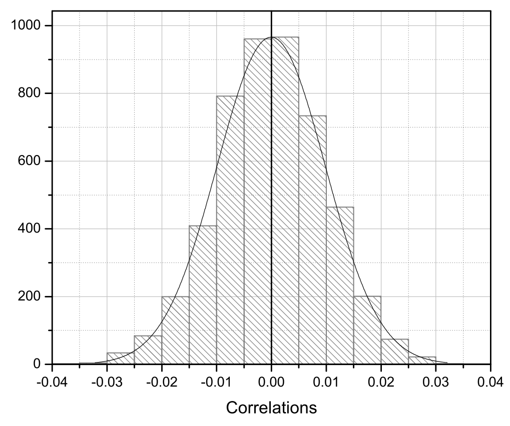

Figure 1. Histogram of the correlation of the numbers sequences generated by the RNG on CUDA.

## V. GPU-BASED PSO ANALYSIS

In this section we present some analysis of the PSO performance running on CUDA (synchronous and asynchronous) and on JAVA (sequential). In all the simulations, we performed 30 trials per function, where 10,000 iterations were executed for each trial in a 30 dimensions search space. We used 30 particles in all simulations. The total number of fitness functions evaluations per trial was 300,000. We used the constricted velocity update equation with $c_1$ = 2.05 and $c_2$ = 2.05.

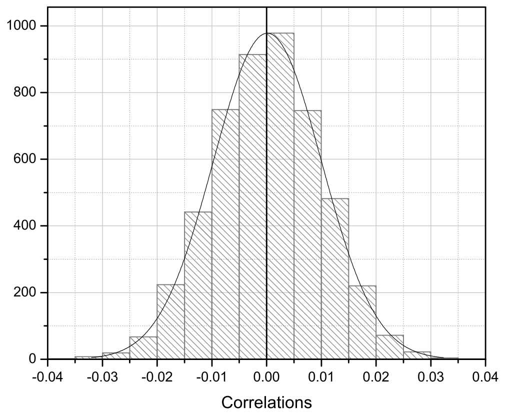

Figure 2. Histogram of the correlation of the numbers sequences generated by the RNG used on Java language.

The following benchmarks functions are all minimizing problems and were used in our experiments.

$$F_{Rastrigin}(\vec{x}) = 10n + \sum_{i=1}^{n} \left[ x_i^2 - 10cos(2\pi x_i) \right] \tag{11}$$

$$F_{Griewank}(\vec{x}) = 1 + \sum_{i=1}^{n} \frac{x_i^2}{4000} - \prod_{i=1}^{n} cos\left(\frac{x_i}{\sqrt{i}}\right) \tag{12}$$

$$F_{Schwefel}(\vec{x}) = \sum_{i=1}^{n} \left( \sum_{j=1}^{i} x_j \right) \tag{13}$$

$$F_{Rosenbrock}(\vec{x}) = \sum_{i=1}^{n} x^2 \left[ 100(x_{i+1} - x_i^2)^2 + (1 - x_i)^2 \right] \tag{14}$$

Table III presents a comparison of the average and the (standard deviation) of the fitness for Rosenbrock, Rastrigin, Griewank and Schwefel functions obtained by the different PSO versions. As can be seen, the results for CUDA Synchronous and JAVA are quite similar. It is expected since they are the same algorithm with a different RNG. Thus, it indicates that the RNG is not influencing the results. However, when we compare the results for CUDA synchronous and CUDA asynchronous, we observe a little worse performance for the last one. It is also expected, since some data can not be considered during the velocity update processes due to the lack of synchronization barriers. Fig. 3 and 4 show the convergence curves for the Rosenbrock and Griewank functions, respectively.

Fig. 5 and 6 show the elapsed time for the PSO execution for Rosenbrock and Griewank functions, respectively. One can note that the elapsed time for the two parallel algorithms (CUDA) have shown better performance than the sequential algorithm for all the functions. The asynchronous version for CUDA is slightly faster than the synchronous version.

Table III
AVERAGE VALUE AND (STANDARD DEVIATION) OF THE FITNESS AFTER 50 TRIALS OF 10,000 EVALUATIONS

| | JAVA | CUDA Synchronous | CUDA Assynchronous |
|---|---|---|---|
| Rosenbrock | **3.79 (9.77)** | 4.28 (9.89) | 5.47(13.97) |
| Rastrigin | 55.06 (14.28) | **52.97 (12.66)** | 54.34 (15.58) |
| Griewank | 0.036 (0.096) | **0.029 (0.048)** | 0.067 (0.102) |
| Schwefel | $2 \cdot 10^{-18}$ $(4 \cdot 10^{-18})$ | $6 \cdot 10^{-18}$ $(10^{-17})$ | $4.4 \cdot 10^{-12}$ $(1.2 \cdot 10^{-11})$ |

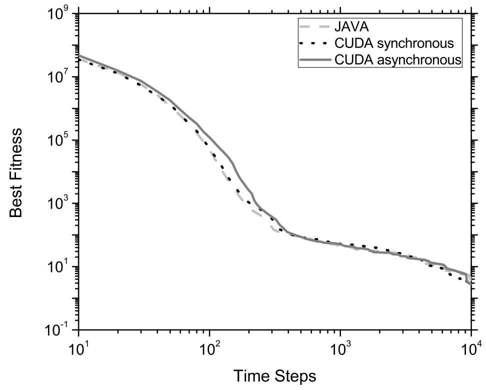

Figure 3. Convergence curve for Rosenbrock function.

## VI. CONCLUSIONS

We presented in this paper an investigation about the influence of the quality of the Random Number Generators on the performance of Particle Swarm Optimization algorithms running on Graphic Process Units. The simulation results show that the Random Number Generator (RNG) suggested in this paper can provide the minimum quality required by the PSO. We also showed that the algorithm running on GPU present a similar performance to find the optima when compared to the sequential version, mainly for the synchronous version. The asynchronous version for the GPU based PSO is slightly faster than the synchronous version. Both GPU based PSO version are much faster than the sequential PSO. Our future research will focus on implementing other Particle Swarm Optimization topologies and check the impact of the RNG in these variations.

## ACKNOWLEDGMENT

The authors would like to thank FACEPE, CNPq and UPE.

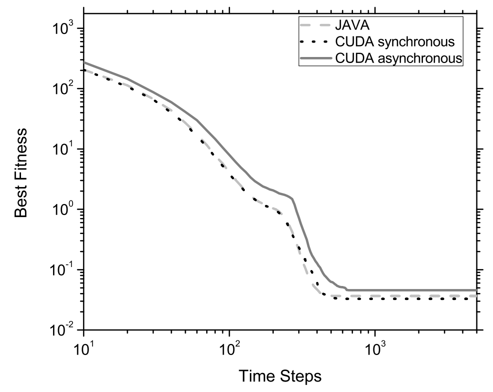

Figure 4. Convergence curve for Griewank function.

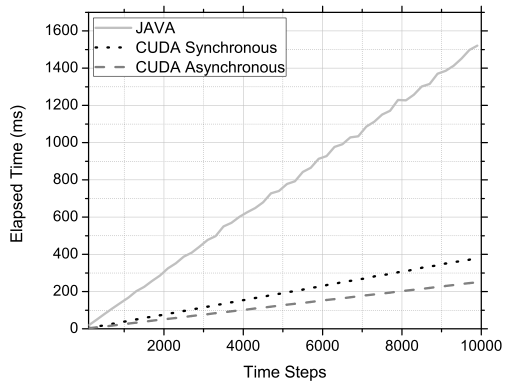

Figure 5. Elapsed time for the PSO execution for Rosenbrock function.

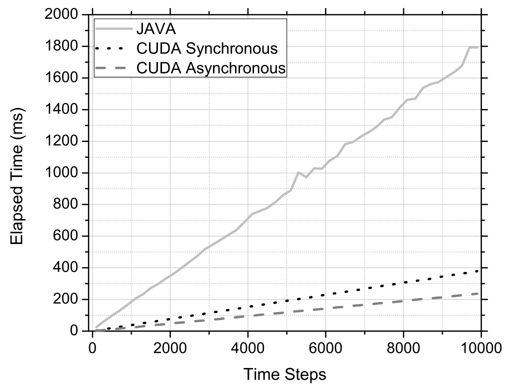

Figure 6. Elapsed time for the PSO execution for Griewank function.

## REFERENCES

[1] J. Kennedy and R. Eberhart, “Particle swarm optimization,” *Neural Networks, 1995. Proceedings., IEEE International Conference on*, vol. 4, pp. 1942 –1948 vol.4, nov/dec 1995.

[2] C. Bastos-Filho, J. Andrade, M. Pita, and A. Ramos, “Impact of the quality of random numbers generators on the performance of particle swarm optimization,” *Systems, Man and Cybernetics, 2009. SMC 2009. IEEE International Conference on*, pp. 4988 –4993, oct. 2009.

[3] D. Bratton and J. Kennedy, “Defining a standard for particle swarm optimization,” *Swarm Intelligence Symposium, 2007. SIS 2007. IEEE*, pp. 120 –127, april 2007.

[4] M. Clerc and J. Kennedy, “The particle swarm - explosion, stability, and convergence in a multidimensional complex space,” *Evolutionary Computation, IEEE Transactions on*, vol. 6, no. 1, pp. 58 –73, feb 2002.

[5] M. Matsumoto, M. Saito, H. Haramoto, and T. Nishimura, “Pseudorandom number generation: Impossibility and compromise,” *Journal of Universal Computer Science*, vol. ”12”, no. ”6”, pp. ”672–690”, ”2006”.

[6] NVIDIA, *NVIDIA CUDA Programming Guide 2.3*, 2009.

[7] Y. Zhou and Y. Tan, “Gpu-based parallel particle swarm optimization,” *Evolutionary Computation, 2009. CEC ’09. IEEE Congress on*, pp. 1493 –1500, may 2009.

[8] D. B. Thomas, L. Howes, and W. Luk, “A comparison of cpus, gpus, fpgas, and massively parallel processor arrays for random number generation,” New York, NY, USA, pp. 63–72, 2009. [Online]. Available: http://doi.acm.org/10.1145/1508128.1508139

[9] M. Matsumoto and T. Nishimura, “Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator,” *ACM Trans. Model. Comput. Simul.*, vol. 8, pp. 3–30, January 1998. [Online]. Available: http://doi.acm.org/10.1145/272991.272995

[10] V. Podlozhnyuk, “Parallel Mersenne Twister,” nvidia Corp., Tech. Rep., 2007.

[11] L. Mussi, S. Cagnoni, and F. Daolio, “Gpu-based road sign detection using particle swarm optimization,” *Intelligent Systems Design and Applications, 2009. ISDA ’09. Ninth International Conference on*, pp. 152 –157, 30 2009-dec. 2 2009.

[12] W. Zhu and J. Curry, “Particle swarm with graphics hardware acceleration and local pattern search on bound constrained problems,” *Swarm Intelligence Symposium, 2009. SIS ’09. IEEE*, pp. 1 –8, 30 2009-april 2 2009.

[13] H. Nguyen, *Gpu gems 3*. Addison-Wesley Professional, 2007.

[14] G. Marsaglia, “Xorshift rngs,” *Journal of Statistical Software*, vol. 8, no. 14, pp. 1–6, 7 2003. [Online]. Available: http://www.jstatsoft.org/v08/i14