Computational Science Technical Note **CSTN-122**

Performance and Quality of Random Number Generators

V. Du Preez and M. G. B.Johnson and A. Leist and K. A. Hawick

2011

Random number generation continues to be a critical component in much of computational science and the tradeoff between quality and computational performance is a key issue for many numerical simulations. We review the performance and statistical quality of some well known algorithms for generating pseudo random numbers. Graphical Processing Units (GPUs) are a powerful platform for accelerating computational performance of simulations and random numbers can be generated directly within GPU code or from hosting CPU code. We consider an alternative approach using high quality and genuinely "random" deviates generated using a Quantum device and we report on how such a PCI bus device can be linked to a CPU program. We discuss computational performance and statistical quality tradeoffs of this architectural model for Monte Carlo simulations such as the Ising system.

Keywords: quantum random number generation; GPU; CUDA

BiBTeX reference:

```
@INPROCEEDINGS{CSTN-122,
        author = {V. Du Preez and M. G. B.Johnson and A. Leist and K. A. Hawick},
        title = {Performance and Quality of Random Number Generators},
        booktitle = {International Conference on Foundations of Computer Science (FCS'11)},
        year = {2011},
        number = {FCS4818},
        pages = {16-21},
        address = {Las Vegas, USA},
        month = {18-21 July},
        publisher = {CSREA},
        keywords = {quantum random number generation; GPU; CUDA},
        owner = {kahawick},
        timestamp = {2011.05.16}
}
```

Technical Report CSTN-122

Performance and Quality of Random Number Generators

V. du Preez, M.G.B. Johnson, A. Leist and K.A. Hawick
Computer Science, Institute for Information and Mathematical Sciences,
Massey University, North Shore 102-904, Auckland, New Zealand
email: dupreezvictor@gmail.com, { m.johnson, a.leist, k.a.hawick }@massey.ac.nz
Tel: +64 9 414 0800 Fax: +64 9 441 8181

April 2011

ABSTRACT
Random number generation continues to be a critical component in much of computational science and the tradeoff between quality and computational performance is a key issue for many numerical simulations. We review the performance and statistical quality of some well known algorithms for generating pseudo random numbers. Graphical Processing Units (GPUs) are a powerful platform for accelerating computational performance of simulations and random numbers can be generated directly within GPU code or from hosting CPU code. We consider an alternative approach using high quality and genuinely "random" deviates generated using a Quantum device and we report on how such a PCI bus device can be linked to a CPU program. We discuss computational performance and statistical quality tradeoffs of this architectural model for Monte Carlo simulations such as the Ising system.

KEY WORDS
quantum random number generation; GPU; CUDA.

1 Introduction

The fast generation of good quality random numbers [1–6] is a long-standing challenge [7, 8]. Random numbers are needed for many applications, but are used in very large quantities in computer simulations that employ the Monte-Carlo algorithm [9, 10]. It is neither trivial nor computationally cheap to generate large sets of pseudo-random numbers that have the right statistical "randomness" needed to perform an unbiased calculation. Until recently it has not been practical to use random number generation hardware that was economically priced and suitably unbiased. Instead, pseudo random numbers that were generated from a suitable deterministic algorithm were employed. A great deal has been written in the literature about such algorithms, but for the most part there are many very good ones that are "random enough" and are at least uncorrelated with the application so that they suffice. One important area of use has been the numerical investigation of phase transitions and critical phenomena. In such work the Monte Carlo algorithm is used to sample appropriate points in a physical model space to simulate the actual dynamical behaviour of a model and identify the location of critical points – abrupt changes - that result when a model parameter such as temperature changes by a small amount.

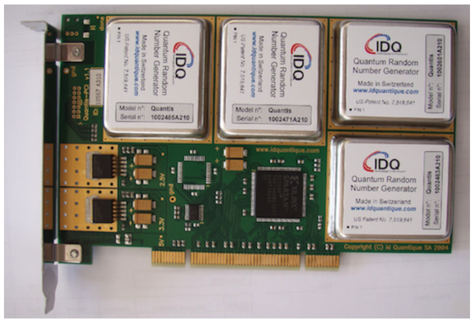
Figure 1: Quantis RNG Card for PCI Bus, showing four independent quantum generator devices.

This work is very demanding and a certain degree of caution is perceived in the reported literature as researchers go to great lengths to be sure their simulations are not overly biased by random numbers with unfortunate statistical properties.

Pseudo-random number generators are often formulated in terms of mathematical recurrence relations [11] whereby repeated application of a transformation will project a number to another in an apparently random or decorrelated sequence - at least to the extend that any patterns discernible in the resulting sequence are on a scale that is irrelevant to the application using them. Any random or pseudo random number generator delivers a sequence of random deviates - either floating point uniform deviates or integers or sometimes just plain bits. An application will use or consume deviates from such a sequence as it runs.

There are still some philosophically deep questions concerning what it really means for a sequence of deviates to be truly random. It is widely believed however that some quantum physical processes yield deviates that are as random as we can ever make them. Such devices are presently available as special purpose cards or external drivers that connect via a bus-based hardware interface such as PCIe or USB. We investigate the use of the ID Quantique "Quantis" device below in section 4. Intel and other CPU manufacturers [12] are now actively considering provision of random number generation hardware directly on the chip. This closeness to the arithmetic and logic hardware means that these devices will produce very fast deviates, and the expectation is that the thermal noise and quantum processes involved are sufficiently well understood at a statistical level to ensure that these sources are also unbiased.

Our paper is structured as follows: In Section 2 we review some key issues for random number generation. In Section 3 we briefly review the Ising model and associated Monte Carlo analysis algorithms as demanding consumers of random deviates. In Section 4 we describe some of the pseudo random number generator algorithms and implementation strategies we have explored. We present some performance and statistical test results for both algorithmically generated and quantum device generated sequences in Section 5. We discuss some of the implications for future generation simulation work and offer some conclusions and ideas for further study in Section 6.

2 Generation Algorithm Issues

Generally speaking there are two main criteria that are considered when choosing a pseudo-random number generator. The first is the period of the generated sequence. Ideally this should be so long as to never repeat during the life-cycle of the application. Modern generators – as we discuss here – usually have periods that are very long and that when run on current computer clock speeds have repeat times comparable with the lifetime of the known universe. In this sense the period is not often a direct concern.

A few deviates generated to make a game program behaviour "interesting" to a player does not require a generator with a challengingly long repeat length. However, Monte Carlo calculations that may take weeks or months of supercomputer resources must have generators with very long period lengths. In the last 20-30 years of steadily increasing supercomputer performance, there has been continued interest in ever longer period generator algorithms. This often ties in with the need for more bits used in the generator. The 16-bit integer based generators of the late 1970s, were superceded by 24-bit (floating-point) algorithms such as the Marsaglia lagged-Fibonacci algorithm [6], by the 64-bit integer based Mersenne-Twistor and in very recent times by 128-bit algorithms [13] and even longer for cryptographically strong random number generation [14].

The second criteria is more subtle however and has definitely been a known concern with some algorithms. This issue concerns just how actually random or uncorrelated the deviates in the sequence are – in the context of the needs of the driving application. There are some widely used statistical tests [15] that are now in wide circulation and which represent the research communities best wisdom on what is "random enough." We discuss these in section 4.

Applications usually need either random integers with a flat uniform probability of obtaining all values within a set range, or uniform floating point deviates in the range $[0, 1)$, again with a uniform probability distribution across the range. Generally if one has a generator that produces either of these, one can construct deviates of more sophisticated distributions with suitable transformation algorithms [16, 17].

The apparatus for implementing pseudo-random number generators usually give rise to raw deviates in one of those two forms - uniform integers or uniform floating point number and one can find ways of transforming one to the other. In the case of floating point deviates one can simply multiply by N to obtain integers on the $[0, N)$ range, and in the case of integers known to be in that range, one can divide by N. Different processing hardware will carry these operations out with different speeds depending on clock speeds and floating point standards. If one has a random source of uncorrelated b-bits [18, 19] one can readily obtain (unsigned) integers [20]. in a suitable range of $[0, 2^b)$ or $[0, 2^b - 1]$. One can then divide accordingly to obtain floating point uniform deviates. The reverse operation is not so simple however [18]. Most processors use the IEEE floating point standard bit representation for 32-bit or 64 bit precision. These specify sign bit, exponent and mantissa from which it is not trivial to obtain evenly unbiased random bits without some arithmetic that must necessarily waste some of the original 32 or 64.

This gives rise to another important criteria for random number generators - ideally they should be well engineered in terms of having plug-compatible software programming interfaces. This means that a code can be tested and implemented using any number of different generator algorithms with little code change required. A pragmatic implementer therefore finds it is often better to have a generator that produces unbiased integers or raw bits from which an unbiased unsigned integer can be constructed. It is often then easier to make a family of suitable software interfaces to supply all the sorts of deviates that are needed by applications from the one root algorithm.

In this present paper, we discuss Monte Carlo algorithmic uses of random numbers where we need a fast supply of good deviates. Another application is for cryptographic use, where usually the need for extreme speed is less, but the need for very high randomness - to the point of uncrackability is extreme [21].

For some applications it is actually desirable to have pseudo-random number generator that is repeatable – from the same starting conditions. Seeding generators is of course an interesting issue in itself and this problem is often exacerbated when using a parallel computer. While a generator algorithm may have a known very long period, often one has to run the generator many times or on parallel processors and the choice of seeds matters to avoid accidentally correlating the sub-sequences generated by each instance [22, 23]. Parallel computing applications such as parallel and supercomputer implementations of Monte Carlo simulations have been a target for many special purpose implementations of pseudo random number generators. Work has been done on: array processors [24]; vector computer architectures [25]; transputers using parallel Occam [26]; and more recently on specialist processors such as the Cell [27] or on Graphics Processing Units(GPUs) [28–30].

Techniques for generating seeds vary. When debugging an application it can be very helpful to be able to specify the same seed and ensure identical results. Once in production mode seeds might be generated by an algorithm based on precise time and dates or from special purpose hardware. Many operating systems will now support a hardware source via for example /dev/random on Unix based systems. This may supply bits from thermal noise or other sources. Such deviates are unfortunately not necessarily statistically unbiased nor necessarily particularly fast - but they certainly suffice for seeding a proven pseudo random algorithm that does have the required qualities.

Another approach which has only recently become economically feasible and which may become more widespread soon [31], is to have a hardware source of genuinely random numbers - that are drawn from some quantum physical phenomena [32] that is as random as we can imagine given our current understanding of the universe, and which therefore do not require a starting seed. Figure 1 shows a special purpose device, produced by Quantis, that generates around 16MBits/s that are – as we have

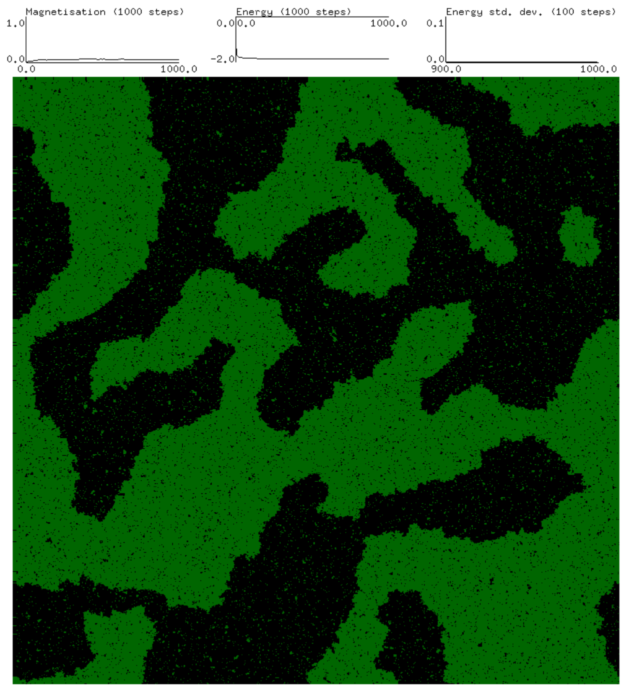

Figure 2: A 1024×1024 Ising model simulation with temperature $T = 2.0$ after 1000 simulation steps.

determined and discussed below – of superb quality.

3 Ising Model Applications

Monte Carlo simulations use random sampling to approximate results when it is infeasible or impossible to compute the exact result for a physical or mathematical system [33]. The Ising model [34–36] uses such a method to calculate the critical point of metal alloy phase transitions. The numbers in these systems need to be as close to truly random as possible to avoid bias in the results which may result in incorrect conclusions

Simulations of the Ising model typically start with a random “hot” system. The system is then quenched to a specific temperature. If this temperature is below a critical “cold” temperature, then the system undergoes a phase transition where like spin values begin to clump together, creating order in the initially random system. The Ising model has just two possible spin values, “up” and “down”, but can be extended to the Q-state Potts model [37] that uses Q spin values. A system quenched to a temperature very close to the critical temperature shows clusters of like-like spins on all possible length scales. Figure 2 illustrates a 2-dimensional Ising model simulation.

A number of different Monte-Carlo update algorithms for the Ising model have been proposed over time [38–41]. The Metropolis algorithm [38], which was later generalised by Hastings [42], has formed the basis for Monte-Carlo statistical mechanics [43, 44] and has been used widely for Ising model simulations [45–48]. It is a Marcov chain Monte-Carlo (MCMC) method, where the transitions from one state to the next only depend on the current state and not on the past. Using the Metropolis update algorithm for the Ising model simulation, at each discrete time step, a new system configuration is chosen at random by picking a spin to "hit" and flipping its value. If the energy E of the proposed configuration is lower than or equal to the current energy, $\Delta E \leq 0$, then the move to the new configuration is always accepted. Otherwise, the new configuration is accepted with probability $exp(-\Delta E/k_B T)$, where T is the temperature and k_B is the Boltzmann constant. The current configuration is retained if the move is rejected.

The spins in the Ising model interact with their nearest neighbours according to an energy function or Hamiltonian of the form: $H = -\sum_{\langle i,j \rangle} J_{ij} S_i S_j$, where $S_i = \pm 1$, $i = 1, 2, ...N$ sites, and J_{ij} is $|J| = 1/k_B T$ is the ferromagnetic coupling over neighbouring sites i and j on the network.

The Ising model and other Monte Carlo algorithms can be used themselves as demanding tests of the quality of random numbers, based on comparisons with known results [7].

4 Implementation & Timing

Common methodologies utilise computer CPUs to produce pseudo-random numbers using bitwise operations and mathematical operations to suitably randomise a number. The Mersenne-Twistor [49] is a common generator algorithm to produce high quality numbers, whereas the linear congruential algorithm, which is used in Unix rand, is a common and well known low quality example. Producing truly random numbers is impossible when using a algorithm running on a computer, this is the realm of the hardware random number generators (RNGs).

The algorithmic tradeoff space covers very high-quality generator algorithms such as the Mersenne-Twistor that are significantly slower than those very-fast but lower-quality algorithms such as linear congruential generators. In between these extreme cases it is often possible to improve low-quality generator algorithms by adding lag tables and shuffle tables to further randomise or decorrelate the sequences of random deviates and indeed to combine several independent algorithmic sources together.

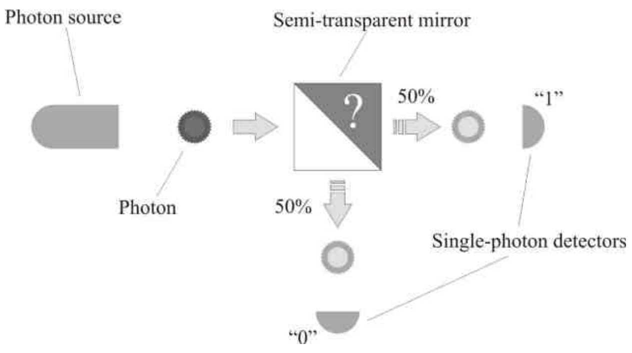

Figure 3: Description of the method for producing a random bit in the Quantis device.

4.1 Quantis Random Number Generator

The quantum random number generator we assess in this paper is the Quantis PCI quantum random number generator produced by ID QUANTIQUE SA [32]. This generator uses a photon emitter directed at a semi transparent mirror, which lets the photons through with a theoretical probability of 50% as shown in Figure 3. Each generator allows for a constant stream of random bits of up to 4 MBits/s. The PCI device contains 4 separate generators, bringing the theoretical maximum random stream to 16MBits/s or ≈ 500 deviates per millisecond.

The Quantis card supports both Windows and various flavours of Linux. For our testing we used Ubuntu Linux with the standard Quantis driver installation. The drivers API facility provides various methods for retrieving different data types. The most low level of these is the `QuantisRead` method:

```
int QuantisRead(QuantisDeviceType deviceType,
                unsigned int deviceNumber,
                void* buffer, size_t size);
```

This generates `size` bytes of random numbers into the variable `buffer`, where `size` is constrained to `QUANTIS_MAX_READ_SIZE`. To get more than this we must loop until the desired size has been reached. Alternately we can use:

```
int QuantisReadInt(QuantisDeviceType deviceType,
                   unsigned int deviceNumber,
                   int* value);
```

To get a signed integer `value` from the device. This method is much slower at reading multiple numbers than reading raw bytes as we show in section 5. To overcome this problem, we use `QuantisRead` in a multi-threaded environment where one thread is caching blocks of random bytes while the consumer thread uses them. This method may still not be sufficient for algorithms such as the Monte Carlo, but will significantly reduce the time over using `QuantisReadInt`.

5 Performance & Quality

For most scientific purposes it is sufficient to say that they need to be sufficiently uncorrelated that when used for a Monte Carlo simulation or other application the deviate quality does not lead to an observable bias [50]. Or put more simply – that the random number generator does not lead the applications programmer to the wrong answer. Various statistical tests [8], both at a straightforward level [51], checking for visual planar correlations [52] planes and other approaches such as the spacing test, scatter-plots, that detect obvious patterns or simple statistics are possible, as well as very specific application related tests that are highly sensitive to correlations.

To evaluate the raw performance of generators we test four different popular pseudo-random number generators: Mersenne Twister (MT), Ran described in the book Numerical Recipes (Ran) [53], the standard Unix rand and Marsaglia's lagged-Fibonacci Generator (LFG). These generators were tested for randomness using the birthday spacings test found in the diehard testing suite for random numbers, with the values $N = 2^{32}, M = 2^{12}$ and $\lambda = 4$. This configuration is advised in [54]. Supplementary tests were also performed with the standard diehard test suite [55] and these confirm the below findings.

Algorithm	Birthday Spacings
	Pass/Fail
Ran	✓
LFG	✓
MT	✓
Quantis (to CPU)	✓
Unix Rand	X

Table 1: Results of Birthday Spacings test of different RNG algorithms. Tick and Cross indicate pass and fail respectively

Table 1 shows that all except the Unix rand random numbers pass the birthday spacings test. This is in line with common knowledge about the periods of these generators [1].

Applications of specific random number generators are dependent on the speed in which the numbers can be attained by the client, where client refers to a central processing unit, graphics processing unit, etc. In random number intensive applications, such as the Monte Carlo algorithms in Ising/ Potts models, computation time is negligible compared to the fetch time for random numbers. Whereas, in cryptography the computation time significantly outweighs the fetch time for the random numbers, which allows slow generators to hide their speed by caching numbers for fast use by other threads.

To test the speed of the algorithms we generate ten million uniform floating point numbers and find the number of deviates per millisecond on an Intel Core 2 Duo at 2.1 GHz using the four algorithms that passed the birthday spacing test. The CPU algorithms only utilise one of the cores available on the CPU. We have also implemented a CUDA GPU version of the lagged-Fibonacci generator [30] and report the performance measured on an NVidia GeForce GTX 580.

Algorithm	Performance
	Deviates Per Millisecond
Ran	24085
LFG	13367
MT	22795
Quantis (Single Thread)	61
Quantis (Multi Thread)	111
CUDA(LFG)	$1.28e10^7$

Table 2: Performance of different RNG algorithms.

Table 2 shows that the results for all of the CPU pseudo-random number generators are comparable in speed, with the Ran algorithm producing the most at 24085 deviates per millisecond. This is more than two orders of magnitude faster than the single threaded Quantis generator at about 61 32-bit deviates per millisecond. The lagged-Fibonacci generator on the CUDA GPU is another 2-3 orders of magnitude faster than the CPU algorithms.

6 Discussion & Conclusions

Section 5 shows that all but the Unix rand pseudo-random algorithms pass the Extended Birthday spacing and Die Hard tests that we have implemented. These are well known algorithms and the results are common knowledge [3], hence it is unsurprising that the widely used Unix rand failed. This is further proof that this function should not be used. It has been suggested [3] that lagged-Fibonacci generators may erroneously fail the Birthday spacings test, but this does not appear to be the case for our implementation, which passes the test.

Although these pseudo-random number generators pass most common and also more stringent tests implemented in this paper, this does not guarantee their true randomness in the face of tests yet to be adviced. Using physical phenomena, such as photon emitters like the one used in this paper or Intel's on chip temperature variation random source, allows us to guarantee that the number is completely random and free from any bias. Although, the question remains how to test these hardware random sources and can we engineer a test that identifies only a truly random number?

Performance of the generators was as expected [30], with the CUDA GPU LFG algorithm producing $1.28e10^7$ random deviates per millisecond. The single threaded Quantis card algorithm produces only 61 32-bit deviates per millisecond and 111 deviates for the multi-threaded implementation. This is much slower than the theoretical maximum of 500 32-bit deviates from the 16MBits/s stream of random bits [32]. We attribute this latency to the fetch time from the card over the PCI bus and the conversion time to the specified data type. The speed-up attained by introducing multiple threads is significant as this allows us to hide the time lost in the conversion process and by fetching the maximum number of bytes at each API call we minimise any latency that is associated in calling the Quantis card via the PCI bus. For Monte Carlo algorithms even the CPU pseudo random algorithms are the bottleneck in the simulation, hence the Quantis card is much too slow for these. A good compromise is to use the numbers produced by the Quantis card to seed a good pseudo-random number generator, thus ensuring that the seeds are statistically independent.

If Intel succeeds in creating a truly random number generator producing 2.4 billion random bits per second [31], then this will significantly increase the reasons for using a hardware random source for random heavy algorithms. Until that point, long period pseudo-random number generators will continue to be the best choice for Monte Carlo algorithms. However, for low random frequency algorithms that depend on high quality random numbers, such as generation of cryptographic keys, current hardware generators are an excellent choice.

We have found that when used in the correct situation the Quantis card is an invaluable resource to computer simulations. However, random number generation is very much an application specific field and we have shown that, when compared to conventional pseudo-random generators, the time it takes to produce a single random deviate with the Quantis card is several orders of magnitude slower. Furthermore, the generation with the Quantis card is inherently serial and does not benefit from parallelisation on either the CPU or GPU. However, we have discussed how this latency may be hidden when the program does not require random numbers often by using a separate thread that fetches the numbers from the Quantis device and prepares them for the main process to use as needed. Another method we have discussed is using the Quantis device to produce truly random seeds for a high-quality pseudo-random number generator.

Graphics processing units offer a performance increase of about 2-3 orders of magnitude over the tested sequential CPU implementations. They have been shown [56] to be a powerful accelerator for Monte Carlo simulations that heavily depend on random numbers. However, developing high-performance code for GPUs is significantly more complex and time consuming than it is to write a sequential or even multi-threaded CPU implementation.

In summary, the field of computer generated random number algorithms is one of "horses for courses" - there is no single best algorithm that will satisfy all requirements. Before starting any project using Monte Carlo algorithms and for which the quality of the random numbers matters, it is therefore of great worth to carefully consider which algorithm to use.

References

[1] L'Ecuyer, P.: Software for uniform random number generation: distinguishing the good and the bad. In: Proc. 2001 Winter Simulation Conference. Volume 2. (2001) 95–105

[2] Brent, R.P.: A fast vectorized implementation of wallace's normal random number generator. Technical report, Australian National University (1997)

[3] Marsaglia, G.: Random Number generation. Florida Preprint, 1983

[4] Marsaglia, G.: A Current view of random number generators. In: Computer science and statistics: 16th symposium on the interface, Atlanta (Mar 1984) Keynote address.

[5] Marsaglia, G., Tsay, L.H.: Matrices and the Structure of Random Number Sequences. Linear algebra and its applications **67** (1985) 147–156

[6] Marsaglia, G., Zaman, A., Tsang, W.W.: Toward a universal random number generator. Statistics and Probability Letters **9**(1) (January 1987) 35–39 Florida State preprint.

[7] Coddington, P.D.: Analysis of random number generators using monte carlo simulation. J. Mod. Physics C **5**(3) (1994) 547–560

[8] Cuccaro, S., Mascagni, M., Pryor, D.: Techniques for testing the quality of parallel pseudo-random number generators. In: Proc. of the 7th SIAM Conf. on Parallel Processing for Scientific Computing,, Philadelphia, USA, SIAM (1995) 279–284

[9] Binder, K., ed.: Monte Carlo Methods in Statistical Physics. 2 edn. Topics in Current Physics. Springer-Verlag (1986) Number 7.

[10] Binder, K., ed.: Applications of the Monte Carlo Method in Statistical Physics. Topics in Current Physics. Springer-Verlag (1987)

[11] Johnsonbaugh, R.: Discrete Mathematics. 5th edn. Number ISBN 0-13-089008-1. Prentice Hall (2001)

[12] Srinivasan, S., Mathew, S., Ramanarayanan, R., Sheikh, F., Anders, M., Kaul, H., Erraguntla, V., Krishnamurthy, R., Taylor, G.: 2.4ghz 7mw all-digital pvt-variation tolerant true random number generator in 45nm cmos. In: VLSI Circuits (VLSIC), 2010 IEEE Symposium on. (june 2010) 203 –204

[13] Deng, L.Y., Xu, H.: A system of high-dimensional, efficient, long-cycle and portable uniform random number generators. ACM TOMACS **14**(4) (2003) 299–309

[14] Schneier, B.: Applied Cryptography. Second edn. Wiley (1996) ISBN 0-471-11709-9.

[15] Rukhin, A., Soto, J., Nechvatal, J., Smid, M., Barker, E., Leigh, S., Levenson, M., Vangel, M., Banks, D., Heckert, A., Dray, J., Vo, S.: A statistical test suite for random and pseudorandom number generators for cryptographic applications. Technical Report NIST Special Publication 800-22,, U.S. National Institute of Standards and Technology, (April 2010)

[16] Hormann, W.: The transformed rejection method for generating poisson random variables. Insurance: Mathematics and Economics **12**(1) (February 1993) 39–45

[17] Press, W., Flannery, B., Teukolsky, S., Vetterling, W.: 7. In: Numerical Recipes in C. Cambridge University Press (1988) 204–241 Random Numbers.

[18] Menezes, A., van Oorschot, P., Vanstone, S.: 5 - Pseudorandom bits and sequences. In: Handbook of Applied Cryptography. CRC Press, Boca Raton, FL (1996) 169–190

[19] Barker, E., Kelsey, J.: Recommendation for random number generation using deterministic random bit generators (revised). Technical Report NIST Special Publication 800-90, U.S. National Institute of Standards and Technology (March 2007)

[20] Coddington, P., Mathew, J., Hawick, K.: Interfaces and implementations of random number generators for java grande applications. In: Proc. High Performance Computing and Networks (HPCN), Europe 99, Amsterdam. (April 1999)

[21] Blum, M., Micali, S.: How to generate cryptographically strong sequences of pseudo-random bits. SIAM J. Comput. **13** (November 1984) 850–864

[22] Coddington, P., Newall, A.: Japara - a java parallel random number generator library for high-performance computing. Technical Report DHPC-144, The University of Adelaide (2004)

[23] Newell, A.: Parallel random number generators in java. Master's thesis, Computer Science, Adelaide University (2003)

[24] Smith, K.A., Reddaway, S.F., Scott, D.M.: Very high performance pseudo-random number generation on DAP. Comp. Phys. Comms. **37** (1985) 239–244

[25] Brent, R.P.: Uniform random number generators for supercomputers. In: Proc. 5th Australian Supercomputer Conference, Melbourne, Australia (1992)

[26] Paul, W., Heermann, D.W., Desai, R.C.: Implementation of a random number generator in OCCAM. Mainz preprint 87/47, 749 (Dec 1989)

[27] Bader, D.A., Chandramowlishwaran, A., Agarwal, V.: On the design of fast pseudo-random number generators for the cell broadband engine and an application to risk analysis. In: Proc. 37th IEEE Int. Conf on Parallel Processing. (2008) 520–527

[28] Giles, M.: Notes on CUDA implementation of random number genertors. Oxford University (January 2009)

[29] Langdon, W.: A Fast High Quality Pseudo Random Number Generator for nVidia CUDA. In: Proc. ACM GECCO'09. (2009)

[30] Hawick, K., Leist, A., Playne, D., Johnson, M.: Speed and Portability issues for Random Number Generation on Graphical Processing Units with CUDA and other Processing Accelerators. In: Proc. Australasian Computer Science Conference (ACSC 2011). (2011)

[31] Anthes, G.: The quest for randomness. Comm. ACM **54**(4) (April 2011) 13–15

[32] ID Quantique White Paper: Random Number Generation Using Quantum Physics. Technical Report Version 3.0, ID Quantique SA, Switzerland (April 2010) QUANTIS.

[33] Landau, D.P., Binder, K.: A Guide to Monte-Carlo Simulations in Statistical Physics. Volume 3rd edition. Cambridge University Press (2009)

[34] Niss, M.: History of the Lenz-Ising Model 1920-1950: From Ferromagnetic to Cooperative Phenomena. Arch. Hist. Exact Sci. **59** (2005) 267–318

[35] Ising, E.: Beitrag zur Theorie des Ferromagnetismus. Zeitschrift fuer Physik **31** (1925) 253258

[36] Onsager, L.: Crystal Statistics I. Two-Dimensional Model with an Order-Disorder Transition. Phys.Rev. **65**(3) (Feb 1944) 117–149

[37] Potts, R.B.: Some generalised order-disorder transformations. Proc. Roy. Soc (1951) 106–109 received July.

[38] Metropolis, N., Rosenbluth, A.W., Rosenbluth, M.N., Teller, A.H., Teller, E.: Equation of state calculations by fast computing machines. J. Chem. Phys. **21**(6) (Jun 1953) 1087–1092

[39] Swendsen, R.H., Wang, J.S.: Nonuniversal critical dynamics in Monte-Carlo simulations. Phys. Rev. Lett. **58**(2) (Jan 1987) 86–88

[40] Wolff, U.: Comparison Between Cluster Monte Carlo Algorithms in the Ising Model. Physics Letters B **228**(3) (September 1989) 379–382

[41] Marinari, E., Parisi, G.: Simulated Tempering - a New Monte-Carlo Scheme. Europhysics Letters **19**(6) (July 1992) 451–458

[42] Hastings, W.: Monte-Carlo Sampling Methods Using Markov Chains And Their Applications. Biometrika **57**(1) (1970) 97–107

[43] Jorgensen, W.: Perspective on "Equation of state calculations by fast computing machines". Theoretical Chemistry Accounts **103**(3-4) (February 2000) 225–227

[44] Chib, S., Greenberg, E.: Understanding the Metropolis-Hastings Algorithm. American Statistician **49**(4) (November 1995) 327–335

[45] Linke, A., Heermann, D.W., Altevogt, P., Siegert, M.: Large-scale simulation of the two-dimensional kinetic Ising model. Physica A: Statistical Mechanics and its Applications **222**(1-4) (December 1995) 205–209

[46] Okano, K., Schülke, L., Yamagishi, K., Zheng, B.: Universality and scaling in short-time critical dynamics. Nuclear Physics B **485**(3) (February 1997) 727–746

[47] Fulco, U., Lucena, L., Viswanathan, G.: Efficient search of critical points in Ising-like systems. Physica A: Statistical Mechanics and its Applications **264**(1-2) (February 1999) 171–179

[48] Lima, F., Stauffer, D.: Ising model simulation in directed lattices and networks. Physica A: Statistical Mechanics and its Applications **359**(1) (January 2006) 423–429

[49] Matsumoto, M., Nishimura, T.: Mersenne twistor: A 623-diminsionally equidistributed uniform pseudorandom number generator. ACM Transactions on Modeling and Computer Simulation **8 No 1.** (1998) 3–30

[50] Knuth, D.: The Art of Computer Programming: Seminumerical Algorithms. 3rd edn. Volume 2. Addison-Wesley (1997)

[51] Coddington, P.D., Ko, S.H.: Techniques for empirical testing of parallel random number generators. In: Proc. International Conference on Supercomputing (ICS'98. (1998) DHPC-025.

[52] Marsaglia, G.: Random numbers fall mainly in the planes. Proc. Natl. Acad. Sci. **61**(1) (1968) 25–28

[53] Press, W.H., Teukolsky, S.A., Vetterling, W.T., Flannery, B.P.: Numerical Recipes - The Art of Scientific Computing. Third edn. Cambridge (2007) ISBN 978-0-521-88407-5.

[54] Marsaglia, G., Tsang, W.W.: Some Difficult-to-pass Tests of Randomness. Journal of Statistical Software **7**(3) (January 2002) 1–9

[55] Marsaglia, G.: Diehard Battery Of Tests Of Randomness. `http://www.stat.fsu.edu/pub/diehard/` (1995)

[56] Hawick, K., Leist, A., Playne, D.: Regular Lattice and Small-World Spin Model Simulations using CUDA and GPUs. Int. J. Parallel Prog. **39**(CSTN-093) (2011) 183–201