Pseudo-Random Number Generation on GP-GPU

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

UMR 6158 CNRS

Pseudo-Random Number Generation on GP-GPU

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

Originally published in: IEEE/ACM/SCS Workshop on Principles of Advanced and Distributed Simulation — June 2011 — pp 146-153
http://dx.doi.org/10.1109/PADS.2011.5936751

Abstract: Random number generation is a key element of stochastic simulations. It has been widely studied for sequential applications purposes, enabling us to reliably use pseudo-random numbers in this case. Unfortunately, we cannot be so enthusiastic when dealing with parallel stochastic simulations. Many applications still neglect random stream parallelization, leading to potentially biased results. Particular parallel execution platforms, such as Graphics Processing Units (GPUs), add their constraints to those of Pseudo-Random Number Generators (PRNGs) used in parallel. It results in a situation where potential biases can be combined to performance drops when parallelization of random streams has not been carried out rigorously. Here, we propose criteria guiding the design of good GPU-enabled PRNGs. We enhance our comments with a study of the techniques aiming to correctly parallelize random streams, in the context of GPU-enabled stochastic simulations.

Keywords: Stochastic Simulations, GP-GPU, Pseudo-Random Number Generators, Random Stream Parallelization

* 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
§ Clermont Université, Université Blaise Pascal, 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

Stochastic simulations are computationally greedy applications. Consequently, domain experts, trying to decrease over and over again their simulation execution time, look for new solutions. For about five years now, we have seen a growing interest in General-Purpose computation on Graphics Processing Units (GP-GPU) through the simulation community. GP-GPU offer a startling amount of computational power at an incredibly low cost, compared to supercomputers. The introduction of this new kind of architecture implies not only the rewriting of the simulation algorithms but also the tools they use. In this article, we have chosen to focus on Pseudo-Random Number Generators (PRNGs), which are the basis of any stochastic simulation.

Sequential PRNGs have been studied for a long time and finding a good quality PRNG to use in a sequential application has not been a problem for more than a decade. A lot of studies have been accomplished to characterize the statistical quality of PRNGs, leading to several tests batteries software. Nowadays, reference test suites are well-known and used to assess PRNGs. But, just like a prime PRNG should not be considered outside the scope of the application it feeds, we should consider different studies according to the domain implied. For instance, cryptographic applications developers ought to base their choice on the NIST test battery [1], whereas simulationists should use TestU01 [2]. These two test batteries can be considered as a standard for their respective domains at the time of writing.

According to [3, 4], a PRNG should perform well on a single processor before being parallelized. Yet, statistical quality is a necessary but not sufficient condition when selecting a PRNG to use in a parallel context: indeed, parallel streams should be independent. Thus, providing high quality random numbers becomes even more difficult when dealing with parallel architectures. We have to take into account the parallelization technique: how will we partition random streams among parallel processing elements (threads or processors)? How will we ensure the independence between parallel streams in order to prevent the involved simulations from producing biased results? The major problem concerning independence between random streams is that no mathematical proof exists to ensure it. However, some studies lay out well-known techniques to spread random streams through parallel applications [3, 5, 6]. They try to ensure the highest independence between random streams using different strategies. We will consider in this article how and whether these techniques could be implemented on GPU.

Apart from the parallelization technique, another point relies directly on the architecture where the involved stochastic simulations run. If we consider a GP-GPU environment, a new difficulty comes into play: harnessing the power of the device requires a rather good knowledge of GPUs. With recent programming frameworks like CUDA (Compute Unified Device Architecture) or OpenCL (Open Computing Language), merely anyone can develop applications for GPUs, but obtaining the announced performance gain implies a higher level of understanding. Most of the work and considerations exposed here rely principally on NVIDIA CUDA solutions. We are also working with the emerging OpenCL standard [7], but the latter is still not robust enough in our opinion. Its current performances are slower than what you could obtain with CUDA [8]. However, we feel that this standard deserves our interest, and should take on an important part of our further work.

In this article, we will name as processing elements the elements effectively computing data in parallel. In a CUDA GP-GPU environment, threads will represent these elements, since this framework relies on a thread level logic referred to as SIMT (Single Instruction, Multiple Threads). The latter abstracts a much more standard designation known as SIMD (Single Instruction, Multiple Data). GPUs are based on this parallel architecture kind. Here, each thread must be given different data that will be computed by an identical operation. In the case of parallel stochastic simulations, we need to furnish an independent stochastic stream to each

thread, in order to prevent potential bias that could be introduced otherwise. The questions are then: how are these random streams produced on GP-GPU? And more importantly: how can we ensure that they are independent?

These two main problems led us to think about design guidelines that will hopefully help developers to design better-shaped PRNGs for GPUs. Throughout these few lines, we will focus on random numbers generated and directly consumed on the GPU. In the next sections, some of the cited works have attempted to speed-up generation using the GPU before retrieving random numbers back onto the host. Current CPU-running PRNGs display fairly good performances thanks to dedicated compiler optimizations. For instance, Mersenne Twister for Graphics Processors (MTGP) [9], the recent GPU implementation of the well-known Mersenne Twister [10], is only announced as being 6 times faster than the CPU reference SFMT (SIMD-oriented Fast Mersenne Twister) [11]. Some studies show that the time spent in generating pseudo-random numbers consumes at most 30% of CPU time for some "stochastic-intensive" nuclear simulations [12], but they are very scarce. In view of the small part of the execution time used by most of the stochastic simulations to generate random numbers, it is not worth limiting GPUs usage at the generation task. Therefore, our study is not tied to the parallelization of random number generation algorithms only, and we do not propose any new PRNG in this paper. On the other hand, we share our experience of using stochasticity in GP-GPU-enabled simulations. Considering the previously evoked SIMT model, stochastic simulations with fewer divergent branches in their source code will behave better on GPU. Still, this work does not intend to list examples of simulations that will run efficiently on GPU. Instead, we focus on the parallelization techniques of pseudo-random streams used to directly feed parallel simulation programs running on GPU (applications are called kernels in the CUDA language).

In this study, we will:

- propose GP-GPU specific criteria for PRNGs design;
- suggest requirements for parallelization techniques of random streams on GP-GPU;
- study, according to the previously introduced requirements, the suitability of well-known PRNGs and random streams parallelization techniques for GP-GPU architectures.

Now, let us start with a brief reminder of GPU architecture essentials, so that we can introduce PRNGs implementation on GP-GPU.

2 Random number generation on GP-GPU

The purpose of this section is not to carry out a full survey of GPU-enabled PRNGs propositions in the literature. An attempt to do so can actually be found in [13]. We focus here on PRNGs implementation level on the GPU. The latter cited survey distinguishes three implementation levels, mapped on the CUDA thread hierarchy: threads, blocks of threads and grid of blocks. These strategies directly impact the PRNG implementation, so as the parallelization technique coupled with it. Let us introduce them to understand the technical prerequisites about the two subjects we are tackling in this study.

Obviously, new PRNG algorithms have to take advantage of GPU intrinsic properties. For simplicity purposes, we will briefly introduce the major concepts that rule GPU architectures, that is to say: heterogeneous memory hierarchy and thread organization. Notions described in this paragraph are represented in Figure 1. On the one hand, the thread organization is here to maximize the performances of the application. Threads are bundled into blocks of threads to be affected to one of the Streaming Multiprocessor (SM) of the GPU. SMs can mostly be

considered as GPU equivalents to CPU cores. The important point here is that they have their own thread scheduler, championing threads which data are available. Selected threads then run on Streaming Processors (SP): the processing units of SMs. It is here that the matching notion, namely heterogeneous memories, comes into play. Threads can access several memories displaying various capacities, which we will discuss further in Section 3. For now, we just need to keep in mind memory areas hierarchy: i.e. the classification of memories based on their quickness and visibility from threads. From the quickest to the slowest, threads can access: registers, shared memory, local memory and global memory. When registers and local memory are dedicated to a single thread, shared memory is visible to all the threads within a block, while every thread can reach global memory.

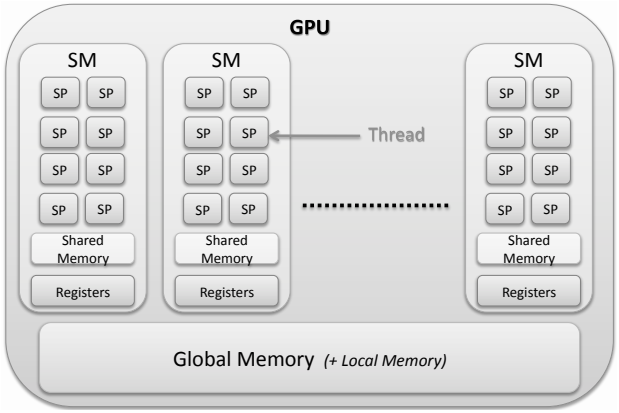

Figure 1: Simple representation of the major elements of a GPU

This memory organization highly affects the PRNG performances. With the first considered implementation level, using a generator per thread, the internal state of the PRNG has to be saved in each thread's local memory. Most of the time, internal states are formed by several elements contained in arrays. Now, CUDA related works, like [14], specify that arrays declared within a thread are stored in the local memory. Although its name indicates a thread scope, please note that local memory is in fact a subset of global memory, and suffers consequently from the same slowness. This area is also allocated to threads when their register set is spent, since registers are available in limited quantities on GPUs.

Equivalently, with the last implementation level, a PRNG for all threads, that is to say a grid-level scope, the global memory is solicited to store the state of the unique PRNG of the application. Each thread draws a number and updates its component of the state in global memory. These two approaches make heavy use of global memory, which has the advantage of being persistent across kernel launches within the same application. Yet, this area is quite slow: it implies a 400 to 800 clock cycle latency because it is not cached [15]. The second implementation scope, the block level, is the only one left to discuss. Every thread in a block can access a shared memory area. PRNG algorithms can consequently store their internal state in this space, enabling every thread of the block to update it. Shared memory is implemented on-chip and is consequently as fast as registers. Thus, PRNGs implemented at a block level will not suffer from the memory latency induced by slow global memory accesses. The three implementation scopes detailed in this section are sketched in Figure 2:

So far, we have described PRNG algorithms, but we also found interesting libraries propositions. We noted two main proposals in this domain: CURAND and Thrust::random. They both aim to provide a straightforward interface to generate random numbers on GPU. Intro-

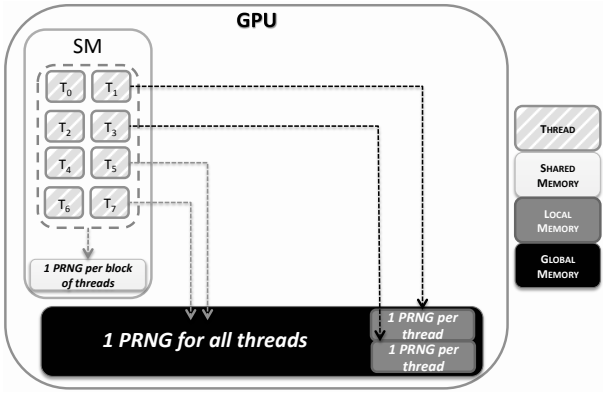

Figure 2: PRNGs implementation scopes and their location in the different memory areas of a GPU

duced in the latest version, at the time of writing, of the CUDA framework, CURAND [16] has been designed to generate random numbers in a straightforward way on CUDA-enabled GPUs. The main advantage of CURAND is that it is able to produce both quasi-random and pseudo-random sequences, either on GPU or on CPU. The quasi-RNG and pseudo-RNG implemented are respectively Sobol and XorShift [17]. The Application Programming Interface (API) of the library stays the same no matter which kind of RNG is selected and the platform on which the application is run on.

Thrust::random is part of a GPU-enabled general purpose library called Thrust [18]. This open-source project intends to provide a GPU-enabled library equivalent to standard general-purpose C++ libraries, such as STL or Boost. Classes are split through several namespaces, of 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) [19] and a Subtract With Borrow (SWB) [20, 21]. Although the latter PRNG is mentioned as Subtract With Carry in Thrust::random documentation, Marsaglia's original proposition is known as SWB. In spite of the known flaws laid out by all these generators, the library offers simple ways to combine them into better quality randomness sources, like L'Ecuyer's Tausworthe combined generators [22].

3 GP-GPU specific requirements for PRNGs

As a result of the SIMD parallelism between threads and of their graphics processors legacy, which we superficially discussed in Section 2, GP-GPUs are not equivalent to a set of standard processors used in parallel. These particularities introduce some constraints that need to be satisfied if we do not want to see the overall simulation performance dramatically drop. Thus, we will introduce in this section the requirements we find compulsory for a PRNG to run efficiently on a GP-GPU architecture.

The main goal targeted when using GP-GPUs is to obtain greater speed-ups. But this kind of device has not been primarily designed to support general computations and is more inclined to realize some arithmetic operations. Nowadays GPUs display different performances with single and double precision floating point numbers. For instance, the previous generation of NVIDIA supercomputing-dedicated GPU, the Tesla T10, was known to deliver ten times

as much computation power when dealing with single precision floating point numbers rather than double. Even if the actual cutting-edge GPU, the NVIDIA new generation Fermi T20, has considerably reduced the gap between these two precisions, it would be wise to remain cautious before using double precision operations on GPU. Most of the time, single precision floats are sufficient enough to handle random values contained in $[0 ; 1[$ and should consequently be favoured. This proposition leads us to our first criterion: *single precision floating point numbers should be preferred throughout the GP-GPU random number generation algorithm.*

Another legacy of graphics processors is the heterogeneous memory organization. To complete what has previously been said on this subject, let us recall that several memory areas are reachable by threads running on a GPU. These memories capacities, some quick and large, others less so, depend on two characteristics. First, the more threads can reach a memory area, the slower it is. Indeed, registers allocated to a single thread are the quickest memory this thread will be able to communicate with. At the same efficiency level, we find shared memory, reachable by a relatively small amount of threads, all belonging to the same block, and so running on the same core of the GPU. On the other hand, every thread running on the GPU, regardless of which core they are located on, can access a wide memory area, commonly called global memory. Therefore, this latter area is far slower than its previous counterparts. Second, read-only memories are quicker since they can fully benefit from cache mechanisms, contrary to read-write memories. In the light of these memory constraints, it is obvious that GPU PRNGs should be designed with particular attention to sparing costly memory accesses. Commonly, static parameters will take place in read-only areas, whereas dynamic elements such as state vectors will be handled at the thread or thread group level. In a more formal way, the following is another criterion of good design: *the algorithm should be designed in a way to avoid global memory accesses.*

Taking into account the particularities of GPUs and their architecture, we have proposed two new requirements for PRNGs to run efficiently on such devices. They are summed up hereafter:

1. *single precision floating point numbers should be preferred throughout the GP-GPU random number generation algorithm*
2. *the algorithm should be designed in a way to avoid global memory accesses*

4 GP-GPU specific requirements for random streams parallelization

The literature is full of references describing the profile of what a good usage of parallel PRNGs should be. For example, [3, 4] list requirements that should meet any sequential or parallel PRNG. GPUs are particular parallel architectures. So any PRNG running on this kind of device should, at least, meet the requirements enumerated in the previous references. In this section, we will successively check how these criteria can be adapted to GPU architectures.

Emphasizing parallel PRNG performances, [3] noticed that *each processor should generate its sequence independently of the other processors.* We consider, indeed, that every processing element should have its own stochastic stream at its disposal. This condition must be satisfied first, not only for efficiency, but especially because GPU-enabled stochastic simulation parallelization principles rely on it. First, it is a necessary, but not sufficient, condition to fulfil to ensure a higher independence of stochastic streams feeding different replications of a simulation. Second, considering a single replication, the SIMT parallelism level leads threads to compute their own data sets, including their own stochastic stream. Thus, our first requirement concerning random streams parallelization can be expressed as follows: *each thread should dispose of its own random sequence.*

As we explained previously, GP-GPU programming frameworks offer a thread scope rather than a processor one. The threads in use for GP-GPU propose an abstraction of the underlying architecture. They are concurrently running on the same device and handle their own local memory area. Thread scheduling is at the basis of GPU performances. Memory accesses are the well-known bottleneck of this kind of device. Indeed, running a large amount of threads in turn allows GPUs to bypass memory latency. There should always be runnable threads while others are waiting for their input data. Disregarding the effective number of processors, we theoretically say that the more threads you have, the better your application will leverage the device. Applications need to be written to use the maximum number of threads, but also to scale up transparently when the next GPU generation will be able to run twice as many threads as today. So, in accordance with [3] who advocates that *the generator should work for any number of processors*, our second GP-GPU specific requirement for parallelization techniques of random streams is that *it must be usable for any number of GP-GPU threads.*

Returning to the original requirements, we then find [3] states that *parallel random streams produced should be uncorrelated.* This criterion is related to both PRNG intrinsic properties and to the parallelization technique set up. We previously stated that a PRNG candidate to parallelization should first perform well on a single processor. Thus, we will not take its intrinsic qualities into account here. However, no matter the worth of the used PRNG, the parallelization techniques must be used carefully, as will be shown in the next section. Please note that this requirement is neither affected by GP-GPU architectures nor by programming frameworks. As a result, we will just recall it without modifying its expression.

[3] also noted that *the same sequence of random numbers should be produced for different numbers of processors, and for the special case of a single processor.* Here, we understand that the PRNG output on each processor should not depend on the number of processors used. This point is very important and must be treated carefully when choosing a parallelization technique. For example, in a distributed environment containing several processors, a scheduler can govern execution. Depending on the scheduler algorithm and on the global system charge, parallel executions of different parts of a simulation might not execute in the same order. In such a case, it is compulsory for the PRNG output to be independent from the order in which simulations parts may run. If this requirement is not met, reproducibility of simulations executions is no longer ensured. We can do this for games, but not for scientific applications. Reproducibility is needed when dealing with stochastic simulations, in order to debug a problematic case raised by a particular random stream for instance. We also think about design of experiments for simulations, where reproducibility is mandatory to isolate parameters variations impact in the results.

In the case of GP-GPU, we find exactly the same problem at the thread level. These entities are also scheduled, not atomically but by bundles. Fortunately, both threads and their bundles own a unique identifier allowing us to distinguish them among executions. Thus, if a parallel random stream is only bound to the unique identifier of a thread, according to our first requirement, output will be reproducible through multiple executions. Therefore, we can obtain the necessary bijective relation between a T_i thread and an SS_i stochastic stream, as stated in Figure 3:

Finally, we can write our last requirement in such a way: *when the status of the PRNG is not modified, the sequence of random numbers generated for a given thread must be the same no matter the number of threads and no matter of threads scheduling.*

Let us now sum up the requirements targeting GPUs we highlighted in this part, continuing the enumeration started in Section 3:

3. *each thread should dispose of its own random sequence*

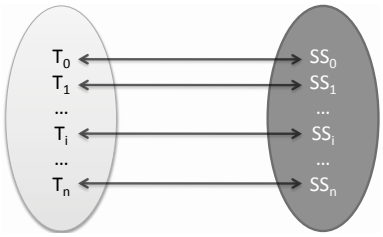

Figure 3: Bijective relation between threads and stochastic streams

4. *the parallelization technique must be usable for any number of GP-GPU threads*
5. *the parallel random streams produced should be uncorrelated*
6. *when the status of the PRNG is not modified, the sequence of random numbers generated for a given thread must be the same no matter the number of threads and no matter of threads scheduling*

5 Do classical random streams parallelization techniques fit GP-GPU well?

This section presents the main techniques used to distribute random streams between processing elements. A more exhaustive and dedicated survey can be found in [6]. We will also study in this section how random streams parallelization techniques can be adapted to GPU architectures, depending on their ability to fulfil the previously introduced requirements (Sections 3 and 4).

The Leap Frog is the way to partition a random stream like a deck of cards. Random numbers are allocated in turn to processors like cards would be dealt to players. Pragmatically, let each processor hold an i identifier. The latter processing element will build a Y_i substream from an X original random stream such as $Y_i = \{X_i, X_{i+N}, \ldots, X_{i+kN}\}$, with N equal to the number of processors. This technique is not quite adapted to split random streams on GP-GPU since it does not satisfy the last constraint expressed in Section 4. In fact, if the number of threads changes, the subsequence affected to each thread will be different. This situation is shown in Figure 4. Now, the number of threads for an application is bound to the underlying device: GPUs can run a different number of threads concurrently, depending on their architecture generation. As a result, we would not be able to ensure the reproducibility of a simulation from a GPU to another, even if we initialized the PRNG with the same parameters. Furthermore, some bugs induced by random draws might not appear on developer's device, while they would on the user's.

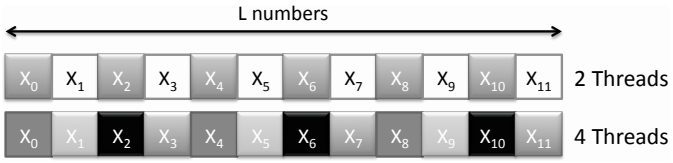

Figure 4: Different threads numbers leading to different random substreams through the Leap Frog method

The solution would be to implement the PRNG at a level with a constant thread number. The CUDA framework handles constant-sized bundles called warps, which always contain 32

threads at the time of writing. They are in fact a subdivision of blocks of threads, and access consequently the same shared memory area. Nowadays, a great number of GPUs do not have enough shared memory to store a PRNG status per warp. However, the newest GPU generations offer larger shared memory spaces that could potentially enable us to use Leap Frog, following this idea.

Apart from any architecture consideration, please note that Leap Frog is not adapted to every kind of PRNGs. [4] has established that this technique could be flawed in some cases, because of spectral tests. We particularly insist on what an unsafe technique it is when used with LCGs. Serious stochastic simulations should ban simple LCGs since they have structural flaws [23].

The second technique to be referenced in [3] is Sequence Splitting (also known as fixed spacing or blocking). It consists in allocating continuous and equally sized blocks from the original random stream to form substreams. This technique implies the knowledge of how many numbers each thread will consume at most. Indeed, knowing that each thread consumes at most L random numbers, then the first L numbers will be attributed to the first thread, the L following to the second thread and so forth. Following the previous formalism, we have $Y_i = \{X_{iL}, X_{iL+1}, \ldots, X_{(i+1)L-1}\}$. This numbers repartition is described in Figure 5.

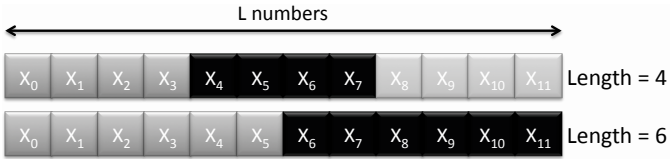

Figure 5: Two random streams parallelizations based upon Sequence Splitting with two different sub-sequences lengths

Efficient Sequence Splitting relies on a particular feature of the PRNG called Jump Ahead, or Skip Ahead. Instead of unrolling the generator, it enables us to compute any given advanced state of the random sequence without processing the previous ones (for example $X_{(i+1)L}$ can be computed directly from X_{iL}). Here, we discern two categories of algorithms. Some PRNGs contain an algorithm able to realize Jump Ahead, thanks to intrinsic properties of the PRNG [24]. This allows us to reach any part of the sequence in equal time, regardless of the destination point. Let this technique be considered as Intrinsic Jump Ahead hereafter. The other solution is to emulate Skipping Ahead: to do so, we have to compute an advanced state by processing step by step the previous ones (for example, starting from $X_{iL}, X_{(i+1)L}$ can be computed by running the PRNG L times in a sequential way in order to get $X_{iL+1}, \ldots, X_{(i+1)L-1}$ and finally $X_{(i+1)L}$). In fact, whichever PRNG you use, you can unfold the sequence to the desired point, and store the state vector at this point in order to be able to load it later. Such a vector is named seed status in [13], since it is able to set a generator in a predefined state. Emulated Jump Ahead can become very costly though: indeed, the further you need to go in the sequence, the more time it takes to compute the seed status.

To conclude on Sequence Splitting, if an intrinsic Jump Ahead algorithm is available for the involved PRNG, Sequence Splitting is a very good approach for GP-GPUs. However, as far as we know there are few ports of algorithms from this family to GP-GPU. On the other hand, Emulated Jump Ahead is not GPU-compliant because statuses computation is a purely sequential operation (we need X_n to compute X_{n+1}). As a consequence, different threads may display different lengths of time for these computations, thus decreasing the SIMD parallelism overall gain. To solve this problem, we propose to pre-compute substreams on the host side, store the seed status at each substream starting point and then transfer all these statuses to the

device. In the past, Sequence Splitting was also considered as unsafe by several studies [25, 26], but this remark applies to any PRNGs that showed potential long-range correlations, such as congruential ones.

We have just seen that emulated Jump Ahead led to expensive calculations in order to compute an initial status for each sub-sequence. Another approach based upon probabilities is known as Random Spacing. The idea is to avoid Emulated Jump Ahead cost by choosing random statuses in the base stream, and set them as initial statuses for the resulting substreams as represented in Figure 6. Although Random Spacing spares us from solving the Jump Ahead problem, it should not be used without care. [27] have given the minimum distance between N sub-sequences which status were randomly chosen: it is in average equal to $1/N^2$ multiplied by the period length. More precisely, the probability of overlapping between N sequences of length L, issued from a PRNG of period P is equal to: $1 - (1 - NL/(P-1))^{(N-1)}$. That is equivalent to $N(N-1)L/P$ when $NL/(P-1)$ is in the neighbourhood of 0.

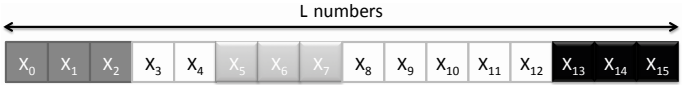

Figure 6: Random Spacing creation of three sub-sequences of equal length but differently spaced from each other

Overlapping risks become sizeable with short period PRNGs. All the PRNGs tested by Pierre L'Ecuyer and Richard Simard in [2] display periods P from 2^{24} to 2^{131072}. Most of them have $\log_2 P$ of the order of a few dozens. Now, given that nowadays longest simulations can consume up to several hundreds of billions of random numbers, ($L = 10^{11}$), a hundred of such replications ($N = 100$) leads to a $N(N-1)L$ of approximately 10^{15}, i.e. 2^{50}. The probability to see an overlapping between two sub-sequences issued from a PRNG of period P far bigger than 2^{50} is negligible. Nonetheless, from 20 LCGs PRNGs tested in [2], 14 laid out an overlapping probability greater than 99.9% (period less than 2^{47}). Only 3 generators have an overlapping probability less than 0.01% ($\log_2 P > 59.7$). In the same way, widely spread PRNGs such as Comblec88 of period $P = 2^{61}$ (Combined LCGs), are on the acceptance borderline for this technique. They are shipped with several renowned software packages though, including: RANLIB, CERNLIB, Boost, Octave and Scilab [2].

As a conclusion, this technique fits GP-GPUs well, but the risk of overlapping between sub-sequences must be evaluated according to the number and the length of the sub-sequences and to the period of the PRNG used. If we select generators with large periods, such as WELLs and Mersenne Twister, this risk is negligible.

Techniques presented so far tried to split a single stream into several substreams. Another approach consists in using several declinations of the same PRNG: each generator has the same structure and generation mechanism with a unique parameter set, called parameterized status hereafter. By allocating a parameterized status per processor, parameterization ensures our first requirement, namely that each thread should dispose of its own sequence. This situation is described in Figure 7. We always have to remember that this cannot ensure our third requirement: strict independence between parallel random streams.

Although no mathematical proof can establish this independence, some implementations of Parameterization are safe according to the current state of the art [28]. We especially think to the Dynamic Creator (DC) algorithm [29] coming along with the Mersenne Twister for Graphics Processors (MTGP) PRNG [9], a quality-proven work benchmarked in [13]. DC integrates a unique identifier, which belongs to the parameterized status of MTGP. This identifier becomes a part of the characteristic polynomial of the matrix that defines the recurrence of the PRNG.

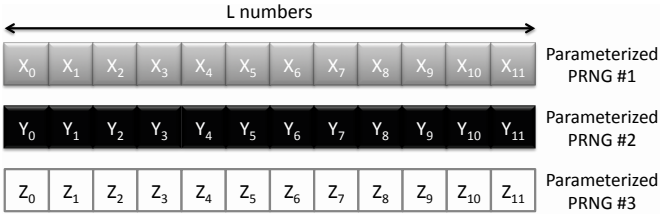

Figure 7: Three parameterized PRNGs producing three highly independent random sequences

Two identifiers will consequently lead to two different parameterized statuses. Furthermore, DC ensures that the characteristic polynomials we obtain are mutually prime and the authors assert that the random sequences generated with such distinct parameterized statuses will be highly independent, even if, as mentioned before, this fact cannot be mathematically proven.

This technique displays some constraints making it difficult to port on GPUs. Unfortunately, few PRNGs propose it intrinsically. Some, such as LCGs, are even recognized as bad candidates for Parameterization [30]. In addition, storing seed statuses being already problematic, we can scarcely imagine spending vast amounts of memory to store a parameterized status per thread.

Every technique introduced in this section, presents advantages and drawbacks. Most of them are related to the chosen PRNG. Depending on the application and environment you own, you might be forced to select a PRNG knowing it has some flaws in particular cases. In this way, Table 1 states PRNG kinds and parallelization techniques that work well together:

Table 1: Summary of the potential PRNG/Parallelization technique associations

Technique	***Preferred PRNGs***	***PRNGs to avoid***
Leap Frog	None (disable reproducibility)	Linear generators
Sequence Splitting	Intrinsic Jump-Ahead compliant	Emulated Jump-Ahead
Random Spacing	Large period	Short period
Parameterization	MT family	LCG

6 Conclusion

We have seen that more and more applications, and especially stochastic simulations, tend to take advantage of recent GP-GPU architectures in order to improve their performance. However GPU computing needs to offer the same tools as other platforms. Good quality PRNGs belong to this category. This paper has shown how difficult it could be to generate good quality pseudo random numbers on GP-GPUs. Indeed, it implies taking into consideration two different domains: GP-GPU programming and PRNG parallelization techniques. Issuing a PRNG that can produce independent stochastic streams when used in parallel is a first hurdle that not all PRNGs can get over. When you have at your disposal one that fulfills this requirement, it has to be ported to GP-GPU. It means that you need to think about a GPU parallelization of your PRNG, if it is not already available.

This paper introduced the main problems that you will be faced with when trying to port your stochastic simulations on GP-GPU. To avoid these difficulties, and above all, errors and

performance drops that could result from the use of a hazardous GPU-enabled PRNG, we first proposed PRNGs criteria dedicated to GPUs. Then we defined parallelization techniques requirements in order to make these PRNGs fit GPU peculiarities. They sum up the experience accumulated in our research team concerning GPU-enabled stochastic simulations.

We encountered lots of parameters brought up by PRNGs and GPU programming. As long as it can dramatically impact both the overall performance of the simulation and the quality of its results, it might be a good point to propose a straightforward API to use well-defined PRNGs on GPUs. In this way, libraries laid out in Section 2 represent in our mind an elegant manner to do so. They allow every user to easily call PRNG functions without wasting time on how parameters should be set. In the same way, they enable advanced users to adjust parameters in their own fashion. As a matter of fact, further works of ours should target this kind of development, embedding PRNGs following the presently established requirements into a GPU-enabled library with a unified API.

Acknowledgements

Special thanks to Susan Arbon, Natacha Vandereruch and Florian Guillochon for their careful reading and useful suggestions on the draft.

References

[1] A. Rukhin, J. Soto, J. Nechvatal, M. Smid, E. Barker, S. Leigh, M. Levenson, M. Vangel, D. Banks, A. Heckert, J. Dray, and S. Vo, "A statistical test suite for random and pseudorandom number generators for cryptographic applications," NIST, Tech. Rep., 2001.

[2] 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.

[3] P. Coddington, "Random number generators for parallel computers," NHSE, Tech. Rep. 2, 1996.

[4] 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.

[5] ——, "Don't trust parallel monte carlo!" in *Proceedings of Parallel and Distributed Simulation PADS98*. IEEE, 1998, pp. 82–89.

[6] 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.

[7] Khronos OpenCL Working Group, "The opencl specification," Khronos Group, Specification 1.1, September 2010.

[8] K. Karimi, N. Dickson, and F. Hamze, "A performance comparison of cuda and opencl," http://arxiv.org/abs/1005.2581v2, August 2010, submitted.

[9] M. Saito, "A variant of mersenne twister suitable for graphics processors," http://arxiv.org/abs/1005.4973, 2010, submitted.

[10] M. Matsumoto and T. Nishimura, "Mersenne twister: A 623-dimensionnally equidistributed uniform pseudorandom number generator," *ACM Transactions on Modeling and Computer Simulations: Special Issue on Uniform Random Number Generation*, vol. 8, no. 1, pp. 3–30, January 1998.

[11] M. Saito and M. Matsumoto, "Simd-oriented fast mersenne twister: a 128-bit pseudorandom number generator," in *Monte Carlo and Quasi-Monte Carlo Methods 2006*, A. Keller, S. Heinrich, and H. Niederreiter, Eds., vol. 2. Springer Berlin Heidelberg, 2008, pp. 607–622.

[12] L. Maigne, D. Hill, P. Calvat, V. Breton, R. Reuillon, Y. Legre, and D. Donnarieix, "Parallelization of monte carlo simulations and submission to a grid environment," *Parallel processing letters*, vol. 14, no. 2, pp. 177–196, 2004.

[13] 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.

[14] D. Kirk and W. Hwu, *Programming Massively Parallel Processors*. Morgan Kaufmann, 2010, iSBN: 978-0123814722.

[15] NVIDIA, *NVIDIA CUDA Programming Guide Version 3.2*, October 2010.

[16] ——, *CUDA CURAND Library*, NVDIA Corporation, August 2010.

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

[18] J. Hoberock and N. Bell, "Thrust: A parallel template library," https://thrust.github.io/, 2010, version 1.3.0.

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

[20] 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.

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

[22] P. L'Ecuyer, "Maximally equidistributed combined tausworthe generators," *Mathematics of computation*, vol. 65, no. 213, pp. 203–213, 1996.

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

[24] H. Haramoto, M. Matsumoto, T. Nishimura, F. Panneton, and P. L'Ecuyer, "Efficient jump ahead for f2-linear random number generators," *INFORMS Journal on Computing*, 2007.

[25] A. De Matteis, J. Eichenauer-Herrmann, and H. Grothe, "Computation of critical distances within multiplicative congruential pseudorandom number sequences," *Journal of Computational and Applied Mathematics*, vol. 39, no. 1, pp. 49–55, 1992.

[26] A. De Matteis and S. Pagnutti, "Critical distances in pseudorandom sequences generated with composite moduli," *International Journal of Computer Mathematics*, vol. 43, no. 3, pp. 189–196, 1992.

[27] P. Wu and K. Huang, "Parallel use of multiplicative congruential random number generators," *Computer Physics Communications*, vol. 175, no. 1, pp. 25–29, 2006.

[28] M. Mascagni and A. Srinivasan, "Parameterizing parallel multiplicative lagged-fibonacci generators," *Parallel Computing*, vol. 30, no. 7, pp. 899–916, 2004.

[29] M. Matsumoto and T. Nishimura, "Dynamic creation of pseudorandom number generators," in *Monte Carlo and Quasi-Monte Carlo Methods 1998*. Springer, 2000, pp. 56–69.

[30] A. De Matteis and S. Pagnutti, "Parallelization of random number gemerators and long-range correlations," *Numerische Mathematik*, vol. 53, pp. 595–608, 1988.

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/