ShoveRand: a Model-Driven Framework to Easily Generate Random Numbers on GP-GPU

Jonathan Passerat-Palmbach, Claude Mazel, Bruno Bachelet, David R.C. Hill

UMR 6158 CNRS

ShoveRand: a Model-Driven Framework to Easily Generate Random Numbers on GP-GPU

Jonathan Passerat-Palmbach*† ‡ §,
Claude Mazel † ‡ §,
Bruno Bachelet † ‡ §,
David R.C. Hill † ‡ §

Originally published in: IEEE International Conference on High Performance Computing & Simulation — July 2011 — pp 41-48
http://dx.doi.org/10.1109/HPCSim.2011.5999805

Abstract: *Stochastic simulations are often sensitive to the randomness source that characterizes the statistical quality of their results. Consequently, we need highly reliable Random Number Generators (RNGs) to feed such applications. Recent developments try to shrink the computation time by using more and more General Purpose Graphics Processing Units (GP-GPUs) to speed-up stochastic simulations. Such devices bring new parallelization possibilities, but they also introduce new programming difficulties. Since RNGs are at the base of any stochastic simulation, they also need to be ported to GP-GPU. There is still a lack of well-designed implementations of quality-proven RNGs on GP-GPU platforms. In this paper, we introduce ShoveRand, a framework defining common rules to generate random numbers uniformly on GP-GPU. Our framework is designed to cope with any GPU-enabled development platform and to expose a straightforward interface to users. We also provide an existing RNG implementation with this framework to demonstrate its efficiency in both development and ease of use.*

Keywords: General-Purpose computation on Graphics Processing Units (GPGPU); Libraries and Programming Environments; Parallelization of Simulation; Object Oriented Programming & Design

* This author's PhD is partially funded by the Auvergne Regional Council and the FEDER
† ISIMA, Institut Supérieur d'Informatique, de Modélisation et de ses Appplications, BP 10125, F-63173 AUBIERE
‡ Clermont Université, Université Blaise Pascal, LIMOS, BP 10448, F-63000 CLERMONT-FERRAND
§ CNRS, UMR 6158, LIMOS, F-63173 AUBIERE

RESEARCH CENTRE
LIMOS - UMR CNRS 6158
Campus des Cézeaux
Bâtiment ISIMA
BP 10125 - 63173 Aubière Cedex
France

1 INTRODUCTION

Random number generation is a sensitive component for stochastic simulation models. For computationally eager applications, High Performance Computing (HPC) devices such as General-Purpose Graphics Processing Units (GP-GPU) are now widely used to reduce computation time. However, we still need better software tools to develop applications on such architectures. CPU-enabled simulations can take advantage of a wide range of statistically sound Pseudo-Random Number Generators (PRNGs) and libraries, but we still need quality random number generators for GP-GPU architectures, and more precisely, a particular care has to be given to their parallelization.

Many tries have been done to propose GPU-enabled implementations of PRNGs. Previous studies like [16, 15] bring up random number generation on GPU. Many strategies were adopted, but few of them distinguish particularly from the others. It is also very difficult to select the right PRNG to use since you need to consider both the statistical quality of the PRNG and the GPU implementation quality if you do not want to see the overall performance of your application seriously drop.

Recent development frameworks like CUDA and OpenCL enabled much more ambitious developments on GPU. We can now make use of some software engineering paradigms, such as Object-Oriented programming or generic programming, and this introduces a new scope in GPU-enabled applications development. As GPU-enabled software become more complex, they also need enhanced tools to rely on. Thus, PRNGs implementations on GP-GPU cannot be achieved in non-structured software development environments anymore. We need a framework to smoothly integrate PRNGs in existing developments.

In this article, we propose a solution merging two main ideas. First, we want to enforce the use of good quality PRNGs on GPU. To do so, this work intends to aggregate well-designed random number generation solutions in order to distribute them in a packaged form. Second, we propose reflections on the form this package will take. To secure this aspect, we rely on sound software engineering principles to provide a modern framework. The latter must display a common interface to its user, regardless of the RNG he has chosen. Another important aspect, in our mind, is that this framework must evolve easily thanks to external developments. It is designed in such a way, to quickly integrate any already developed GPU-enabled RNG.

In this study, we will:

- Propose a meta-model for RNG libraries on GPU;
- Define implementations constraints to build such a dedicated framework;
- Implement a generic declination of a quality-proven GPU-enabled PRNG.

2 GPU-ENABLED RNG LIBRARIES

This section will present two libraries dedicated to random numbers generation on GP-GPU architectures: NVIDIA's CURAND [13] and Thrust::random [5]. They are, to our knowledge, the most relevant libraries to obtain random sequences on GP-GPU, at the time of writing. Next, we will describe their intrinsic properties, considering both the RNGs they embed and the quality of their implementations.

2.1 NVIDIA's CURAND

2.1.1 Presentation

Introduced in the 3.2 version of the CUDA framework, CURAND has been designed to generate random numbers in a straightforward way on CUDA-enabled GP-GPUs. The main advantage of CURAND is that it does not target any platform or application kind. It is indeed able to produce both quasi-random and pseudo-random sequences either on GP-GPU or on CPU. Except from the initialization part, the API (Application Programming Interface) of the library stays globally the same, no matter which kind of random sequence you select or the platform you run your application on.

2.1.2 Characteristics

The current version of CURAND, shipped with CUDA SDK 3.2, comes with two kinds of random numbers (pseudo-random and quasi-random). This leads to two different implementations cohabitating in the library. On the one hand, we find a Sobol quasi-random numbers generator (QRNG), a very good choice that has also been done for the GNU Scientific Library (GSL). The "quasi" modifier denotes that we do not deal with random or pseudo-random numbers. Instead, we are in presence of low-discrepancy deterministic sequences, which are interesting to improve numerical integration or other applications with better space filling and density properties. Toughly speaking, these generators seek to uniformly fill an n-dimensional space with points, without creating clusters of points. They are mainly used with quasi-Monte-Carlo computations (in physics, finance, etc.). On the other hand, the pseudo-random side, a XORWOW [12] PRNG has been implemented. Direct applications of PRNGs are stochastic simulations, which require fast randomness sources but also need to be reproducible as any scientific experiment and also for debugging purposes.

Since we focus on simulations fed by pseudo-random sequences, we will only discuss the XORWOW algorithm. It is reckoned for its speed and small memory footprint. The latter property is a significant advantage when you need to implement this PRNG on a GP-GPU. The quicker the memory areas are, the fewer amount of data they are able to store. Thus, a PRNG saving memory is more likely supposed to be implemented using quick memories on GP-GPU. It is also fairly unpleasant to waste large amounts of bytes to store data for a PRNG when you still have to place your algorithm's data in the quickest memory areas you have at your disposal.

However, the main drawback we found with this algorithm is that it has shown limited statistical qualities in the literature [14]. The CURAND documentation even specifies that problems could be encountered when using this PRNG[13] (pp. 38-39).

2.2 Thrust::random

2.2.1 Presentation

Thrust::random is part of a GP-GPU-enabled general purpose library called Thrust. This open-source project intends to provide a GP-GPU-enabled library equivalent to some classic general-purpose C++ libraries, such as STL or Boost. Classes are split through several namespaces, which Thrust::random is an example. The latter contains all classes and methods related to random numbers generation on GP-GPU. Thrust::random implements three PRNGs, each through a different C++ template class. We find a Linear Congruential Generator (LCG), a Linear Feedback Shift (LFS) and a Subtract With Carry (SWC), which we will all describe later and that are, in our opinion, not adapted to High Performance Computing. These classes are the base

of this module that reuses them to define combined PRNGs or API adaptors of these original classes.

2.2.2 Characteristics

We need to distinguish two aspects when attempting to characterize this library. First, the template form of Thrust::random enables it to execute quickly on the host. Generic programming is used here to bypass the fact that we do not have true object-oriented features, such as inheritance and polymorphism, which would however imply an overhead at runtime. Polymorphism is in fact not available on every GP-GPU architecture yet, as we will see in Section 4.1. Even if the three PRNGs implementations are not directly related through a class hierarchy, they display an unchanged API. This way, PRNG intrinsic particularities are hidden to users who can simply call identical methods through every object issued by these classes. Furthermore, the library makes a clever use of the Adapter design pattern [2] to furnish combined PRNGs or to return extracts of an original random sequence. Nonetheless, Thrust::random does not provide a unified framework to add new PRNGs so that a potential newcomer would have to respect implicitly library conventions in order to integrate his own PRNG implementation in it.

Second, the substance of Thrust::random, formed by the underlying PRNGs, may also be discussed. We previously mentioned three PRNGs implementations. Let us now focus on their statistical qualities according to the literature. The first to be considered is a LCG, but we will only present it briefly, because it has several times been spotted as a flawed PRNG for both sequential [7, 9] and parallel [4, 8] applications. Therefore, we discourage users to choose it, even through Thrust's implementation. The LFS PRNG also implemented in Thrust::random is derived from the Boost.Random library. The latter implements in fact a Tausworthe PRNG introduced in [19]. Tausworthe generator alone is known to have bad empirical performance, as stated in [3]. That is why it is commonly coupled with other generators to form combined generators. Hopefully, generators can be easily combined thanks to Thrust::random design. The third PRNG proposed is a Subtract With Carry PRNG announced to be based upon [10, 11]. Please note that Marsaglia rather proposed a Subtract With Borrow (SWB)... Nevertheless, this generator does not seem flawless. It misses some classics statistical tests such as BirthdaySpacing according to [7]. A more recent study from the same authors [9] even likens SWB to LCG, which are structurally weak for modern applications as we previously stated.

3 A MODEL-DRIVEN LIBRARY TO OVERCOME KNOWN ISSUES

In the light of the previous descriptions, we spotted lacks in different domains involved in RNG libraries design: on the one hand, the embedded RNGs choice and on the other hand, the API design. These shortages showed the importance of defining a novel framework for RNG implementations on GP-GPU. To enable developers to use RNGs easily, we still trust the library approach is the best option. In this section we will introduce the choices we made to prevent our model-driven framework to display such misses. Therefore, we consider two axes in this presentation: the PRNG that will be embedded in ShoveRand and the meta-model which our library relies on.

3.1 Default shipped PRNG

Libraries are a reliable way to spread software elements. Considering our previous assertions on PRNGs embedded in other libraries, we took care to issue quality proven generators with our proposal. In this work, we have selected Mersenne Twister for Graphics Processors (MTGP) [17], benchmarked in [16] and recently improved by Matsumoto and Saito. Briefly, this PRNG inherits from former Mersenne Twister generators. It is based upon the same kind of algorithm that allows its elders to display huge periods. MTGP can bear periods up to 2^{110503}. This feature is not mandatory for modern applications as said in [4, 8] and can be a drawback when we consider the memory footprint of the generator. As a matter of fact, the higher the period is, the more memory is consumed.

Moreover, as for the initial Mersenne Twister generator, MTGP is shipped with an algorithm called Dynamic Creator (DC) that issues independent parameter sets, called parameterized statuses hereafter. DC enables parallelization through the parameterization technique, which is supposed to furnish independent random sequences thanks to an upstream step providing a distinct parameterized status to each parallel element. A unique identifier is directly integrated as a part of the characteristic polynomial of the matrix that defines the recurrence, and belongs to the parameterized status of MTGP. Two identifiers will consequently lead to two different parameterized statuses. Furthermore, DC ensures that the characteristic polynomials we get are mutually prime. Its authors assert that the random sequences generated with such distinct parameterized statuses will be highly independent. Even if this fact cannot be mathematically proven, it is widely admitted in the scientific community.

The couple formed by MTGP and DC fulfills the requirements of a parallel PRNG, thanks to its ability to produce independent random sequences. Details of the integration process of this generator in ShoveRand are provided in Section 4.3.

3.2 Inputs of the meta-model

To improve the software design, we propose hereafter a meta-model to represent RNGs running on any platform. As we have seen with the previous examples, there is currently no RNG framework on GP-GPU. This leads to heterogeneous GP-GPU-enabled RNG propositions. The latter can even sometimes be of very good quality but a bit tough to use for non-RNG experts. Thus, our goal is to design this generic RNG framework and furnish an implementation embedding quality-proven GP-GPU-enabled RNGs. If we tried to avoid the previously noticed flaws in the literature propositions, we also acknowledge good design elements that we reused without any hesitation in our model.

Once again, ShoveRand is designed as a base framework for other developers to integrate their GP-GPU-enabled RNGs implementations. We intend to furnish an empty shell that will require a small amount of work to embed a new RNG. By doing this, we detach our production from potential deprecation of the RNGs it proposes. Newer implementations will always be available at the smallest effort of integration, in the direct line of our primarily announced goal: simplicity of use.

The previous conclusions led us to work on a meta-model matching the evocated needs. Hopefully, most of the RNGs share common characteristics that we were able to factorize in few classes. Before describing the machinery of our library model, let us introduce its schematized version through the UML class diagram in Figure 1:

We have chosen to handle each part of a classical RNG through different classes. This distinction enables our model to represent various kinds of RNGs, from the most naive to the most complicated one, regardless of the type of the generated random numbers. We obtain a resulting hierarchy constituted by four base classes for the library: *RNG*, *Algorithm*, *ParameterizedStatus*

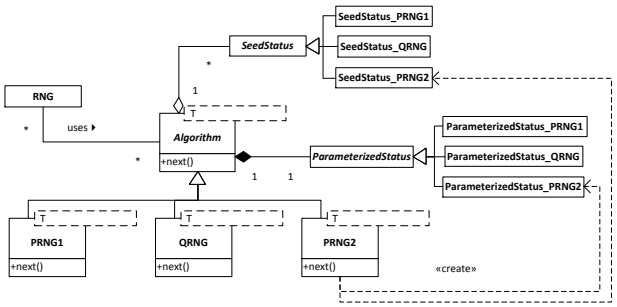

Figure 1: **Polymorphic Version of the Model**

and *SeedStatus*. Relations are quite simple between these elements. *ParameterizedStatus* and *SeedStatus* are aggregated by the different *Algorithm* declinations. *RNG*, as far as it is concerned, handles the previous classes and exposes the public interface to the users. It is the key of the convenient uniform API of the library.

For understanding purpose, parameters representation needs to be detailed. As it is described on the class diagram, we split them in two classes. First, *ParameterizedStatus* contains the effective input parameters that determine the whole random sequence. It is highly tied to the *Algorithm* using it, so that a couple of instances from these two classes define a random sequence. Second, *SeedStatus* mirrors the internal state vector that stores the current state of the generator. Please note that the concrete implementations of these two *Status* classes are bound to the *Algorithm* employing them. You can notice that this relation is explicitly drawn for *PRNG2* only on the class diagram in Figure 1, in order not to overload the diagram. Still, every *RNG* is intended to do so. It is also compulsory for each Algorithm implementation to furnish a representation of each of the two introduced classes, even if one is empty. For instance, some PRNGs do not employ parameters, so such generators will furnish an empty implementation of the *ParameterizedStatus* class.

Finally, every member of this meta-model is generic. This feature allows us to abstract the algorithms from the data type (noted T on the figures) of the generated random numbers, which are commonly 32-bit or 64-bit integers, or floating-point numbers of single or double precision. The library is also abstracted from the underlying programming framework used to develop applications that will run on GP-GPU. Indeed, the core of the library is not bound to a particular GP-GPU device code technology, allowing developers to write RNGs implementations in their favorite language (CUDA or OpenCL), as long as it uses a C++ compliant compiler. For instance, the PRNG shipped with our library is a CUDA implementation perfectly compiling with NVIDIA's nvcc CUDA compiler.

4 META-MODEL

IMPLEMENTATION

We intend to offer a straightforward API, enabling one to call a RNG without wondering indefinitely which parameters he should set to use it correctly. As it has been done in the libraries we studied, setting RNGs parameters to default values prevents users to set parameters to wrong values, without understanding their meaning.

Other libraries already proposed a common API regardless of the employed RNG. This is in our mind the key feature of these libraries. Indeed, it can be interesting to rapidly switch between two or more RNGs in order to compare the output results of the same application depending on its randomness source. With a uniform API, enhanced by generic programming, the only thing you have to do is to change a single parameter. Consequently, we obviously reused this design element in our meta-model. This section will show how ShoveRand takes advantage of the introduction of generic programming to propose even more advanced features.

4.1 Policy-based class design

Several pseudo-random number generation algorithms can be implemented in ShoveRand through different classes. Indeed, the RNG class gets its behavior from these classes. This situation largely recalls the Strategy design pattern [2], which intends to dynamically modify the behavior of the class instances according to the Strategy they are combined with. This feature simply relies on virtual functions calls. Unfortunately, even cutting-edge GP-GPUs prevent us from using strong polymorphism, i.e. virtual functions. As a matter of fact, GP-GPU architectures do not support the implementation of virtual functions tables, at the time of writing. Yet, this latter capacity is mandatory to enable polymorphism. Hopefully, [1] introduced a compile time equivalent to the Strategy design pattern, which are called policies.

A policy defines a software component designed to be a unit of behavior. These components are basic classes that can potentially be combined to form complex classes. Policies intend to be an efficient design element, in this way they are passed as template parameters to the classes using them: the host classes. Now, remember that the dynamic Strategy solution presented the aspect to display a uniform API, thanks to inheritance and virtual functions. Policies provide an equivalent to this notion, given that they are a set of rules defining how a class should look like in order to furnish the features they propose. These rules constrain the interface of a policy's implementations classes, called policy classes, to guide their behavior towards the policy's expectations. Finally, all the policy classes implementing the same policy must display an identical interface, so that they can be swapped without any problem in the class using them.

In our case, the policy idiom is applied to the random number generation algorithm. Concretely, Algorithm appears as a template parameter of the RNG class. Each policy class implements a different kind of generator that can be interfaced with RNG and used identically. Thus, policies helped us to overcome hardware limitations and to set up an equivalent to the polymorphic Strategy design pattern. The Algorithm policy can be formulated as follows: *Algorithm prescribes a class template of one type T representing the type of the generated values. Algorithm exposes at least two member-functions init and next that respectively initializes the generator and generates the next random value. The implementation must own representations of both a SeedStatus and a ParameterizedStatus, in order to handle its internal parameters.*

4.2 Strong static typing

Template meta-programming presents other advantages in our case than genericity facilities. To structure our GP-GPU-enabled RNG framework, it is important to define some rules that will guide the future RNG implementations. The previous model fulfills this need by two aspects.

First, when RNG is defined for a particular (T; Algorithm) couple, it defines a new type allowing us to guess statically, understand at compilation time, whether two RNG instances possess the same template parameters. This feature can be very useful when using RNGs in parallel since we need to ensure the independence between random streams produced by the RNGs. For statistical analysis, it would be a nonsense to mix random sources during successive executions

of a simulation. Such a misconception could wreak havoc on a stochastic simulation, for example by introducing undesired correlations in the random streams allocated to its replications. When CURAND exposes functions that take any of their implemented generators' status as parameter, our strong typing feature, coupled with an object-oriented design, prevents users to feed their applications with different RNGs for the same execution.

Second, we set up the concept checking mechanism at the heart of the library, in order to prevent users to provide any class as an Algorithm to RNG. Concept checking is a generic programming feature available through different implementations with C++. Introduced by [18] and implemented in the Boost Concept Check Library (BCCL), concept checking enables us to statically check the interface of a given class. This mechanism enables us to ensure of the correctness of the interfaces of the policy classes. In the previous part, we have enounced a constraints list to define the Algorithm policy. We have translated this list into concept checking rules thanks to BCCL. For instance, the Algorithm policy can be basically checked with the few lines exposed in Figure 2.

```
// concept requirements
BOOST_CONCEPT_USAGE(RNGAlgorithm) {
  // require Algo<T>::init()
  al_.init();
  // require T Algo<T>::next()
  value_ = al_.next();
  // require Algo<T>::ss_ to be of
  // SeedStatus<Algo> type
  same_type(ss_, al_.ss_);
  // require Algo<T>::ps_ to be of
  // ParameterizedStatus<Algo> type
  same_type(ps_, al_.ps_);
}
```

Figure 2: **Policy's Interface is Checked Through Boost Concept Check Library**

As a result, when users try to provide an Algorithm implementation to instantiate a particular RNG, the input Algorithm class is checked for the expected requirements list. Beyond the fact of proposing a generic uniform interface to RNGs, we ensure here that future implementations respect the standard we propose. You need to puzzle out that these constraints are here for information purpose. Users become aware of some lacks in their implementations with clear compiling errors, rather than letting misconception errors lurk in the shadows until someone calls a missing method. To conclude, this feature is a full part of the developer-friendly aspect of our library, and its straightforward uniform API is likewise.

Generic programming inputs from the two last studied points have been integrated in the meta-model. Figure 3 shows a class diagram based upon Figure 1, but where policies and concept checking have replaced the Strategy design pattern:

4.3 MTGP integration

MTGP is the PRNG we chose to integrate as an example for our framework. Its authors provide a CUDA implementation of their work. The latter intends to accelerate the generation of pseudo-random numbers on GP-GPU to get them back to the host side afterwards. However, we want

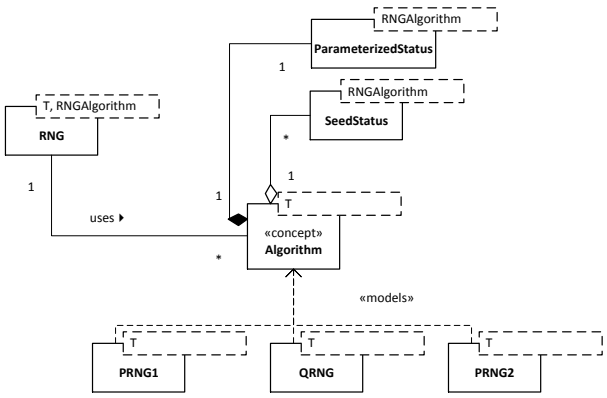

Figure 3: **Meta-Model Describing the ShoveRand Framework**

to generate numbers that can be directly consumed by the application currently running on the GP-GPU. Thus, we had to slightly modify Saito's original proposition to change its behavior regarding generated numbers. The original version is able to generate three numbers per GP-GPU thread, reducing this way memory transfer latency between host and GP-GPU by making each thread compute three times more. In our case, threads draw numbers in order to directly feed their next operation. We cannot make them draw several numbers and then use only one: this would introduce an overtime to finally get numbers wasted. Thus, we slimly changed the original code to make it fit our utilization constraints. Please note that these changes have been done upstream from the framework integration and are not tied to this operation.

Natively, MTGP issues three random number data types: integers, single precision floats and single precision floats contained in [0;1[. As we said previously, our framework implies to separate Parameters and Algorithm into different meta-classes. Algorithm depends on a data type, whereas Parameters depends on an Algorithm. This template machinery allows us to write MTGP code once by abstracting the output data type. We will then be able to use the MTGP algorithm through the RNG meta-class, without paying attention to the involved data type. Outside our framework, the same code would have been duplicated three times, once per supported data type. Parameters sometimes depend on the handled data type. Luckily, templates enable us to cover this kind of situation. Thanks to our meta-model that manages separately parameters and algorithms, and to the template specialization mechanism, we are able to treat particular parameters by specializing the template ParameterizedStatus class to provide an implementation dedicated to the Algorithm and the involved data type.

Considering the MTGP template class that implements the aforementioned RNG, the following code snippet gives an insight of ShoveRand's capacities and ease of use. A simple CUDA parallel program, a kernel, declares and uses MTGP to compute the product of two random numbers:

```
__global__ void testShoveRandMTGP(int* outputData) {

  shoverand::RNG< int, MTGP > rng;

  outputData[threadIdx.x] =
            rng.next() * rng.next();
}
```

Figure 4: **MTGP Integrated in ShoveRand**

5 CONCLUSION

We benchmarked two widespread libraries, furnished by NVIDIA research teams. The first one, CURAND, is dedicated to random number generation on GP-GPU, whereas the second, Thrust::random, is a part of the well known Thrust, a global purpose GP-GPU library.

However, according to several statistical studies, the encapsulated RNGs are mostly stated as bad quality randomness sources, in spite of their quickness and small memory footprint. Thus, we proposed a PRNG framework designed 'à la Boost', so as Thrust::random, but encapsulating quality-proven RNGs. In this way, we integrated Saito's work MTGP introduced in [17], which is a GP-GPU-enabled parallel PRNG. It was originally written to use the GP-GPU as an hardware accelerator to furnish large amounts of random numbers in a little time. The purpose here was to use it as a regular generator for stochastic computing on GP-GPU. Part of this work implied to modify the behaviour of this PRNG to have it generate a single random number per thread that could be directly consumed by the program currently running on the device. We intend to implement now other fine generators from L'Ecuyer's team, such as MRG32k3a [6].

Another major input of ShoveRand is its concept-checking mechanism. Thanks to this feature, we ensure that every RNG integrated in ShoveRand will display a common structure. This enables us to conceive stochastic applications independently of the underlying generator. Other future components will also take advantage of this statically checked skeleton. Indeed, we plan to implement the distributions features that come with the Boost.Random library. The interesting point here is that ShoveRand's will simplify the implementation of distributions because of its concept checking capabilities. In fact, constraints added to RNG structures will force every RNG implementation to very specific class members used by distributions.

Currently developed in C++, our proposition must be compiled for every new host it will run on. This solution supposes that the user owns the development toolkit to compile our sources to his own configuration. To bypass this constraint, we wish to implement our library in an abstract language such as Java or Scala, which are both executed in the Java Virtual Machine. Thanks to the OpenCL bindings proposed for the latter languages, our library could run in any environment as long as it supports Java and OpenCL.

Interested readers can obtain source code and documentation relative to ShoveRand on our university's software forge, at: `http://forge.clermont-universite.fr/projects/shoverand`.

ACKNOWLEDGMENTS

This work was financially supported by the regional council of Auvergne (France) in the InnovaPole/Auvergrid project. The authors would like to thank to Hélène Beaumont, Natacha

Vandereruch, Clément Beitone, Sébastien Cipière, Florian Guillochon and Nicolas Jean, for their careful reading and useful suggestions on the draft.

References

[1] A. Alexandrescu, *Modern C++ Design*. Addison-Wesley, 2001, iSBN: 0-201-70431-5.

[2] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design patterns: elements of reusable object-oriented software*. Addison-wesley Reading, MA, 1995.

[3] P. Hellekalek, "Good random number generators are (not so) easy to find," *Mathematics and Computers in Simulation*, vol. 46, no. 5-6, pp. 485–505, 1998.

[4] D. Hill, "Practical distribution of random streams for stochastic high performance computing," in *IEEE International Conference on High Performance Computing & Simulation (HPCS 2010)*, 2010, pp. 1–8, invited paper.

[5] J. Hoberock and N. Bell, "Thrust: A parallel template library," 2010, version 1.3.0. [Online]. Available: https://thrust.github.com/

[6] P. L'Ecuyer, R. Simard, E. Chen, and W. Kelton, "An object-oriented random-number package with many long streams and substreams," *Operations Research*, vol. 50, no. 6, pp. 1073–1075, 2002.

[7] P. L'Ecuyer, "Testing random number generators," in *Proceedings of the 1992 Winter Simulation Conference*, 1992, pp. 305–313.

[8] ——, *Encyclopedia of Quantitative Finance*, 2010, ch. Pseudorandom Number Gemerators.

[9] P. L'Ecuyer and R. Simard, "Testu01: A c library for empirical testing of random number generators," *ACM Transactions on Mathematical Software*, vol. 33, no. 4, pp. 22:1–40, August 2007.

[10] G. Marsaglia, B. Narasimhan, and A. Zaman, "A random number generator for pc's," *Computer Physics Communications*, vol. 60, no. 3, pp. 345–349, 1990.

[11] G. Marsaglia and A. Zaman, "A new class of random number generators," *Annals of Applied Probability*, vol. 3, no. 3, pp. 462–480, 1991.

[12] G. Marsaglia, "Xorshift rngs," *Journal of Statistical Software*, vol. 8, no. 14, pp. 1–6, 2003.

[13] NVIDIA, *CUDA CURAND Library*, NVDIA Corporation, August 2010.

[14] F. Panneton, "Construction d'ensembles de points basée sur des récurrences linéaires dans un corps fini de caractéristique 2 pour la simulation monte carlo et l'intégration quasi-monte carlo," Ph.D. dissertation, Département d'informatique et de recherche opérationnelle - Faculté des arts et des sciences - Université de Montréal, 2004.

[15] J. Passerat-Palmbach, C. Mazel, and D. Hill, "Pseudo-random number generation on gp-gpu," in *IEEE/ACM/SCS Workshop on Principles of Advanced and Distributed Simulation*, 2011, accepted, to be published.

[16] J. Passerat-Palmbach, C. Mazel, A. Mahul, and D. Hill, "Reliable initialization of gpu-enabled parallel stochastic simulations using mersenne twister for graphics processors," in *ESM 2010*, 2010, pp. 187–195, iSBN: 978-90-77381-57-1.

[17] M. Saito, "A variant of mersenne twister suitable for graphics processors," 2010, submitted.

[18] J. Siek and A. Lumsdaine, "Concept checking: Binding parametric polymorphism in c++," in *First Workshop on C++ Template Programming, Erfurt, Germany*, 2000.

[19] R. Tausworthe, "Random numbers generated by linear recurrence modulo two," *Mathematics of Computation*, vol. 19, no. 90, pp. 201–209, 1965.

UMR 6158 CNRS

RESEARCH CENTRE
LIMOS - UMR CNRS 6158
Campus des Cézeaux
Bâtiment ISIMA
BP 10125 - 63173 Aubière Cedex
France

Publisher
LIMOS - UMR CNRS 6158
Campus des Cézeaux
Bâtiment ISIMA
BP 10125 - 63173 Aubière Cedex
France
http://limos.isima.fr/