Contributions to parallel stochastic simulation: Application of good software engineering practices to the distribution of pseudorandom streams in hybrid Monte-Carlo simulations

Jonathan Passerat-Palmbach

Contributions to parallel stochastic simulation: Application of good software engineering practices to the distribution of pseudorandom streams in hybrid Monte-Carlo simulations

by

Jonathan Passerat-Palmbach

A Thesis

submitted to the Graduate School of Engineering Sciences

of the Blaise Pascal University - Clermont II

in fulfilment to the requirements for the degree of

Docteur of Philosophy

in Computer Science

Supervised by David R. C. Hill

at the LIMOS laboratory - UMR CNRS 6158

publicly defended on October, 11^{th} 2013

Committee:

Reviewers:	Pr. Michael Mascagni	-	Florida State University, USA
	Pr. Stéphane Vialle	-	Supélec, Campus de Metz, France
Supervisor:	Pr. David R. C. Hill	-	Blaise Pascal University, France
Chairman:	Pr. Alain Quillot	-	Blaise Pascal University, France
Examiners:	Pr. Pierre L'Ecuyer	-	Montreal University, Canada
	Pr. Makoto Matsumoto	-	Hiroshima University, Japan
	Dr. Bruno Bachelet	-	Blaise Pascal University, France
	Dr. Claude Mazel	-	Blaise Pascal University, France

Abstract

The race to computing power increases every day in the simulation community. A few years ago, scientists have started to harness the computing power of Graphics Processing Units (GPUs) to parallelize their simulations. As with any parallel architecture, not only the simulation model implementation has to be ported to the new parallel platform, but all the tools must be reimplemented as well. In the particular case of stochastic simulations, one of the major element of the implementation is the pseudorandom numbers source. Employing pseudorandom numbers in parallel applications is not a straightforward task, and it has to be done with caution in order not to introduce biases in the results of the simulation. This problematic has been studied since parallel architectures are available and is called pseudorandom stream distribution. While the literature is full of solutions to handle pseudorandom stream distribution on CPU-based parallel platforms, the young GPU programming community cannot display the same experience yet.

In this thesis, we study how to correctly distribute pseudorandom streams on GPU. From the existing solutions, we identified a need for good software engineering solutions, coupled to sound theoretical choices in the implementation. We propose a set of guidelines to follow when a PRNG has to be ported to GPU, and put these advice into practice in a software library called ShoveRand. This library is used in a stochastic Polymer Folding model that we have implemented in C++/CUDA. Pseudorandom streams distribution on manycore architectures is also one of our concerns. It resulted in a contribution named TaskLocalRandom, which targets parallel Java applications using pseudorandom numbers and task frameworks.

Eventually, we share a reflection on the methods to choose the right parallel platform for a given application. In this way, we propose to automatically build prototypes of the parallel application running on a wide set of architectures. This approach relies on existing software engineering tools from the Java and Scala community, most of them generating OpenCL source code from a high-level abstraction layer.

Keywords: Pseudorandom Number Generation (PRNG); High Performance Computing (HPC); Software Engineering; Stochastic Simulation; Graphics Processing Units (GPUs); GPU Programming; Automatic Parallelization

Acknowledgements / Remerciements

> “ *Anyone who ever gave you confidence, you owe them a lot.*
>
> — Truman Capote, *Breakfast at Tiffany's*

First of all, I'd like to thank all the members of my committee for having accepted to evaluate my work, read this thesis and attend my defence.

Merci à David, mon directeur de thèse, qui a été si présent durant ces 3 années (et même avant). Il a su trouver la bonne formule pour guider mon travail et me permettre d'atteindre mes objectifs, tant professionnels que personnels. Son soutien permanent en fait un proche que j'ai besoin de quitter pour grandir, mais envers qui ma reconnaissance est sans limite.

Dans le même esprit, je tiens à évoquer Laurent, mon entraîneur qui me suit depuis tant d'années, qui m'a emmené là où je suis sportivement et humainement. Je lui associe mes succès présents et futurs, et où que je sois, il restera une présence inamovible pour mon équilibre.

Après avoir évoqué mes 2 “pères spirituels”, je ne peux qu'évoquer mon père, qui croit tellement en moi depuis toujours, qui m'a donné goût au voyage et à la découverte, et dont le regard sur ce que je fais est si important pour moi.

À ton tour, mère, de recevoir mes pensées. Toi qui m'a poussé vers le sport étant jeune, et qui aujourd'hui est un soutien sans faille, peu importe mes résultats ou mes performances, tout ce que tu fais pour moi n'a pas de prix.

Je me tourne à présent vers les personnes qui ont guidé mes choix, m'ont passionné et donné envie d'aller dans cette direction plutôt qu'une autre. Je pense à Guénal, François, Sylvie, Michel, Olivier, Édith, Murielle, Philippe et Alain. Mais aussi les “grands frères” : Paul, Paul (encore), Romain, Guillaume et Julien.

Cette thèse ne serait pas là sans mes amis présents au quotidien, au travail, à l'entraînement et pour les extras, merci à vous tous de me supporter moi et mon emploi du temps. Un hommage donc à Guillaume, futur associé potentiel, doctorant, machine à écrire et pokéfan; Florian, compagnon de voyage, de billard et de débat philosophiques, dont l'hyperactivité m'épuiserait presque parfois; Jean-Baptiste, mon binôme de toujours, dans les mauvais mais surtout les très bons moments; Lorena, qui m'a donné sa confiance, et fait preuve d'une patience infinie; Nicolas, que j'aime même s'il contribue à tuer des gens, expert en discussion par boîte vocales interposées; Thomas, que j'ai découvert sur le tard, mais qui me fait partager son expérience et surtout ses bonnes histoires; Nicolas (l'autre), soirée foot ou soirée tout court, même délires, mêmes histoires, parfait! Sébastien, ancien co-bureau, actuel co-détenteur du record de présence au labo une veille de Noël; Nathalie, dont je loue la capacité à nous supporter jour après jour; Luc, source intarissable de connaissances informatiques et cascadeur amateur; Jonathan, co-auteur du "bijou", qui m'a transmis un échantillon de sa rigueur légendaire (petit jeu : il y a un double espace dans ce document); Pierre, le jeune, co-bureau plein d'avenir et de talent, a révélé en moi une vocation d'agent immobilier; Romain, ma conscience, même parti je me demande toujours ce que tu penserais; Clément, mon petit frère, polyglotte notoire dans certains champs lexicaux; Christine, que je ne vois que trop peu souvent; Hélène, cuisinière hors-pair, toujours proche de moi malgré la distance; Benoît, premier contact humain sur le campus, premier retard en cours, toujours là; Faouzi, ami des biologistes, déménageur amateur à mes côtés; Wajdi, my bro! ou Dracula, compagnon de nuit au labo, prétend être tunisien, même si tout le monde sait qu'il est norvégien; Yannick, mon premier stagiaire, développeur de talent, globe-trotter, ogre à sushi; Isaure, ma jumelle, pas besoin de parler pour savoir ce qu'il en est; Lionel, éclaireur de tous les chemins que j'emprunte, apprenti-jardinier; Cédrick, l'homme qui défie la chance, me suivrait jusqu'au bout du monde pour se casser un membre; David, mon exemple de motivation, travaille sans relâche mais trouve toujours du temps pour m'emmener dans des soirées improbables; Romina, ces 10 années de correspondance me donnent l'impression que tu es si proche ma Suissesse; Mathilde, fauve insaisissable parfois, énervante souvent, passionnante tout le temps; Audrey, y'a-t-il quelque chose au-delà de meilleure amie pour représenter ton importance ? Sarah, qui a visiblement su trouver les mots et l'accent pour se faire une place, étonne-moi aussi longtemps que tu le souhaites.

Je tiens à présent à remercier mes collègues de travail à l'ISIMA, au LIMOS ou dans d'autres équipes, avec une pensée particulière pour Claude, Bruno et Ivan qui ont joué un rôle majeur dans le travail que je présente. Merci aussi à Loïc, Antoine, Firmin, Violaine, Romuald, Christophe, Corinne, Susan, Béatrice, Françoise et Séverine pour

leur soutien et leur bonne humeur qui ont rendu les journées de travail (et les nuits pour certains) très agréables. Dédicace au passage à tous ceux qui n'ont pas cru en moi, preuve s'il en faut qu'on ne gagne pas à tous les coups.

Pour Elles, je laisse parler Baudelaire...

Vous que dans votre enfer mon âme a poursuivies,
Pauvres sœurs, je vous aime autant que je vous plains,
Pour vos mornes douleurs, vos soifs inassouvies,
Et les urnes d'amour dont vos cœurs sont pleins.

Un mot enfin pour tous ceux que, pressé par le temps, j'ai oublié de citer mais qui ont compté tout au long de mon parcours. Plus généralement, merci à tous pour votre tolérance quant à mes retards chroniques, mais les hommes libres de Jarry ne m'ont que trop inspiré : *"La liberté, c'est de ne jamais arriver à l'heure"* !

Contents

List of Figures

List of Tables

Listings

List of Algorithms


I'LL JUST COPY AND PASTE ALL THE PAPERS I'VE WRITTEN AND MAKE IT MY THESIS!
ctrl-c
ctrl-v
ctrl-c
ctrl-v
ctrl-c
ctrl-v
...AND I'M DONE!
NO, YOU'RE NOT.
ctrl-z?
JORGE CHAM © 2009
WWW.PHDCOMICS.COM

Introduction

> “ *Not so! Alas! Not so. It is only the beginning.*
>
> — Bram Stoker, *Dracula*

Pseudorandom Numbers for Parallel Stochastic Simulations

The need to reproduce runs of simulations forces the simulation community to champion PseudoRandom Number Generators (PRNGs) as a random source for their applications. However, this kind of random number generation algorithm is sensitive to the way its output random stream is partitioned among computational elements when used in parallel. Consequently, we need highly reliable Random Number Generators (RNGs) and sound partitioning techniques to feed such applications. From the practitioner point of view, sound partitioning techniques of stochastic streams must be employed in the domain of parallel stochastic simulations.

Parallel and Distributed Simulation (PDS) is an area where extensive research of effective solutions has been developed. Deterministic communication protocols for synchronous and asynchronous simulation have been studied to avoid deadlocks and preserve causality and the principle of determinism. When considering stochastic simulations, the pseudorandom numbers must be generated in parallel, so that each Processing Element (PE) can autonomously obtain its own stream of pseudorandom numbers independent of other PEs. If independence is not established, the intrinsic statistical qualities are not guaranteed anymore, and the consequences are much more severe because the quality of the results can be flawed.

GPUs and the Emergence of Hybrid Computing for Simulation

Recent developments sometimes use trillions of pseudorandom numbers necessitating the use of modern generators [Maigne et al., 2004]. In this context, the problem is that

the execution time of the simulation can be prohibitive without parallel computing. Lately, it has been possible to reduce the computation time of the heaviest simulations based on generalist graphics processors: the GP-GPUs (General Purpose Graphics Processing Units), also named GPUs for the sake of simplicity. These devices open up new possibilities for parallelization, but they also introduce new programming difficulties. Actually, GPUs behave as SIMD (Single Instruction, Multiple Data) vector processors. Such architectures are usually not very convenient to develop on for most developers as they are not familiar with the way they work. In order to democratize GPU programming among the parallel developers community, NVIDIA, the leading vendor of GPUs for scientific applications, has introduced a new programming language called CUDA (Compute Unified Device Architecture) [Kirk and Hwu, 2010]. CUDA exposes threads instead of vectors to developers. As a result, GPUs programming has become more similar to multithread programming on traditional CPUs.

Still, there are still lots of constraints bound to GPUs, going from their particular memory hierarchy to thread scheduling on the hardware. Moreover, efficient developments must actually harness both CPU and GPU at the same time to balance the workload on the most appropriate architecture for a task. This paradigm is known as hybrid computing.

All these parameters make it difficult for non-specialists to leverage the computing power of GPUs. In addition, they need to handle correctly the distribution of pseudorandom numbers on the device. Theoretical concerns regarding the use of pseudorandom numbers in parallel environments are known for a while and it is interesting to study how well they cope with recent parallel platforms, such as GPUs.

On the need for good software engineering practices

CPU-enabled simulations can take advantage of a wide range of statistically sound PRNGs and libraries, but we still need quality random number generators for 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 [Bradley et al., 2011] bring up random number generation on GPU. Many strategies were adopted, but few of them perform well enough to distinguish themselves. It is also very difficult to select the right PRNG to use since both the statistical quality of the PRNG, and the quality of the GPU implementation need to

be considered for the overall performance of the application not to seriously drop.

Recent development frameworks like CUDA enabled much more ambitious developments on GPU. We can now make use of some software engineering techniques, such as Object-Oriented programming or generic programming within GPU applications. This introduces a new scope in GPU-enabled applications development. As GPU-enabled software becomes more complex, it also needs enhanced tools to rely on. Thus, PRNGs implementations on GPU cannot be achieved in non-structured software development environments anymore. We need high-level software tools to smoothly integrate PRNGs in existing developments.

In this thesis, we will also put software engineering considerations at the heart of our work and exploit successively Object-Oriented programming, generic programming and Aspect-Oriented programming to serve the needs of parallel stochastic simulation on GPUs and manycore architectures. On the one hand, these tools can fill the gap displayed by recent technologies such as GPU programming. We will present, for instance, an implementation based upon generic programming in C++ to overcome the lack of Object-Oriented features of old NVIDIA CUDA-enabled GPUs. On the other hand, techniques such as Aspect-Oriented programming will enhance existing tools and allow us to introduce extensions to the language without having to write a compiler or a Domain Specific Language (DSL) [Van Deursen et al., 2000]. Finally, Object-Oriented modelling and programming will be involved when it comes to interface with existing developments, or propose counterparts to standard libraries such as those issuing with the Java Development Toolkit.

Organization of the thesis

This thesis studies the efforts that have to be made to port a parallel stochastic simulation on GPU, focusing on NVIDIA hardware and the associated CUDA technology. More particularly, it shows the need for good software engineering practices when taking care of pseudorandom numbers distribution across parallel platforms such as GPUs (Chapter 1).

We will start by proposing a review of the current literature regarding Pseudorandom Number Generation in parallel and distributed environments. This will summarize the techniques employed to correctly distribute pseudorandom streams across Processing Elements (PEs), with a particular focus on GPUs (Chapter 2).

Having surveyed the behaviour of Pseudorandom Number Generators (PRNGs) in parallel environments, and particularly on GPU, we will propose theoretical guidelines to lead the design and implementation of these tools on GPU platforms. This advice takes into account the specificities of the architecture of GPUs, such as their memory hierarchy and thread scheduling (Chapter 3).

Then, we will apply software engineering techniques to implement these theoretical guidelines on both GPU and manycore architectures. Our GPU-enabled development is a C++/CUDA library named ShoveRand that uses Template Metaprogramming to check design constraints at compile time. We target manycore architectures through Java task frameworks. To do so, we introduce TaskLocalRandom, a Java class that handles pseudorandom stream distribution across the tasks of a Java parallel application (Chapter 4).

The presentation of a stochastic simulation model where some of these developments have been integrated will follow. Our main simulation application is a chromosome folding model entirely running on GPU. It is a first parallel version aiming at improving an initial sequential model which encounters troubles to evolve in some configurations (Chapter 5).

We will eventually discuss a major issue when it comes to parallelize an application: choose the right parallel platform and programming languages. Our final chapter studies the possibility to automatically build prototypes of parallel applications targeting various platforms for free, with the help of software engineering tools from the literature. This strategy obviously needs to be combined with a good quality software toolchain, especially when it comes to Pseudorandom Number generation. Through this approach, the main input of this work about correct Pseudorandom Number distribution across parallel environments is reinforced (Chapter 6).

We will now start by stating the context and problems studied in this thesis in Chapter 1.

Chapter 1

Parallel Pseudorandom Numbers at the Era of GPUs: The Need for Good Software Engineering Practices

---

> “ *[...] I estimate that even if fortune is the arbiter of half our actions, she still allows us to control the other half, or thereabouts.*
>
> — Niccolò Machiavelli, *Il Principe*

Contents

1.1 Parallel Stochastic Simulation and Random Numbers

Stochastic simulation is now an essential tool for many research domains. They can require a huge computational capacity particularly when we design experiments to explore large parameter spaces [Kleijnen, 1986]. New technologies in HPC and networking, including hybrid computing, have very significantly improved our computing capacities. The major consequence is the increased interest for parallel and distributed simulation [Fujimoto, 2000].

Random number generators (RNGs) are mainly used in simulation to model the stochastic phenomena that appear in the formulation of a problem or to apply an iterative simulation-based problem solving technique (such as Monte Carlo simulation [Gentle, 2003]) or coupling between simulation and Operations Research optimization tools.

Reproducing experiments is the essence of science, and even if this is not always necessary, in the case of stochastic simulation, the need for reproducible generators is essential for various purposes, including the analysis of results. To investigate and understand the results, we have to reproduce the same scenarios and find the same confidence intervals every time we run the same stochastic experiment. When debugging parallel stochastic applications, we need to reproduce the same control flow and the same result to correct an anomalous behaviour. Reproducibility is also necessary for variance reduction techniques, for sensitivity analysis and many other statistical techniques [Kleijnen, 1986; L'Ecuyer, 2010]. In addition, for rigorous scientific applications, we want to obtain the same results if we run the application in parallel or sequentially. Consequently, software random number generation remains the prevailing method for HPC, and we will see that specialists are warning us to be particularly careful when dealing with parallel stochastic simulations [De Matteis and Pagnutti, 1990a; Pawlikowski and Yau, 1992; Hellekalek, 1998b; Pawlikowski and McNickle, 2001].

1.2 About Random Numbers

1.2.1 Pseudorandom Numbers

Pseudorandom Number Generators (PRNGs) use deterministic algorithms to produce sequences of random numbers and have been studied in pertinent reviews, many from Michael Mascagni and Pierre l'Ecuyer, and also from other authors, including the well-known first edition of Knuth's Art of Computer Programming, volume 2 and its re-editions [Knuth, 1969; James, 1990; L'Ecuyer et al., 2002a; Mascagni and Chi, 2004]. If some applications can cope with 'bad' or poor randomness according to current standards, we cannot afford producing biased results for simulations in nuclear physics or medicine for instance [Li and Mascagni, 2003; Maigne et al., 2004; Lazaro et al., 2005; El Bitar et al., 2006; Reuillon, 2008a].

For parallel stochastic simulations, there is an even more critical need for a large number of parallel and independent random sequences [L'Ecuyer and Leydold, 2005], each with good statistical properties (approximating as close as possible a truly random sequence). We cannot give a mathematical proof of independence between two random sequences, so **each time this word is used, it should be considered as (pseudo) independence according to the best accepted practices in mathematics and statistics**.

In mathematics, the notion of 'pseudo'-random refers to polynomial-time unpredictability properties in the area of cryptology. However, in this thesis, we do not consider RNGs for cryptographic usage but only for scientific simulations. We do not consider quasi-random numbers either. They can improve the convergence speed (and accuracy) of Monte Carlo integration and also of some Markovian analysis [Niederreiter, 1992]. However, these numbers are not independent and even if they are interesting they can only be used for some specific applications.

1.2.2 Why do we settle for Pseudorandom Numbers when True Random Numbers are available?

Before considering the use of True Random (TR) sequences in stochastic simulations, we need to recall the basic principles of such applications. Stochastic simulations are developed for their results to be analysed and reproducible. To do so, we need to master every parameter of the model. Considering stochastic models, the random sequence

feeding the simulation is also a parameter that we need to be able to reuse and provide to other scientists. In doing so, they can check or rerun our experiments and check the results. As long as True Random sequences are not issued by a deterministic algorithm but by quantum phenomena, they are not reproducible if they are not stored.

This supposes that True Random sequences in Stochastic Simulations would have to be prerecorded in files before being used. This is perfectly suitable for small scale simulations where memory mapping techniques will help to reduce the impact on performances (see [Hill, 2003] for more details about unrolling of random sequences).

However, this approach has the major drawback that the files containing the TR numbers will have to be located on the same machine as the simulations programs they feed. In order to explore a Design of Experiments (DOE), or to run several replications of a High Performance Simulation (MRIP: Multiple Replications in Parallel), we will often want to distribute these independent executions in order to speed-up the whole experiment time. This raises the problem of the random data migration. For more than 7 years, our research team has been faced with simulations in nuclear physics and medicine that needed up to hundreds of billions of random numbers (for a single replicate). This represents tenths of gigabytes. As a matter of fact, moving such amounts of data to distributed environments, such as computing grids for instance, might significantly impact the overall execution time of the simulation.

Now, we can think of other approaches. Being not reproducible is not the sole constraint of True Random sequences. Most of the time, their numbers are generated with the help of either internal or external devices plugged into a computer. These devices may vary from a small USB key to recent GPU boards. Consequently, if we want to take advantage of True Random features in a distributed environment, we must ensure that every machine that will run our binaries is equipped with such devices. Unfortunately, this is hardly possible when we consider serious distributed systems where resources are shared by a large community across a wide area. For example, as users of the European Grid Initiative (EGI), we are not aware of any True Random equipment available in EGI's sites (among more than 337,000 cores available at the time of writing).

Lastly, we could rely on a central element to provide random numbers in a distributed environment. This would imply having one or several servers, dedicated to True Random numbers generation across a network. Moreover, we would also need a user-friendly API (Application Programming Interface) allowing one to easily get random numbers from these servers. The web site located at `http://random.irb.hr/` offers such a service [Stevanovic et al., 2008], but only for very low scale applications.

Registered users benefit from an easy to use library to get True Random numbers directly from a remote server. Basically, centralized approaches are well-known for the bottleneck problem they introduce. Apart from this previously evoked problem, which makes a central server hardly implementable in a High Performance Computing context, the 'irb' library displays other weaknesses.

First, it is based upon the network availability of the server, which might not be a 100% reliable way to draw random numbers. Moreover, what appears as an attempt to prevent bottlenecks becomes a hindrance: there are quotas limiting the daily numbers output to 100 MB per user, with respect of a monthly barrier of 1GB per user. This does not meet at all HPC expectancies.

Finally, we have described three ways to employ TR numbers within stochastic simulations:

- Take advantage of a device plugged in the machine running the simulation;
- Record True Random sequences in files;
- Query a remote server for the numbers.

These approaches have all displayed major flaws, and are clearly not satisfying solutions in an HPC environment executing a large amount of simulations. As a conclusion, we still have to champion Pseudorandom Numbers Generators, whose statistical quality is good enough for any scientific application nowadays. Pseudorandom Numbers Generators are a definitely more portable solution when tackling distributed applications. The portability of PRNGs usually results from the few hundred lines of code necessary to implement them.

Although True-Random numbers would bring a perfectly unpredictable randomness, only sensitive cryptographic applications require this characteristic. On the other hand, scientific applications might even take advantage of the deterministic behaviour of PRNGs. At the time of writing, True Random numbers sequences are not a relevant solution neither for HPC nor for stochastic simulation because of hardware considerations.

However this might evolve in the future in view of an innovative Intel chip, called Bull Mountain, which takes advantage of True Random number generation facilities. This chip is already integrated in the cutting-edge Intel's Ivy Bridge architecture, but at the moment the TRNG is so slow that it is dedicated to seed a hardware-implemented PRNG.

1.3 Pseudorandom Number Generation in parallel

When dealing with stochastic parallel simulation, we have to consider two main aspects: the quality of the PRNG and the technique used to distribute pseudorandom streams across parallel Processing Elements (PE). If stochastic sequential simulations always require a statistically sound PRNG, in the case of parallel simulations, this requirement is even more crucial.

The parallelization technique should ideally generate pseudorandom numbers in parallel, that is, each PE should autonomously obtain either its own pseudorandom sequence or its own subsequence of a global sequence (partitioning of a main sequence). If such independence is not guaranteed, the parallelism is affected. Then, the sequences assigned to each PE must not depend on the number of processors, or the simulation results could not be reproduced.

Designers of parallel stochastic simulations always have to reply to this fundamental question: how can we make a safe RNG repartition to keep, on the one hand, efficiency, and on the other hand, a sound statistical quality of the simulation to obtain credible results? Indeed, the validation of such parallel simulations is a critical issue. Paul Coddington precisely states: 'Random number generators, particularly for parallel computers, should not be trusted' [Coddington and Ko, 1998]. Much research has been undertaken to design good sequential RNGs. Whatever the parallelization technique is, we have to rely on a 'good' sequential generator according to a set of main principles proposed by specialists [Lagarias, 1993; Coddington and Ko, 1998; Hellekalek, 1998a]. Among the best generators currently available, we can cite Mersenne Twister (MT hereafter) [Matsumoto and Nishimura, 1998] introduced in 1997. SFMT [Saito and Matsumoto, 2008] is currently less known. It is an SIMD-oriented version of the original Mersenne Twister generator with the following improvements: speed (twice as fast as MT), a better equidistribution and a quicker recovery from bad initialization (zero-excess in the initial state). The periods being supported by SFMT are incredibly large, ranging from 2^{607} to 2^{216091}. The WELL generators (Well Equidistributed Long-period Linear), on the basis of similar principles (Generalized Feedback Shift Register), have been produced by Panneton in collaboration with L'Ecuyer and Matsumoto [Panneton, 2004]. L'Ecuyer suggests that multiple recurrence generators (MRGs) with much smaller periods (above 2^{100} but less than 2^{200}), such as MRG32k3a [L'Ecuyer and Buist, 2005], can also have very interesting statistical properties, and are easier to parallelize according to our current knowledge. At the end of the 1980s, as the interest for distributed simulation increased, numerous research works took on to

design parallel RNGs [De Matteis and Pagnutti, 1988; Durst, 1989; Percus and Kalos, 1989; Eddy, 1990; De Matteis and Pagnutti, 1995; Entacher et al., 1999; Srinivasan et al., 2003].

Assessing the quality of random streams remains a hard problem, and many widely used partition techniques have been shown to be inadequate for some specific applications. For instance, several studies in networking and telecommunication question the relevance of many stochastic simulations because of the use of poor quality RNGs [Pawlikowski and Yau, 1992; Pawlikowski, 2003b; Hechenleitner, 2004], as well as the mediocre parallelization techniques used in some software [Entacher and Hechenleitner, 2003]. Chapter 2.3 will give a survey of those parallelization techniques.

1.4 Pseudorandom streams on GPU

Recent developments try to shrink computation time by relying more and more on Graphics Processing Units (GPUs) to speed-up stochastic simulations. Such devices bring new parallelization possibilities thanks to their high computing throughput compared to CPUs, as explained by Figure 1.1. They also introduce new programming difficulties. Since the introduction of Tesla boards, Nvidia, ATI and other manufacturers of GPUs have changed the way we use our high computing performance resources. Since 2010, we have seen that the top supercomputers are now often hybrid.

Given that RNGs are at the base of any stochastic simulation, they also need to be ported to GPU. In this thesis we intend to present the good practices when dealing with pseudorandom streams on parallel platforms, and especially on GPUs. Two main questions arise from these considerations:

- How are these random streams produced on GPU?
- How can we ensure that they are independent?

In our case, we focus on the parallelization techniques of pseudorandom streams used to directly feed parallel simulation programs running on GPU (called kernels in the CUDA language). Before stating what we will study in this thesis, let us point out first, that we will not propose any new PRNG, and second, that our study is not tied to the parallelization of random number generation algorithms, albeit we do survey some parallel PRNG algorithms. Still, Chapter 3.3 proposes guidelines that will hopefully help developers to use reliable parallelization techniques of random streams to

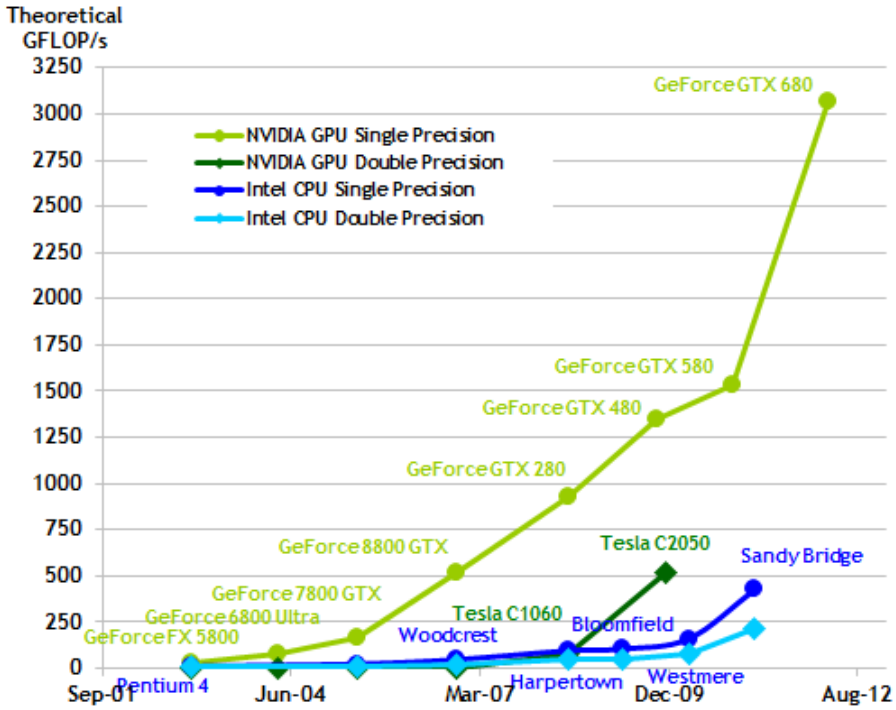


Figure 1.1: Floating-Point Operations per Second for the CPU and GPU (from [NVIDIA, 2011])

ensure their independence, and that are, in addition, well adapted to GPU architectures particularities.

Some works have attempted to speed-up generation using the GPU before retrieving random numbers back onto the host. However, current CPU-running PRNGs display fairly good performances thanks to dedicated compiler optimizations. For instance, Mersenne Twister for Graphics Processors (MTGP) [Saito and Matsumoto, 2013], the recent GPU implementation of the well-known Mersenne Twister [Matsumoto and Nishimura, 1998], is announced as being 6 times faster than the CPU reference SFMT (SIMD-oriented Fast Mersenne Twister) [Saito and Matsumoto, 2008], which is already very efficient in terms of performance.

Meanwhile, previous studies have shown that the time spent generating pseudorandom numbers could represent at most 30% of CPU time for some "stochastic-intensive" nuclear simulations [Maigne et al., 2004], but they are very scarce. For less intensive simulations, where less than a billion numbers are needed, there is no real need for parallelization and unrolling is still the most efficient technique [Hill, 2003]. Considering the small part of the execution time used by most stochastic simulations to generate random numbers, it is not worth limiting GPUs usage at the generation task. To har-

ness the full potential of GPUs, we are more interested in providing random numbers to GPU-running applications that will consume them directly on the device.

1.5 Details about GPUs: programming and architecture

In this section we will briefly introduce some background about NVIDIA GPUs on which this thesis mainly relies. We will describe in turn the architecture, the programming model and languages available to program such devices. Although this section focuses on NVIDIA GPUs only, most of the notions introduced hereafter can apply to other brands of devices.

1.5.1 Architecture

1.5.1.1 Architecture of GPUs (before 2010): the example of the Tesla C1060 (T10) board

The Tesla C1060 card with the NVIDIA T10 processor was the first GPU dedicated to scientific computing. It does not even have a graphical output!

GPUs were originally designed to perform relatively simple, but also very repetitive, operations for a large number of data (each pixel of an image for instance). They dedicate a wide area of their chip to processing units which can perform the same operation in parallel [Nickolls and Dally, 2010]. For example, while the majority of recent microprocessors have less than 10 cores, NVIDIA announced its Tesla T10 as owning 240 "CUDA cores". However, the cores present in GPU and CPU cores are very different and it is not simple to take advantage of the enormous power of GPU.

The core architecture of GPUs from Tesla T10 generation consists of 30 Streaming Multiprocessors (SMs) (3 in each of the 10 TPC - Thread Processor Clusters). Each of these streaming processors consists of several components:

- for calculation:
  - 8 thread processors (SP Thread Processor or SP) for floating-point computations;

  - 2 Special Function Units (SFU) performing complex mathematical calculations such as cosine, sine or square root;
  - 1 Double Precision unit (DP) for double-precision floating-point computations;
- for memory management:
  - Shared Memory;
  - Cache.

The 240 “CUDA cores” announced by NVIDIA correspond to 240 (30x8) thread processors on the board. However, this is more a marketing denomination than an actual description. The element of a GPU that matches most of the characteristics of a CPU core are SMs. They have their own memory, cache and particularly scheduler. Thus, two distinct Streaming Multiprocessors can perform different instructions at the same time, whereas the thread processors belonging to the same SM cannot. Thread processors behave according to the SIMD (Single Instruction Multiple Data) paradigm, executing a single statement on 8 different data sets only.

Moreover, unlike microprocessors, few efforts have been made over legacy GPUs to speed up memory accesses, making them particularly disadvantageous for an application.

1.5.1.2 GPU Architecture (2010-2012): the inputs of Fermi

The second architecture of NVIDIA GPUs, codenamed Fermi, corrects the main limitations mentioned above and improves the raw performance. The declination of this GPU architecture dedicated to scientific computing is the Tesla C2075. In terms of computing power, the C2075 has only 14 streaming multiprocessors, compared to the 30 SMs of the previous C1060. They are composed of two groups - scheduled separately – of 32 SIMD thread processors (448 "CUDA cores" versus 240 previously). In addition, double precision floating point calculations on these thread processors support the IEEE754-2008 standard. The double precision calculations have been significantly improved: each SM now owns 1 double-precision unit for 2 single-precision units, whereas the previous ratio was 1 double-precision unit for 8 single-precision units [Wittenbrink et al., 2011].

The major changes brought by this new architecture is the appearance of L1 and L2 caches. The 64KB L1 cache is also used to implement the shared memory area of the SM. It is partly configurable as it allows allocating the largest part (48KB) to

either L1 cache or shared memory. The appearance of these caches usually helps to increase the performance of an application for free. With regard to reliability, Fermi introduces a major input with ECC (Error Correcting Code) RAM available with the scientific range of NVIDIA GPUs. This is mandatory for HPC applications on the road to exascale computing, where lots of physical parameters (alpha particles, ...) can occasionally modify data in memory.

Finally, the architecture, coupled with a version of the CUDA SDK from 4.0 and later, improves compatibility with the C++ standard. It proposes the main features of C++ on the device: dynamic allocation, inheritance, polymorphism, templates, ...

1.5.1.3 GPU Architecture (> 2012): Kepler

The new architecture proposed by NVIDIA in late 2012 was named Kepler [NVIDIA, 2012]. The Kepler architecture continues to improve the performance of GPU computing while focusing on a more economical power consumption. The number of CUDA cores continues to increase, rising from 240 CUDA cores for the C1060 to 448 cores for the C2050, and now 2,496 cores for the K20, the scientific version of the Kepler generation. However, let us recall that these CUDA cores are nothing more than vector units and should not be compared to traditional CPU cores.

The two major features of this architecture are called Dynamic Parallelism and Hyper-Q. Dynamic Parallelism can break the master-slave relationship between the CPU and the GPU. Traditionally, the CPU controls the execution and delegates massively parallel computing to the GPU. Dynamic Parallelism allows the GPU to gain autonomy in being able to launch new parallel computations without CPU intervention.

The other novelty, Hyper-Q, focuses on a problem tied to the popularization of GPUs. Many developments including GPU cannot take advantage of 100% of their capacity. To compensate this under-utilization, Hyper-Q allows multiple CPUs to address the same GPU. Previously, only one physical connection was possible between CPU and GPU. Now, up to 32 connections can be created. Several applications could also benefit from this new opportunity. We think primarily of applications using MPI to distribute the workload across multiple CPU cores, which can now collaborate with each GPU installed in the host machine. Dynamic Parallelism and Hyper-Q are only enabled on Kepler devices which host runs a CUDA 5.0 SDK.

1.5.1.4 Summary

The main advantage of GPUs is obviously the potential computing capacity that this technology offers at low cost. At the time of writing, parallel applications running on GPUs are still strongly impacted by the poor performance of some memory accesses within the same GPU and memory transfers between the CPU and the GPU. Even if recent architectures have greatly improved things by adding caches, it is always necessary to perform the largest number of instructions on the smallest amount of data by computing element, in order to obtain the announced performances.

1.5.2 Programming model of NVIDIA GPUs

NVIDIA call their parallel programming model SIMT (Single Instruction, Multiple Threads) [NVIDIA, 2011]. It is actually an abstraction layer of SIMD (Single Instruction, Multiple Data) parallel architectures described in Flynn's taxonomy [Flynn, 1966]. The idea behind SIMT is to ease the use of a SIMD architecture to developers that are more used to create threads than to fill in vectors. The following lines will detail the base concepts of the SIMT programming model and its interaction with the hardware.

1.5.2.1 Grids, Blocks and Threads

To take advantage of the many thread-processors present on the GPU, it is not necessary to manually create as many threads as desired and then assign each thread to a processor. Simply, the number of threads is specified when the function intended to run on the GPUs is called (named *kernel* in CUDA terminology). This number is provided by filling the block size (the number of threads included in a block) and the grid size (number of blocks included in the grid). To allow a large number of threads to run the same kernel, the threads must be grouped into blocks, which are grouped in a grid. The distribution between the grid and the blocks is left free to developers. These figures can significantly impact the performance of the parallel application. The two concepts of *grid* and *blocks* will be detailed later, it is sufficient for now to understand that a block contains a given number of threads and the grid contains a given number of blocks.

Before coming to the very important concept of blocks, it is necessary to make a short digression on the concept of *warp* introduced by NVIDIA. A warp is a software

concept that refers to a group of 32 threads that will execute the same instruction at the same time on a Streaming Multiprocessor (SM). The fact that it is a software concept means that there is no material equivalent to warps. The concept of warp has an important impact on the programming of GPUs because of their role in memory access and in thread scheduling. Actually, the memory accesses within a warp can be merged. Thus, if all the threads in a warp access contiguous memory locations, only two requests will be required. In addition, versions of pre-Fermi GPUs cannot schedule more than 48 warps at the same time on a single SM. A warp itself containing 32 threads, it is impossible to exceed 1,536 threads run simultaneously.

Blocks are used to group the threads in a first logical group. Unlike warps, block can be controlled and configured by the developer. The maximum number of threads that can be contained in a block depends on the GPU board used (up to 1,024 threads per block on current GPU at the time of writing). The execution of all the threads belonging to the same block will be performed on a the same SM (conversely, threads present on different blocks can be executed on different SMs). Now being run on the same SM is not insignificant and will allow threads of the same block to take advantage of the shared memory area of the SM (to communicate but also to pool the memory accesses). It will also be possible to synchronize all the threads of a block at a specific location of a kernel.

The sole utilization of blocks does not maximize the power of GPUs. They can run multiple blocks at the same time and thus increase the number of calculations in parallel. It is then necessary to have a large number of threads, a number that can easily happen to exceed the maximum of 1,024 threads in a block. In this case, it is necessary to work with multiple blocks in the *grid*. The grid represents the top bundle of threads launched for a given kernel. Historically, a single grid was associated with a kernel, but recent versions of CUDA (5.0 and later) now allow to launch new kernels, and thus create new grids, from a kernel. Figure 1.2 sums up the multi-scale hierarchy of threads at the heart of the SIMT programming model.

1.5.2.2 Scheduling on the GPU

Most of the considerations in this section are from our personal experience, intersected with a smattering of documentation provided by NVIDIA on the subject. Although the CUDA language is well documented, the proprietary architecture that executes it often appears as a black box. This is particularly the case for *GigaThread*, the scheduler discussed in this section, whose behaviour may seem quite erratic at first.

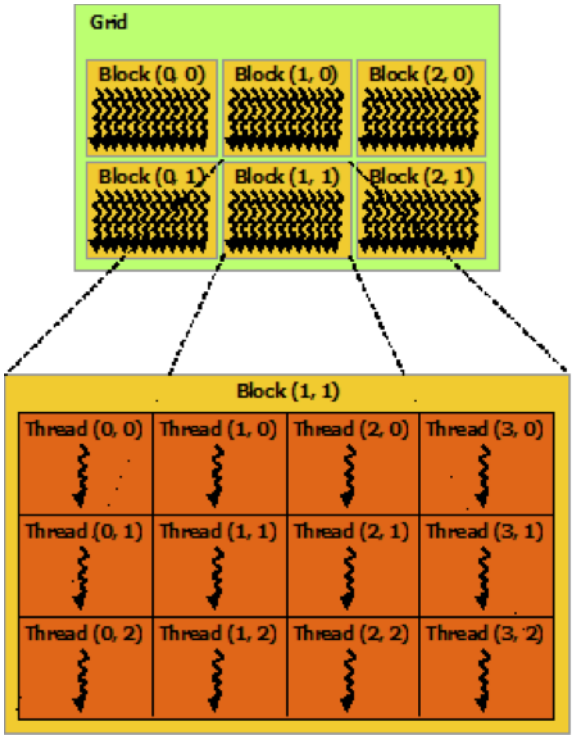


Figure 1.2: Example of a grid containing 6 blocks of 12 threads (4x3) (from [NVIDIA, 2011])

A Streaming Multiprocessor (SM) of the Fermi architecture is composed of 32 thread-processors. Now, the SM works on warps with 32 threads. A SM is able to perform an operation for 32 threads of a warp at each clock tick. At the same time, warps are scheduled by two warp schedulers on each SM. Each warp scheduler, however, can only assign threads to half of the thread-processors, 16 threads in a single warp. Thus, it takes 2 clock cycles to fully execute an instruction for a warp.

Moreover, thread-processors perform operations on data in memory that it is obviously necessary to load first. These memory accesses required to execute an instruction by an entire warp are made by warp (32 memory accesses at a time) on recent architectures, unlike the half-warp access of the previous architectures. Of course, the instructions can not be executed until data has been loaded and the performance decreases if no warp is ready to run. One key to achieve satisfactory performance on GPU is handling a large number of warps to hide latency. It is customary that the Fermi architecture requires a minimum of 18 per SM warps to hide latency for example. Thus, warp scheduling is necessary in order to hide at most the latency of memory accesses.

Another scheduling level takes place with blocks. A GPU owns several SMs, and it is possible to run several blocks simultaneously on different SMs. Scheduling is required when the number of blocks is too large to allow their simultaneous executions. In this case, the blocks are executed in several runs. The GPU can schedule up to 8 different blocks on a single SM, provided the limit of the maximum number of threads that can be scheduled on a SM is not reached at the same time. Blocks scheduling allows to some extent to reduce the impact of memory access by running other threads when some are awaiting data from the memory.

Since each SM has its own unit to read and decode instructions, each block can be managed independently. Thus, a block which all threads have completely finished their execution can be immediately replaced by a new block without having to wait for all the blocks already assigned to the SM to complete. This mechanism explains why, in some cases, the execution time will slightly change when the number of blocks used is increased: as block execution can still be performed simultaneously, the total execution time does not increase in proportion to the number of threads used (although it generally grows, as increasing the number of threads usually leads to a rise in the size of the data to process, and thus of the number of memory accesses).

1.5.3 Programming languages

1.5.3.1 CUDA

The NVIDIA's CUDA technology[1] is the toolkit designed to exploit highly parallel computation using NVIDIA GPUs. Its SDK was first released by NVIDIA in February 2007. The main purpose of CUDA is to let people use GPUs as platforms for computations and not only for games.

CUDA not only refers to a particular range of GPU architectures, but it also designates the tools and the language used to program the GPU. NVIDIA offers a lot of tools to code, debug and profile their applications, to developers willing to use their devices. The CUDA language provides an extension to the C++ language to allow interface with the GPU. All the constructs available in C++ are available in CUDA, but extra keywords and new functions are provided to communicate with the GPU, and parallelize the application.

Even if NVIDIA releases a C++ SDK, wrappers exist that enable other languages

[1]CUDA Zone: `http://www.nvidia.com/object/CUDA_home_new.html`, *last access 8/12/13*

like Python[2] [Klöckner et al., 2012], Fortran[3] or Java[4] to leverage CUDA-enabled devices.

The drawback of this set of technologies is that it is proprietary and fully designed to work with NVIDIA's devices. For example, it is not possible to run a CUDA application either on an AMD or an Intel hardware (be it processing unit or graphical unit).

Technically, CUDA relies on a NVIDIA preprocessor: NVCC, which will transform CUDA language extensions into standard C/C++ later processed by a classical compilation toolchain. This technology has encountered such an interest from its launch that it was introduced in supercomputers leading the World "Top 500" ranking as early as in 2010.

1.5.3.2 OpenCL

OpenCL is a standard proposed by the Khronos group that aims to unify developments on various kinds of hardware accelerators architectures like CPUs, GPUs and FPGAs. It provides programming constructs based upon C99 to write the actual parallel code (called the kernel). Kernels are executed by several work-items, that will be mapped to different execution units depending on the target: for instance, GPUs will associate them to local threads. For scheduling purposes, work-items are then bundled into work-groups each containing an equivalent amount of work-items.

The basic constructs are enhanced by APIs (Application Programming Interface) used to control the device and the execution. At the time of writing, the latest version of the API is 1.2 [Khronos, 2011] that has been released in November, 2011. OpenCL programs execution relies on specific drivers issued by the manufacturer of the hardware they run on. The point is OpenCL kernels are not compiled with the rest of the application, but on the fly at runtime. This allows specific tuning of the binary for the current platform.

OpenCL brings three major inputs to HPC developments: as a cross-platform standard, it allows developing applications once and for all for any supported architecture. It also provides an abstraction layer that lets developers concentrate on the parallelization of their algorithm, and leave the device specific mechanics to the driver.

[2]PyCUDA: `http://mathema.tician.de/software/pyCUDA`, *last access 8/12/13*
[3]CUDA Fortran: `http://www.pgroup.com/resources/cudafortran.htm`, *last access 8/12/13*
[4]Java bindings for CUDA: `http://www.jCUDA.de/`, *last access 8/12/13*

The major drawback of OpenCL is its complicated API. Concretely this lies in three major problems that will be detailed hereafter: bloated source code, double calls and risky constructs. First, when kernel functions that execute on the hardware accelerator remain concise and thus fully expressive, host API routines result in verbose source code where it is difficult for an external reader or for a non-regular developer, to determine the purpose of the application among all those lines.

Second, some design choices of the API make it even more verbose when trying to write portable code that can run on several hosts without any change. For instance, OpenCL developers are used to calling most of the API query functions twice. A first call is often required to determine the number of results to expect, and a second actually gets the right amount of elements. For instance, a function such as *clGetPlatformIDs* will return a list of the available OpenCL platforms in an array passed as a parameter along with its size. This array must have been allocated upstream and its size is consequently the maximum amount of results it can store. In addition to filling the array with the available platforms, *clGetPlatformIDs* also returns the total number of platforms in the system. Consequently, an application needs to invoke *clGetPlatformIDs* twice in order to figure out dynamically the amount of platforms on a given system: first, the function is called to determine the number of platforms and allocate an array according to this result, second, the function is summoned to actually fill the array. Such a design is widely used in the OpenCL API, and developers often have to call the same function twice in a row to make their code portable. The whole process results in bloated source files, whose real behaviour might become difficult to comprehend. We will designate these two subsequent invocations of the same routine as the double-call pattern hereafter.

Finally, such verbose constructs discourage developers to check the results of each of their calls to the API. At the same time, the OpenCL API is very permissive with regards to the type of the parameters its functions accept. Now imagine that two parameters have been awkwardly swapped in an API call, it is very likely that the application keeps quiet about this error if the developer has not explicitly checked the error code returned by this call. In an heterogeneous environment such as OpenCL, where the parallel part of the computation will be relocated on hardware accelerators, runtime errors that only issues error codes are not the easiest bugs to get rid of.

All these drawbacks make it clear that although the OpenCL standard is a great tool offering portability to high performance computing applications, it does not meet the expectations awaited from high level APIs. Parallel developers need such APIs to help them avoid common mistakes, and to produce efficient applications in a reasonable

lapse of time. This is partly why, we chose the CUDA technology at the beginning of this thesis for our GPU-enabled developments. In the same time, CUDA displayed far better performance than OpenCL back in 2010 [Karimi et al., 2010]. Although we disregard raw OpenCL source code written from scratch, we have later considered OpenCL programs resulting from an automatic code-generation process.

1.6 On the need for software engineering tools

1.6.1 Software Engineering

The study of good programming techniques to tend to the best possible software has become a discipline in its own right: the Software Engineering. The aim is to establish more or less stringent rules, which when applied intelligently, can improve the quality of the created software [McConnell, 2004; Sommerville, 2010].

1.6.2 Object-Oriented Modelling

In an object-oriented program, a major part of the work consists in designing classes. In this thesis, we propose software tools for HPC that rely on good software engineering principles. A number of important points must be considered when designing classes of an object-oriented application. The overall objective will have an important influence on the design. Here we introduce the major concerns that have guided the design of the tools presented in Chapter 4.

A class is more understandable and therefore much easier to change if it is organized in a coherent set of attributes and methods. Particular emphasis is placed on the design of the public interface of a class. The idea of this approach is to keep the most homogeneous interface possible: expose the same level of abstraction for all its elements. Homogeneity will be encountered with NVIDIA's cuRand PRNG library, described in Chapter 2.5.2.

Beyond having a homogenous interface, it is useful to keep classes simple. Several points can be addressed in this way. Here, we focus on the issues for which statistics have shown for many years a decrease of the number of errors generated, and therefore in terms of design time [Basili et al., 1996].

The interface of a class wins first to be a limited number of methods. During their

work on the subject, [Basili et al., 1996] had several groups of people develop a software solution to solve the same problem. Solutions displaying a larger number of methods in their classes have also been those in which the number of errors in the design of the solution was the greatest. In his book “Code Complete” [McConnell, 2004], Steve McConnell proposes the number of seven methods (plus or minus two) per class. This figure is derived from the psychologist George Miller’s paper [Miller, 1956] in which he explains that it corresponds to the number of different elements a person can use when performing a task without having to make additional efforts to remember to use each. Obviously, this number can vary because of the background and training of people. Still, this it is the proper basis when comes the time to think about the number of methods that a class must offer, or the maximum number of parameters expected by these methods, or the maximum inheritance depth of a given class, etc ...

Another important source of error in the design of an object-oriented program is tied to inheritance. It is a very powerful concept to model certain problems, but like any concept, an erroneous use often leads to a shabby design. Let’s first consider the following rule: public inheritance corresponds to the relationship of generalization / specialization that is represented by an ontological relationship “is a”. This rule is important enough to be stated by Meyers in one of the sections of his “Effective C++” book [Meyers, 2005] as “the most important rule in the design of object oriented programming in C++”. In 1987, Liskov introduced the substitution principle that bears her name and which defines that any instance S subtype of T, may replace objects of type T in a program without altering any of the desirable properties of that program. When the Liskov Substitution Principle is transgressed, it certainly is a design error. The concept that lies behind is then rather “has a” or “is in terms of implementation” instead of "is a". In such situations, Meyers advocates to use composition or aggregation in another section of his book.

Beyond the problem of design inheritance as “is a”, other problems may occur. Bertrand Meyer refers to the Open/Closed principle as: “software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification” [Meyer, 1988]. However, the use of inheritance has the disadvantage of increasing the complexity of a class: the knowledge of the implementation of the subclass is no longer sufficient for a developer who would like to know the full interface of the class or the way all the methods are implemented. Much of this information will belong to the super class. And this is a major problem because it will have a big impact on the encapsulation of the class, which can be broken. Then, it is often necessary to know the implementation of one or more methods of the mother class functionality to ensure that the child class is implemented correctly. We will study the case of a broken

Open/Closed principle in the ThreadLocalRandom class from the early versions of the Java Development Kit 7 (JDK 7) in Chapter 4.3.

1.6.3 Template MetaProgramming

The Template MetaProgramming (TMP) mechanism takes advantage of C++ templates to execute a portion of the instructions at compile time instead of the runtime. By evaluating a template expression, it is thus possible to calculate constants, and eliminate these calculations during the execution of the operations or to perform actions on types, such as type checking for instance [Touraille et al., 2010]. If this technique can greatly improve the execution time of applications, it can also greatly complicate the source code. In addition, performing calculations at compile time, can increase the total time of compilation very significantly, and even in unacceptable proportions in some cases. In this thesis, we will make use of TMP in Chapter 4.2 to check constraints on the classes at compile time.

These software engineering concerns had previously been evoked in Jonathan Caux's PhD thesis [Caux, 2012]. The interested reader can find a more thorough survey of software engineering and optimization techniques in this thesis.

This chapter has stated the context in which the work presented in this thesis evolves. Our goal is to introduce modern software engineering practices to serve parallel stochastic simulation, and especially those running on GPU. Prior to any proposition, Chapter 2 will survey the techniques available to handle pseudorandom streams in parallel environments. We will then discuss of their potential application to GPU platforms.

Chapter 2

Pseudorandom Number Generation in Parallel Environments and the Particular Case of GPUs

---

> “ *[...] j'ai cru devoir creuser jusqu'à la racine [...]*
>
> — Jean-Jacques Rousseau, *Discours sur l'origine et les fondements de l'inégalité entre les peuples*

Contents

2.1 Introduction

Sequential PRNGs have been studied for a long time [L'Ecuyer, 2010], and finding a good quality PRNG to use in a sequential application has not been a problem for more than a decade. Many works have been accomplished to characterize the statistical quality of PRNGs, leading to several testing libraries. Nowadays, reference testing suites are well known and are used to assess PRNGs. However, a PRNG should always be considered in relationship with the scope of the application it feeds. For instance, cryptographic applications developers ought to base their choice on the NIST testing battery [Rukhin et al., 2001], whereas simulationists should use TestU01 [L'Ecuyer and Simard, 2007]. These two testing batteries can be considered as a standard for their respective domains at the time of writing.

According to [Coddington, 1996; Hellekalek, 1998b], 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 for instance)? How will we ensure the independence between parallel streams in order to prevent the simulations involved 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 [Coddington, 1996; Hellekalek, 1998a; Traoré and Hill, 2001; Reuillon et al., 2011; Hill et al., 2013]. These techniques try to ensure the maximum independence between random streams using different strategies. We will consider in this work whether or not, and how, these techniques can 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 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), almost 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 [Khronos, 2010], but the latter is still not robust enough in our opinion. Its current performances are slower than what you could obtain with CUDA [Karimi et al., 2010]. However, we feel that this standard deserves our interest, and should take on an important part of our future work.

Hereafter, we will name as Processing Elements (PEs) those effectively computing data in parallel. In a CUDA GPU environment, threads will be regarded as 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 kind of parallel architecture. 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 provide each thread with an independent stochastic stream, in order to prevent potential biases that could be introduced otherwise.

In this chapter, we will survey the techniques ruling the generation of pseudorandom numbers in parallel environments. We will focus on the particular case of GPU architectures. First, we will expose the tools stating the statistical quality of pseudorandom streams (Section 2.2). Then, we will present the various techniques that can be used in order to correctly distribute an original pseudorandom stream across parallel PEs (Section 2.3). Section 2.4 addresses the particular case of GPUs when it comes to distributing a pseudorandom stream across their parallel PEs: the threads. Eventually, we will expose software libraries handling the distribution of pseudorandom streams at the heart of stochastic simulation, running on both CPU and GPU (Section 2.5).

2.2 Statistical and Empirical Testing Software for Random Streams

Knuth, in [Knuth, 1969], proposed a set of statistical tests for random streams. Marsaglia designed a testing suite, named *DieHard* [Marsaglia et al., 1990], highly regarded for many years. The statistical test suite developed by the National Institute for Standards and Technology is also interesting, particularly when cryptographic qualities are

required. As mentioned in the previous section, *SPRNG* is also providing a set of statistical tests. A detailed description of the main statistical tests for pseudo-random numbers was given in Rütti's thesis [Rütti et al., 2004], and he also proposed a testing suite with some colleagues [Rütti, 2004]. The *DieHarder* testing suite is proposed and updated by Brown *et al.* [Brown et al., 2009]. Although DieHarder is also an interesting testing suite, we highly recommend the *TestU01* software library, which currently offers the most complete collection of utilities for the statistical testing of uniform random number generators [L'Ecuyer and Simard, 2007].

In addition to the classical statistical tests for RNGs and the other tests previously cited and proposed in the literature, *TestU01* proposes new original tests as well as predefined tests suites (Crush and BigCrush with more than a hundred tests). Test of bit sequences are included, and *TestU01* also considers the size of the sample according to the period length and to the kind of test considered. The *TestU01* software also proposes interleaving of random streams; this is particularly interesting in the context of parallel simulation, to test the influence of parallel substreams inter-correlations. Empirical techniques to test parallel random number generators have been proposed by Coddington and Ko [Coddington and Ko, 1998]. In 2008, Romain Reuillon considered testing 65,536 independent sequences created by DC [Matsumoto and Nishimura, 2000] with *TestU01* (Crush). This was achieved using the former EGEE (known as EGI) computing grid [Reuillon et al., 2011].

Besides testing software designed for sequential generators, Coddington proposed a set of criteria for 'parallel generators' [Coddington and Ko, 1998]:

- The generator should be able to work on any number of processors;
- The sequential sequence in use for each processor should satisfy the current statistical tests;
- The parallel sequences should be reproducible;
- Parallel sequences or streams should be uncorrelated.

To this set of criteria, we add the fact that each PE should possess its own sequence or subsequence to ensure the reproducibility of execution on each PE independently of the parallel programming model (PEs can be: threads, processes or processors). For instance, if we detect a bug during the execution on a specific PE, we have to be able to reproduce it with the same sequence and on any platform. The need for mutual independence between the parallel sequences or streams has to be checked carefully to avoid long-range correlations [De Matteis and Pagnutti, 1988, 1995, 1990b]. The problem of long-range correlations has been identified in various simulation applications

for many years, and techniques have been proposed to avoid them as far as possible because to our knowledge, we cannot have rigorous mathematical proofs for this problem [De Matteis and Pagnutti, 1988; De Matteis et al., 1992; Eichenauer-Herrmann and Grothe, 1989; Entacher et al., 1998]. De Matteis and Pagnutti have proposed interesting approaches to control correlations in [De Matteis and Pagnutti, 1990a].

2.3 Design of Parallel and Distributed Random Streams

There are many approaches to obtain parallel random numbers streams either by partitioning the main sequence (stream) of a given generator into subsequences (substreams) or by parameterizing one generator to have several sequences (multiple streams). Some techniques have been surveyed in [Traoré and Hill, 2001; Bauke and Mertens, 2007] and more recently, we have proposed the following survey [Hill et al., 2013]. This section tries to give a thorough overview of the distribution techniques found in the literature. Chapter 3.6 will use the elements introduced hereafter to propose a taxonomy of distribution techniques, depending on the type of original random streams they target. We also try to identify the scope of each of these techniques, so that any practitioner can refer to this section to figure out which technique best fits his application.

2.3.1 Partitioning a unique original stream

Any PRNG whose state vector is n-bits wide generates, from an initial state, a periodic random number sequence whose period is inevitably less than 2^n. Random numbers distribution techniques introduced in this section share a common principle referred to as cycle division in [Mascagni, 1998]. It splits random numbers from the considered cycle between the different PEs. Techniques detailed hereafter differ from each other only by the method they use to split the main stream. In view of the literature, we consider four main partitioning techniques.

2.3.1.1 Central server technique

This first approach is not truly parallel. It consists in using a central server, running an RNG and providing on-demand pseudo-random numbers to different PEs. This approach displays three major drawbacks. First, a simulation using this technique will not be reproducible because of scheduling policies that might change the order in which

numbers will be provided to PEs. This is very problematic when trying to debug a stochastic simulation because the same numbers will not necessarily be assigned to the same PE at each run. Moreover, the results of a simulation cannot be checked by other scientists for the same reason: there are few chances that the same random sequence will be issued twice. Second, the central server approach will create a bottleneck if too many PEs are considered. Third, reproducibility cannot be ensured when the number of available PEs changes from one simulation run to another. As a consequence, this is suitable for serious games with limited parallelism but not for scientific applications.

2.3.1.2 Sequence splitting

The sequence splitting method is also known as 'blocking' or 'regular spacing'. It consists in allocating non-overlapping, contiguous and equally sized blocks from the original random stream to form substreams. When partitioning a sequence $x_i, i = 0, 1, ...$ into N streams, the j^{th} stream is $x_{k+(j-1)m}, k = 0, ..., m-1$, where m is the length of each stream; m must be chosen so that each stream is long enough to achieve the stochastic simulation performed by the corresponding process. For instance in [Hechenleitner and Entacher, 2002], Hechenleitner showed that in the OMNeT++ network simulation package, if the spacing between sequences was set to 1 million draws, it led to biased results (due to inter-sequence correlations) for processes using more random numbers. However, the determination of a good value for m is not the only difficulty. If overlapping can easily be avoided, long-range correlations in the initial RNG can lead to small-range correlations between the potential substreams [De Matteis and Pagnutti, 1988, 1990b]. The impact of this kind of correlation is problematic as shown in [Srinivasan, 1998; Reuillon, 2008a]. Figure 2.1 represents an original stream chunked through sequence splitting for 2 and 3 PEs. The schema also insists on the fact that an original stream is split in equally sized parts which length may vary from one application to another.

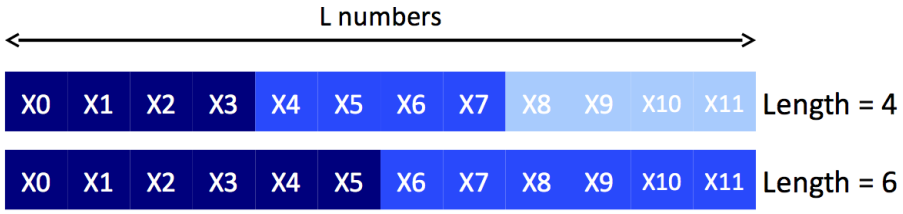


Figure 2.1: Sequence splitting in a unique original stream considering 2 processing elements (PEs), then 3 PEs

2.3.1.3 Random spacing

The random spacing or indexed sequences method builds a partition of n streams by initializing the same generator with n random statuses. In the case of old LCGs, it was named random seeding (because the status was limited to a unique number called a seed). For modern generators with a more complex status, the random statuses are generated with another RNG, and this technique is interesting when generators have a huge period. This technique is easy to set up. Wu and Huang in [Wu and Huang, 2006] showed that the minimum distance between n statuses generated in this way is on average $1/n^2$ multiplied by the period length. The risk is of course to have a bad initialization linked to the fact that two random statuses could be too close to each other, implying an overlapping of corresponding sequences. For a PRNG with a period P, the probability that n sequences of length L, generated by a random spacing technique will overlap is equal to $1-(1-nL/(P-1))^{n-1}$, which is equivalent to $n(n-1)L/P$ when nL/P is in the neighbourhood of 0. A situation implying 3 PEs is sketched in Figure 2.2 where we can notice the non-equal gaps between two random numbers ranges.

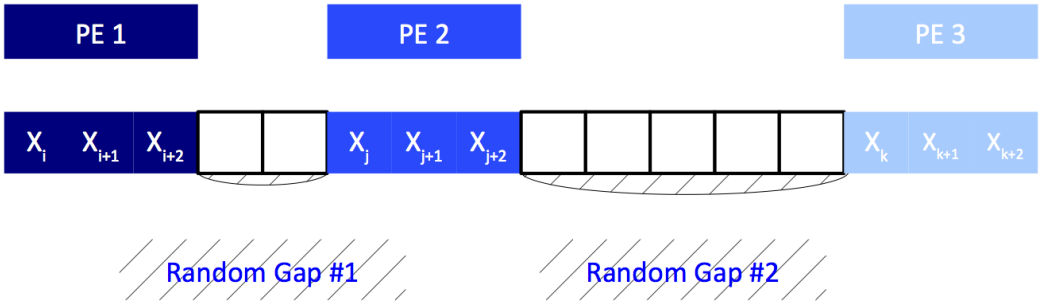


Figure 2.2: Random spacing applied to 3 processing elements (PEs)

Overlapping risks become sizeable with short-period PRNGs. All the PRNGs tested by Pierre L'Ecuyer and Richard Simard in [L'Ecuyer and Leydold, 2005] 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 thousands of billions of random numbers [Li and Mascagni, 2003; Maigne et al., 2004; El Bitar et al., 2006; Schweitzer et al., 2013], ($L = 10^{12}$), a hundred of such replications ($n = 100$) makes $n(n-1)L$ on the order of 10^{16} (i.e., more than 2^{53}). The probability to see an overlapping between two subsequences issued from a PRNG of period P far bigger than 2^{53} becomes negligible. Nonetheless, from 20 LCGs PRNGs tested in [L'Ecuyer and Simard, 2007], 14 laid out an overlapping probability greater than 99.9% (period P such that $log_2(P) < 50.4$). Widely spread PRNGs, such as *Comblec88* of period

$P = 2^{61}$ (combined LCGs), are on the acceptance borderline for this technique (with $n = 100$ and $L = 10^{12}$, the overlapping probability is equal to 0.43%). They are shipped with several renowned software packages though, including RANLIB, CERNLIB, Boost, Octave and Scilab [L'Ecuyer and Simard, 2007]. Our advice would be however to avoid such LCGs or combined LCGs for modern simulations because of their structural weaknesses [L'Ecuyer et al., 2002a].

2.3.1.4 Leap frog

The leap frog (LF) is the way to partition a random stream in the manner of dealing cards to several players. Random numbers are allocated in turn to PEs, as cards are dealt to players. Pragmatically, let each processor hold an i identifier. Every such PE will build a Y_i substream from an X original random stream such as $Y_i = X_i, X_{i+N}, ..., X_{i+kN}$, with N equal to the number of processors [Aluru, 1997].

Given the period P of the global sequence, the period of each stream is P/N. As with the splitting technique, the long-range correlations in the initial RNG can lead to small-range correlations between the potential substreams, particularly if we have a large number of PEs. In addition, Wu and Huang [Wu and Huang, 2006] showed that depending on the interval used (i.e., the number of PEs and the length of the random sequences), poor spectral values could be observed. A case where the quality of the original RNG is seriously affected by the LF technique is shown in [Hellekalek, 1998b]. In addition, when this technique is used without jump ahead (Section 2.3.4), performances are divided by the number of PEs because of the bottleneck problem appearing in the central server technique.

This technique needs to be used with caution depending on the environment where it is set up. Thus, to preserve reproducibility between two executions of the same simulation, one should not use this technique when the parallelism grain is not fixed yet. As a matter of fact, different random streams will be assigned to PEs depending on the chosen grain. This situation is illustrated in Figure 2.3.

For the classical LF approach, one must ensure that the original status can be shared between all the PEs when the underlying PRNG algorithm relies on a linear recurrence to produce the next numbers of the sequence. This limits the target architectures of the classic LF technique to shared memory architectures, where a PRNG state can be stored and accessed efficiently by the PEs. For distributed memory architectures, a smart implementation of the LF technique can be used when the number of PEs is a power of 2. In this case, using a unique Lagged Fibonacci generator, each PE can

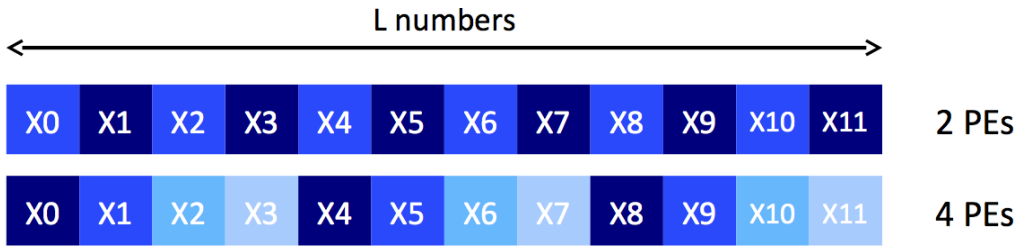


Figure 2.3: Leap frog in a unique original stream considering 2 processing elements (PEs), then 4 PEs

implement its own version of the generator that performs the right jump corresponding to the identifier of the PE [Janowczyk et al., 2008]. This approach is also discussed in Section 2.5.2, focusing on GPUs.

2.3.2 Partitioning multiple streams: Parameterization

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.

Although no mathematical proof can establish this independence, some implementations of parameterization are safe according to the current state of the art [Mascagni, 1998]. We especially think of the Dynamic Creator (DC) algorithm [Matsumoto and Nishimura, 2000] coming along with most of the generators from the Mersenne Twister family. DC integrates a unique identifier, which belongs to the Parameterized Status of the PRNG. This identifier becomes a part of the characteristic polynomial of the matrix that defines the recurrence of the PRNG. 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. An example of parameterization can be found in Figure 2.4, which states three independent parameterized PRNGs. In the case of LCGs and multiplicative congruential generators, this can rapidly lead to poor results [De Matteis and Pagnutti, 1990b; Wu and Huang, 2006] even when the parameters are very carefully checked. For instance Mascagni and Chi proposed that the modulus be Mersenne or Sophie Germain prime numbers [Mascagni and Chi, 2004] to improve the case of LCGs parallelization.

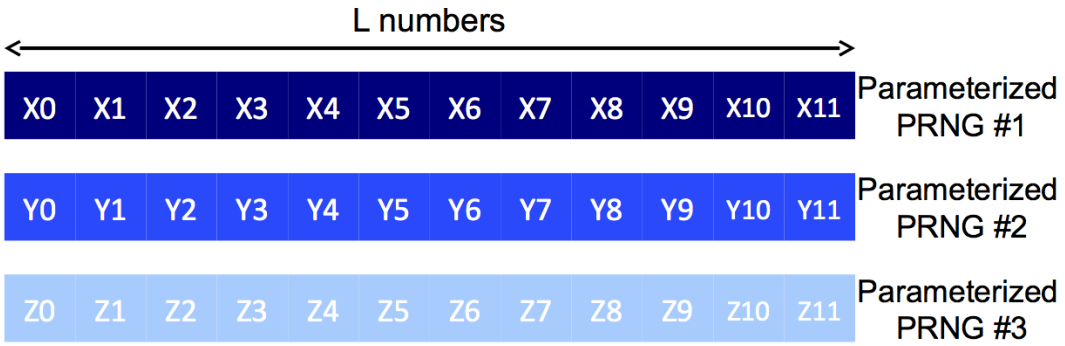


Figure 2.4: Example of Parameterization for 3 processing elements

2.3.3 Other techniques

Now that the main partitioning methods have been presented, we can indulge in a bit of history of other approaches because the parallelization of pseudo-random numbers has been under study for at least 30 years.

Variants have been developed on the basis of classical methods such as the shuffling LF. Its principle is to parameterize both the seed and recursive functions of LCGs. One of the main contributions to this variant is that it results in a scalable period, hence, the number of different random numbers that can be used increases with the number of parallel streams. A parameterization method was used in [Percus and Kalos, 1989] to obtain parallel streams from a LCG, by choosing for the j^{th} stream a multiplier $a(j)$ and an additive constant $c(j)$. It is shown in this case that good results can be obtained if $c(j)$ is the j^{th} prime number less than $\sqrt{(m/2)}$ where m is the modulus. A specific way to generalize the partitioning methods was proposed in the 1980s [Frederickson et al., 1984]. It results in what has been called a 'Lehmer tree'. But the suggestion is apparently limited to LCGs, which we strongly discourage to use for modern scientific applications.

We can also think of hybrid approaches, but they are not distribution techniques in their own right. Yet they combine several standard distribution techniques to take advantage of their respective features. Please note that the resulting random stream of this combined approach needs to pass through the same test batteries than its parents to be validated. Indeed, combining several distribution techniques might not preserve the statistical quality of the original random stream.

In the end, let us add that Boolean Cellular Automata have been considered to generate parallel pseudo-random numbers by Sipper and Tomassini [Sipper and Tomassini, 1996; Tomassini, 1999; Tomassini et al., 1999], who tested the generated sequences for

distributed environments. This technique is rather slow, and [Ackermann et al., 2001] has considered its hardware implementation in programmable chips. We are not aware of any evaluation of this technique by any state-of-the-art testing battery; thus, we cannot advise the use of this technique.

2.3.4 A tool for partitioning: jump-ahead algorithms

A jump-ahead algorithm is a tool used by distribution techniques such as sequence splitting and LF to deal numbers efficiently. This technique enables the analytical computing of the generator state in advance after a huge number of cycles (generations) and corresponds to a jump ahead in the random stream. Knowing the recurrence formula $s_{n+1} = f(s_n)$, where s_{n+1} is the generator status, the difficulty is to determine $f^{(n)} = f \circ f \circ ... \circ f$ ($n-1$ compositions of f by itself) because $s_n = f^{(n)}(s_0)$.

Such a technique is interesting with generators that have very large periods, but it can be slow. Among widespread generators, MT and WELL generators, on the basis of linear recurrences modulo 2 (F_2-linear generators), proposed an algorithm providing jump-ahead facilities. For such long-period generators, Matsumoto and L'Ecuyer teams joined [Haramoto et al., 2008] to develop a viable jump-ahead algorithm running in few milliseconds on current processors. In this case, f is a linear application in F_2^w, with a matrix A with w lines and w columns, where w represents the size of the state vector. The problem is to compute A^n, and this can be achieved, thanks to the characteristic polynomial of the A matrix. When an F_2 linear generator proposes a full period (2^w-1) and when the size of the w state vector is big, the computation of the characteristic polynomial will take time [L'Ecuyer et al., 2002a].

More efficient algorithms exist, for a generator such as MRG32k3a from [L'Ecuyer, 1999] for example. However, this generator is slower than MT [Saito and Matsumoto, 2008]. Also, MRG32k3a has a much shorter period length than MT, although it can be sufficient for most modern applications [L'Ecuyer, 2010].

2.3.5 Joint use of the partitioning of a single stream and parameterization

The two main techniques of random streams distribution, presented in Sections 2.3.1 and 2.3.2, are not exclusive.

To simplify, we can say that any number x_n of a sequence provided by a conventional

PRNG is the result of a call $x_n = g(s_n)$, where s_n is the internal state of the PRNG and g a function usually very simple. The statistical quality of these generators only lies in the complexity of the state transition function f that allows us to deduce the following state: $s_{n+1} = f(s_n)$. However, the downside of this complexity is that it induces a sequential dependence between the successive states of the PRNG. The direct and rapid access to a state s_n (required by the Sequence Splitting or LF techniques, for example) is possible only if one has a jump-ahead function (Section 2.3.4).

With counter-based PRNGs [Salmon et al., 2011], so named because $f(s_n) = (s_n + 1)\, mod\, 2^p$ (where p is the size of the state vector), this sequential dependence disappears because the numbers are generated by calls such as $x_n = g(n)$ where n is the state reduced to a simple counter. The parallelism inherent in the latter relationship allows immediate use of single stream partitioning techniques (Section 2.3.1).

The statistical quality of counter-based PRNGs are therefore due to the complexity of the g functions used (for PRNGs used in simulation, these functions are simplifications of cryptographic block ciphers, such as *AES* or *Threefish*, used in cryptographically secure PRNGs). These functions are indexed by keys, allowing a natural distribution of pseudorandom numbers by parameterization over the key space (Section 2.3.2).

Finally, the interest of counter-based PRNGs is that the generation of numbers $x_n = g_k(n)$ can be parallelized either by partitioning each stream according to the values of the counter n (Section 2.3.1) or by following a parameterization approach (Section 2.3.2) on the basis of the k keys. When the size of counter space and key space is sufficient (up to 2^{128} and 2^{64}, respectively, for counter-based PRNGs presented in [Salmon et al., 2011]), being able to use both techniques presented previously independently even allows, within each PE, to associate a PRNG to each software entity. For instance, in the case of an individual based stochastic simulation with a very large number of individuals, the sequence splitting technique can be applied in the counter space to distribute the pseudorandom sequences on different replications of the simulation, while setting on the keys to assign each individual its own PRNG. This idea will be more thoroughly evoked in the Perspectives of this thesis (Chapter 6.5).

2.4 Pseudorandom Numbers on GPUs

2.4.1 The dark age

Until recently, designing a PRNG for GPU-enabled platforms could be very complicated as it forced programmers to deal with graphics Application Programming Interfaces (APIs). Some implementations are presented in [Sussman et al., 2006]. The authors especially list the limitations of these GPU dedicated PRNGs due to the past weaknesses of the hardware. Limited output per thread or untruthful operations were part of the restrictions that made these PRNGs feeble for High Performance Computing (HPC) applications. Consequently, a common way to deal with random numbers on GPU was to generate them on CPU before transferring them on the graphics processor. This solution has to face the well-known bottleneck of data transfer between the CPU host and the GPU device. Even with PCI Express 16X running at 8GB/s, this remains a challenge for high performance applications.

2.4.2 GPUs as hardware accelerators of PseudoRandom Number Generation

Let us now focus on PRNGs implementations using GPUs as an hardware accelerator only. This techniques aims at providing random numbers faster to host applications. Actually, such implementations can mostly be found during the genesis of GPU programming, when the underlying architecture and programming languages features narrowed the application scope.

Since 2008 and the advances from NVIDIA, new GPU software and hardware architectures offer the precision and speed needed by many HPC applications. Now, PRNGs can be directly implemented into GPU applications. Recent works propose this new kind of generators. Langdon presented a minimal implementation of the standard Park Miller PRNG [Park and Miller, 1988] on a NVIDIA 8800 GTX GPU in its paper from 2008 [Langdon, 2008]. He announces a speed up of more than 40 compared to his Intel 2.40 GHz CPU. One year later, [Langdon, 2009] increased the speed of his application by four by using the new NVIDIA technology: CUDA (Compute Unified Device Architecture) [NVIDIA, 2010b] with a Tesla T10 GPU. Nevertheless, we do not advise the use of this old generator which has many known flaws, though it was still in use until recently in some well distributed networking simulation software [Entacher and Hechenleitner, 2003].

CUDA has been designed to allow developers to easily harness the computation power of GPUs. In his first implementation, Langdon had to deal with a complex and unadapted graphics API. With CUDA, developers can program GPUs without wasting their time making algorithms and their data fit into graphics dedicated data structures, such as pixels shaders. Furthermore, CUDA does not propose a new programming language but only some C extensions, making it easier to learn for C familiars. The CUDA appellation also refers to the name of the NVIDIA GPU architecture. This generation of graphic boards tries to fulfil the requirements noted in the conclusion of the previously cited [Sussman et al., 2006]. Sussmann called for instance for an implementation of the IEEE 754 floating point numbers standard. Since the 2.0 architecture, codenamed Fermi, CUDA-enabled devices propose configurable L1 cache, ECC memory and a considerable increase of performance in the computation of double precision floating point operations.

Although these highly parallel devices bring much more peak performances than CPUs, they must be carefully programmed to deliver the expected power. In fact, GPU architectures combine a manycore approach with SIMD vector cores. As vector processors do, GPU-enabled algorithms need to repeat the same operation on different data to correctly exploit the device. This is the main reason of the recent PRNGs proposals, especially tuned for GPU architectures. In 2006, [Saito and Matsumoto, 2008] proposed an SIMD version of the famous 'Mersenne Twister' called SIMD-oriented Fast Mersenne Twister (SFMT). Although this algorithm can be used on regular CPUs or on SIMD-enabled CPUs (using either SSE or AltiVec vector instructions), it cannot be directly transposed to a GPU architecture. Most PRNGs have to be rethought from scratch to make the most out of GPUs. In addition, the target application has to be taken into account when choosing the algorithm. In the case of a CUDA implementation, some PRNG/simulation pairs are surveyed in [Howes and Thomas, 2007].

2.4.3 Implementation strategies

Given that CUDA defines software levels that map the device architecture, PRNGs implemented using this technology can be organized at one of the following scopes, corresponding to the main elements of the CUDA framework: a thread, a block of threads or a kernel (the program running on the whole GPU). All these approaches have been studied in the literature. In [Zhmurov et al., 2010], authors present three basic generation algorithms working either with a single instance of the PRNG for the kernel or with an instance per thread. The three algorithms exposed are quite basic: Ran2, Hybrid Taus and a Lagged Fibonacci generator. In the same way, [Langdon,

2009] chooses to generate a number per thread in its GPU version of the Park-Miller algorithm. The last strategy is proposed in [Saito and Matsumoto, 2013] where a new variant of the MT algorithm spreads independent PRNGs through each thread block, thanks to an algorithm known as Dynamic Creator (DC) that we will detail later.

Beyond the nature of the PRNG algorithm, we prefer to focus on the scope chosen for each implementation. Indeed, we formerly insisted on the need to consider the target application and the PRNG as a pair. Obviously, new PRNG algorithms have to take advantage of GPU intrinsic properties such as heterogeneous memories, or thread organization. The former highly impacts the performance of the PRNG. Considering the approach using a generator per thread, an internal state array has to be saved in each thread. CUDA related works like [Kirk and Hwu, 2010] specify that arrays declared for a thread are stored in the local memory, implemented in RAM. Equivalently, with a PRNG for all thread approach, the global memory is solicited to store the state of the PRNG. Each thread draws a number and updates its component of the state in global memory, implemented in RAM too. These two approaches make a heavy use of global memory, which has the advantage to be persistent across kernel launches within the same application. Yet, this RAM area is quite slow, it implies a 400 to 800 clock cycles latency because it is not cached [NVIDIA, 2010b]. So, even if the global memory storage is compulsory to save the PRNG state between two kernel calls, one can use the shared memory, reachable by every thread within a block, to manipulate the data of the PRNG. Indeed, it is implemented on-chip and is consequently as fast as registers. A good example of this choice is the paper introducing Mersenne Twister for Graphics Processors (MTGP) [Saito and Matsumoto, 2013].

In our opinion, a good GPU PRNG should locate its data structure in shared memory. A PRNG per block approach seems to be the most appropriate way to implement a source of randomness. First, because it exploits the fastest memory. Second, for the sake of applications upgradability. Since hardware architectures evolve very quickly, we cannot afford to rethink algorithms every time the memory amount or number of threads available doubles. So, fixing a block of threads scope for a PRNG algorithm is the safest solution to avoid lots of modifications tied to frequent hardware evolutions. This is the reason why we have decided to study in details the MTGP proposition in Section 2.4.5.

2.4.4 PRNGs designed to be used within GPU-enabled applications

At the time of writing, MTGP release in the first quarter of 2010 and published in [Saito and Matsumoto, 2013], is to our knowledge the sole parallel PRNG that has been specifically designed to run on GPU. The algorithm is intrinsically parallel, and targets the first goal of random number generation on GPU: speeding-up numbers output.

Recent GPU architectures, such as Fermi, opened new development perspectives. Having larger and faster memory areas available per thread, and being able to use object-oriented features, applications have become more and more ambitious, so have PRNGs implementations. GPUs are now considered as fully capable platforms, able to run entire applications by themselves. To do so, developers need PRNG implementations for GPUs that allow their applications to directly consume the issued numbers. Such a trend can be observed with the increasing number of available libraries for CUDA. Pseudorandom number generation follows the same tendency, and we have noticed several contributions in the last two years.

Although GPUs bring much more peak performances than CPUs, they must be carefully programmed to deliver the expected power, and most PRNGs have to be rethought from scratch to leverage GPUs characteristics. In fact, GPU architectures combine a manycore approach with SIMD vector cores. As vector processors do, GPU-enabled algorithms need to repeat the same operation on different data to correctly exploit the device. This is the main reason of the recent dedicated PRNGs proposals.

Other quality implementations can be found in [Bradley et al., 2011]. This study does not propose any new PRNG but details how a small set of reference quasi-random and pseudo-random number generators (Sobol, MRG32k3a and Mersenne Twister) have been successfully ported to GPU through CUDA. We draw attention of the reader to the fact that these fine RNGs led to different GPU implementations, depending on their characteristics. For example, the small memory footprint of MRG32k3a allows an instance per thread whereas Mersenne Twister's large data structure conduct to a block of threads implementation level. We will detail the different options offered when implementing a PRNG on GPU in a further subsection, but the work of [Bradley et al., 2011] implicitly distinguishes two RNG groups: those which CPU code can directly be ported to GPU, with very few changes, and those who need to be redesigned to fit with GPU constraints.

Implementing PRNGs in a way to draw numbers directly on GPU led us to think about the best design of such pieces of software, considering both PRNG characteristics and GPU constraints.

2.4.5 Description of MTGP

2.4.5.1 Data Structure

MTGP is described in the recent paper [Saito and Matsumoto, 2013]. In this section, we are presenting its features learned from both the paper and the study of the source code. These properties are recalled hereafter.

First of all, MTGP is obviously inheriting from the properties of its elder, though it is quite different from a simple GPU implementation of the original MT, as seen in [Podlozhnyuk, 2007]. As a matter of fact, the authors use the original paper describing MT [Matsumoto and Nishimura, 1998] to lay out their generator. Since we often champion this family of generators, [Reuillon, 2008b] has studied 2^{16} statuses of the original MT algorithm using the TestU01 Crush test battery from [L'Ecuyer and Simard, 2007]. The involved tests verify the linear complexity of the random sequence. MTGP is based upon the same linear recurrence to create random sequences, so it is not designed for cryptographic purposes either.

We have also noted that MTGP specified a common notion of the generators belonging to the parameterized family. We distinguish this cast of generators by the compound form of their data structures. It contains two distinct elements implied in the generation algorithm, we call them seed status and parameterized status. The first is basically the common seed given by the user to initialize a generator. The second stores parameters determined at a particular PRNG creation and acts as a unique signature of the generator. Both these concepts were already present in MT. MTGP makes them more precise by explicitly using a data structure of the form we described in this paragraph. We propose a simple UML class diagram of this concept in Figure 2.5:

MTGP takes this idea one step further by introducing two kinds of statuses: the references and the fasts. Fast statuses use pre-stored elements to decrease the initialization time, and programming techniques such as inlining to speed up the execution time. However, it results in a memory greedier status.

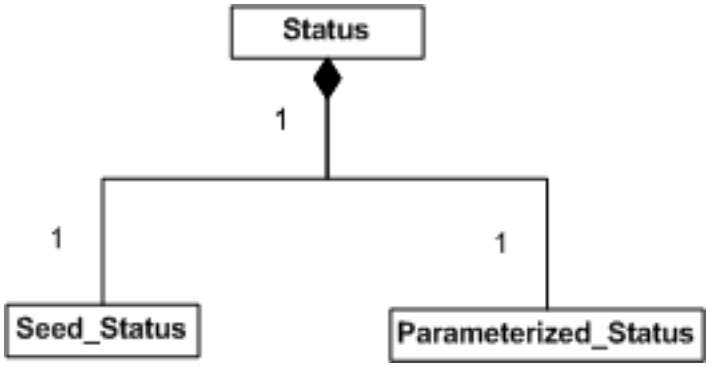


Figure 2.5: UML class diagram of a parameterized PRNG

Parameterized statuses are a common way to ensure independence between parallel stochastic streams. This technique is called parameterization in the literature [Hill et al., 2013]. Depending on the way it is settled, it can lead to poor results [De Matteis and Pagnutti, 1995]. Unfortunately, we do not have any mathematical theorem allowing us to check the independence between two generators, according to their parameters. However, MT came along with the DC algorithm, designed to create large sets of independent generators. This algorithm integrates an identifier, often the one of the processor or thread that will host the PRNG, to distinguish parallel random streams. The said identifier then partly forms the characteristic polynomial of the matrix used by the MT algorithm. Then, if characteristic polynomials are prime to each other, their associated matrices will also be unique in the same context. Hence, we obtain independent parameter sets.

This algorithm has been renewed for MTGP, enabling us to proceed in the same way. Furthermore, it improves the original algorithm by allowing the user to get a larger set of 2^{32} parameters, whereas the original algorithm could only deliver 2^{16} sets. This number is now too small for large computing grids such as the European Grid Initiative (EGI), with more than 337,000 cores at the time of writing this paper. This latter point forces us to experimentally check the independence of the MTGP instances produced by the new DC, in the first released version of the software. Its author explain that a SHA1 (Secure Hash Algorithm) checksum of each characteristic polynomial is generated to let the user verify he did not get duplicated entries.

2.4.5.2 Architecture Independence

One of the most interesting features of MTGP is to be available for both CPUs and GPUs architectures. Even if MTGP has been designed to run on GPUs, a CPU version

is also shipped within the same package. We have based our study on the capability of the generator to merge transparently in CPU-based applications. Hence, we were able to test the PRNG on any CPU-based host with reliable and well-known tools like TestU01. This way, we have avoided the hazardous implementation of a new empirical test battery, which would have to be validated before. Moreover, this property is really precious in our opinion. We intend to use this PRNG for stochastic simulations following the hybrid computing paradigm, where the sequential part of the application runs on a CPU host, while the parallel-one is executed on a GPU board. In such cases, stochastic streams will furnish consistent random numbers to both the CPU and the GPU. With a PRNG like MTGP, we can keep our simulations homogenous, using the same PRNG on each computing element involved in the simulation. Handling independent parallel stochastic streams becomes understandably important when you have to deal with such hardware configurations. We will study in a further section the specific DC coming along with MTGP to maximize this independence. Now considering the new elements we introduced in this subsection, we can extend our previous object model to the particular MTGP. Figure 2.6 depicts its main components.

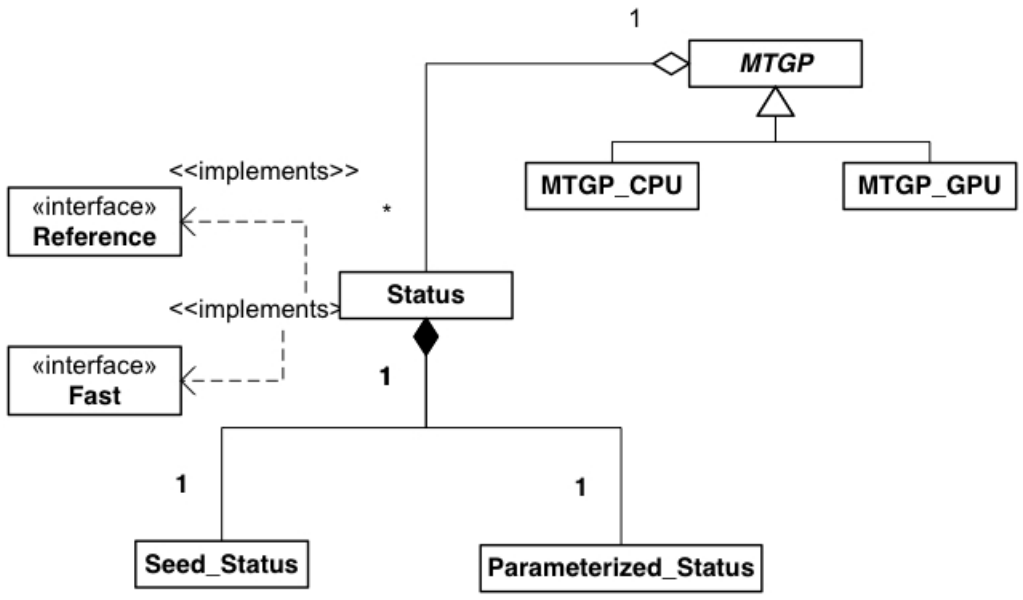


Figure 2.6: Class Diagram for MTGP and its components

We have widely presented MTGP statuses in this subsection. Since this generator utilizes a PRNG per block of threads approach, we will need a different status per block to ensure the independence between the random streams produced. The sample program furnished by the authors takes a number of blocks to use as an argument and owns a set of 128 different statuses to feed these blocks. In Chapter 3.2, we will present

the protocol that helped us to issue a large number of statuses for MTGP users.

2.4.6 Other GPU-compatible PRNGs

2.4.6.1 MRG32k3a

Introduced by Pierre L'Ecuyer in [L'Ecuyer, 1999], MRG32k3a is particularly suited to parallelization among small computational elements such as threads thanks to its intrinsic properties. The lightweight data structure of this PRNG only stores 6 integers to handle its state. The algorithm itself is quite short, and relies on simple operations to issue new random numbers. The parameters chosen for MRG32k3a are such that it has a full period of 2^{191} numbers. This period is fairly enough since L'Ecuyer suggests that periods between 2^{100} and 2^{200} are highly sufficient even for large-scale simulations. MRG32k3a has been designed to produce independent streams and substreams from its original random sequence thanks to its parameters that enable safe Sequence Splitting [Hill et al., 2013]. The internal parameters split the initial sequence into 2^{64} adjacent streams of 2^{127} random numbers, themselves divided into 2^{51} sub-streams containing 2^{76} elements. This situation is represented in Figure 2.7, from the expanded version of [L'Ecuyer et al., 2002a].

Now considering the distribution aspect, we can assign a stream or a sub-stream to each computational element according to the Sequence Splitting technique. As long as we are focusing on parallel applications that are CUDA-enabled, we are dealing with fine-grained Single Instruction, Multiple Threads (SIMT) applications. It means that the computational elements are, in our case, the logical threads of a CUDA kernel and the principle of SIMT is to load the device with as much threads as possible. Still, we do not expect having to deal with more than 2^{64} parallel threads, which is the total number of independent streams bearing 2^{127} random numbers each that MRG32k3a can provide.

2.4.6.2 TinyMT

TinyMT is the latest offspring from the Mersenne Twister family. TinyMT is not described in any scientific article yet, but information about it can be found on its dedicated webpage [Saito, 2011]. This PRNG is described as producing a good quality output, according to TestU01 statistical tests, and displays a long-enough period of 2^{127} numbers. TinyMT leverages parameterization to provide highly independent streams,

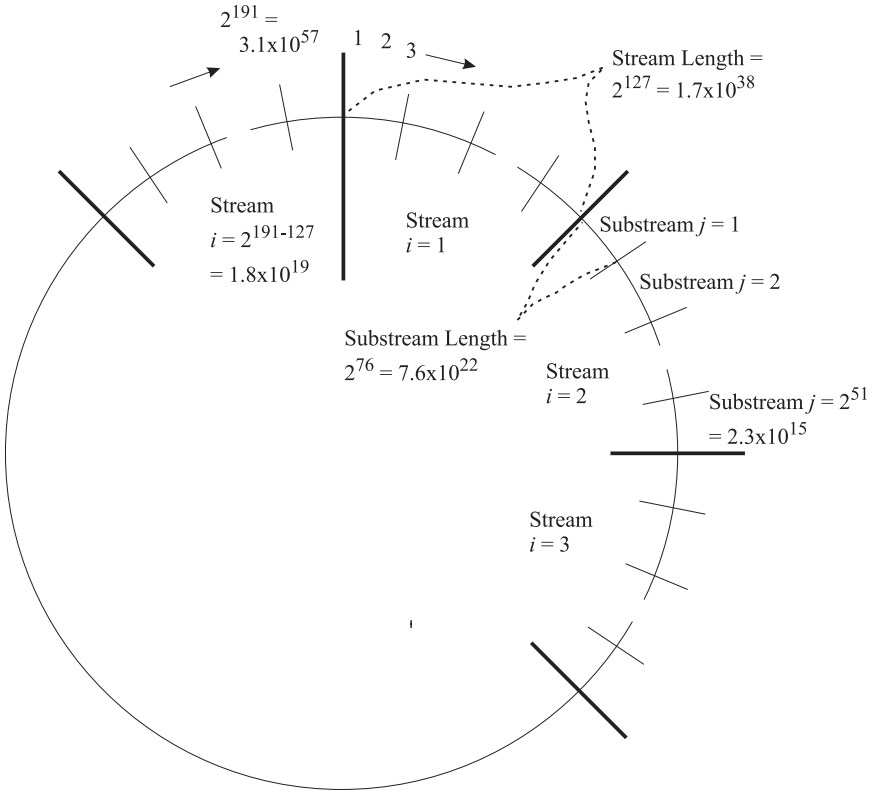


Figure 2.7: Overall arrangement of streams and substreams of MRG32k3a (from the expanded version of [L'Ecuyer et al., 2002a])

each stream being represented by a unique Parameterized Status.

The theoretical aspect of this approach is very satisfying. As usual, the Parameterized Statuses of TinyMT need to be precomputed by a piece of software called Dynamic Creator (DC), which is shipped with the PRNG as an open-source binary. Kenji Rikitake has made available millions of these statuses through his GitHub account[1]. Again, the idea is to initialize each computing element with a different status, since DC can create over $2^{32} \times 2^{16}$ independent statuses. However, memory footprint considerations can lead to hybrid implementations where the same independent Parameterized Status is shared among all the threads of a CUDA block. Independence between random sources is achieved by feeding each thread with a substream of the original stream, following the Sequence Splitting technique. To do so, the original stream is sliced in equal chunks whose starting point, the Seed Status, is assigned to threads depending on their unique identifier. As a consequence, each thread will always consume the same random sequence in different replications of the same execution, thus ensuring

[1] `https://github.com/jj1bdx/tinymtdc-longbatch`, *last access 8/22/13*

reproducibility of the experience.

2.4.6.3 Philox and Threefry

Philox and Threefry are counter-based PRNGs [Salmon et al., 2011] also relying on parameterization to solve random streams partition concerns. Like any other PRNG considered in this study, they are Crush-resistant and display good performance with regards to their low memory footprint and high number throughput. Please note that the GPU implementation of these PRNGs is directly provided by their authors. Both CUDA and OpenCL implementations are proposed for Philox and Threefry. Still, apart from the empirical validation obtained though TestU01, no theoretical guarantee assessing the statistical quality of the output of this generator can be found in the literature, at the time of writing.

2.5 Random Numbers Parallelization Software

2.5.1 Techniques designed for CPUs

In the early 1990s, at the European Simulation Symposium, Pawlikowski and his colleagues proposed an interesting methodology for the parallelization and automatic partitioning of stochastic parallel simulations [Pawlikowski et al., 2002; Pawlikowski, 2003a]. As in [Pawlikowski and Yau, 1992], Li and Mascagni gave advice on how to achieve safe Multiple Replications in Parallel [Li and Mascagni, 2003]. To our knowledge, the first reliable and sound library that takes into account the parallelization of pseudorandom numbers is SPRNG presented in [Mascagni and Srinivasan, 2000]. The authors give an overview of the mathematical grounds of random number generators. It also introduces implementations of various parameterization techniques for different families of generators, which were presented in [Mascagni and Srinivasan, 2004]. This library has been proved reliable for parallel Monte Carlo computations and also proposes a small test suite. An example of multiple replications in parallel is also given in [Ewing et al., 1999] dealing with performance evaluation studies of modern multimedia telecommunication networks. In [Mascagni, 1998], we find a short and interesting discussion around Monte Carlo tools for HPC at the end of the last millennium. As stated previously in the parameterization section, Matsumoto and Nishimura proposed the Dynamic Creator software to generate mutually independent Mersenne Twister generators for parallel computing [Matsumoto and Nishimura, 2000]. This kind of approach

is still a very good parallelization technique for MT generators even if we consider the recent development of efficient jump-ahead techniques for linear recurrences modulo 2 (MT and WELL generators) [Haramoto et al., 2008]. At the beginning of the millennium, L'Ecuyer and his team started to propose a package able to produce many long streams and substreams in C, C++, Java, and FORTRAN [L'Ecuyer and Buist, 2005]; a version for R was proposed in 2005 [L'Ecuyer, 1996]. In 2004, Coddington and Newell released JAPARA [Coddington and Newell, 2004], a Java parallel random number library for HPC for the Java language. This library proposes three good generators (two by L'Ecuyer), parallelized by the sequence splitting or the indexed sequence technique. To our knowledge, the library is limited to shared memory machines. In our opinion, the current best library for Java is SSJ [L'Ecuyer, 2001]. The SSJ package provides a RandomStream Java interface that defines the basic structures to handle multiple streams of uniform random numbers. Each stream of random numbers is a Java object for which the original sequence from a given period can be cut into adjacent streams (or segments) as in the sequence splitting approach. Efficient methods are proposed to move around streams.

2.5.2 The case of GPUs

Dealing with GPUs, we have only found a limited number of techniques and few high quality generators [Bradley et al., 2011]. The purpose here is not to list current PRNG implementations for GPUs. We have presented in [Passerat-Palmbach et al., 2010] a survey of this kind, and in most cases, the purpose was to improve the generation speed, thanks to a GPU accelerator. If this point is interesting, it is not what simulationists are looking for. Our main concern is to be able to massively run independent stochastic functions on GPUs (named 'kernels' in the NVIDIA CUDA terminology).

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. Introduced in version 3.0 of the CUDA toolkit, CURAND [NVIDIA, 2010a] has been designed to generate random numbers easily 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 were originally Sobol for quasirandom numbers and XorShift [Marsaglia, 2003] for pseudorandom. XorShift generators are stated as fast, but they also present statistical flaws as explained by Panneton in his PhD thesis [Panneton et al., 2006]. The API of the library is impacted by changes of RNG algorithms. For instance, choosing MTGP forces the user to call particular initialization functions that are totally

irrelevant for another algorithm.

Thrust::random is part of a GPU-enabled general purpose library called Thrust [Hoberock and Bell, 2010]. 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) [Tausworthe, 1965] and a Subtract With Borrow (SWB) [Marsaglia et al., 1990; Marsaglia and Zaman, 1991]. 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 [L'Ecuyer, 1996].

In Section 4.2, we introduce our own pseudorandom number generation toolkit for GPU named ShoveRand [Passerat-Palmbach et al., 2011b]. It distinguishes from its counterparts by introducing a meta-model that enables the description of every PRNG characteristics. Moreover, this meta-model is implemented exclusively through C++ compile-time template mechanisms, thus introducing no overhead at runtime. ShoveRand offers a common API to users, regardless of the PRNG they select. It also guides developers who would like to integrate a new generator into the framework. The latter performs compile-time analysis on the provided source code to ensure that only PRNG implementations which public interface matches our guidelines (see Section 3.3) will compile successfully.

2.6 Conclusion

We have tackled the use of random number generation for HPC and presented the main partitioning methods for stochastic parallel simulations. The current 'best' generators according to the latest test libraries have been discussed. We have also considered various existing tools because the use of random numbers in parallel stochastic simulations is still a challenging technical problem.

The use of GPUs for intensive independent stochastic hybrid computing can be limited by the current hardware constraints, such as the small size of fast shared memory area, as well as the predominant SIMD paradigm. We have also noticed

the promising TinyMT that is carefully crafted for the small shared memory areas of GPUs. Furthermore, the new hybrid architectures proposed are precisely working on the two aforementioned hardware limitations. In conjunction with the increase of the number of 'cores' in the near future (such as NVIDIA's Maxwell architecture for instance) and the improvement of GPU programming user-friendliness, this could change the way many scientists will consider the use of 'desktop' HPC.

We have seen that more and more applications, and especially stochastic simulations, tend to take advantage of recent GPU architectures in order to improve their performance. However GPU computing needs to offer the same tools as other platforms. High quality PRNGs belong to this category and have existed for more than a decade, although some recent publications dealing with GPU implementations of PRNGs still propose old and weak generators. In this chapter we have shown how difficult it could be to obtain good quality pseudorandom sequences on GPU. Indeed, it implies taking into consideration two different domains: 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. Then, when a PRNG fulfils this requirement, it has to be ported to GPU. It means that a new implementation tuned for GPU platforms must be designed, if it is not already available.

We distinguished several 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 this chapter represent interesting proposal in this way. Still, they do not combine good software practice and sound theoretical basis. The two next chapters will address this concern, as we will introduce in Chapter 4.2 a framework named ShoveRand, which embeds PRNGs following the requirements established in Chapter 3.3 into a GPU-enabled library with a unified API.

Chapter 3

Guidelines Regarding Pseudorandom Number Generation on GPU

> “ *Il n'y a point de hasard.*
>
> — Voltaire, *Zadig, ou la destinée*

Contents

3.1 Choosing Pseudorandom Streams

Finding a fast and reliable Pseudo Random Number Generator (PRNG) to feed a sequential stochastic simulation is not a problem for many application domains since more than a decade. This issue has been tackled in many reference studies for CPU based PRNG [L'Ecuyer, 1990]. [Park and Miller, 1988] raised the fact that one should consider the pair {application, PRNG} instead of limiting the study to the intrinsic qualities of the PRNG. For instance, a very good generator like Mersenne Twister (MT) [Matsumoto and Nishimura, 1998] is not designed for cryptographic applications and the initialization of such generators has to be done carefully. Indeed, for some years, this very nice generator was sensible to its initialization status. Although no generator is universal, the MT family of generators [Saito and Matsumoto, 2008], the WELL generators [Panneton et al., 2006] and some advanced Multiple Recursive pseudo-random numbers Generators (MRGs) from L'Ecuyer [L'Ecuyer et al., 2002a] give very good results when considered for parallel computing in a wide range of applications.

If some criteria have been gathered by Coddington [Coddington, 1996] for sequential and parallel simulation to characterize good PRNGs, it is often safer to also consider the output of empirical testing software. Those have been introduced in Section 2.2.

The main problem we are still facing today is to ensure the correct behaviour of the PRNG when distributed across a tight or large coupled computing architecture. The literature currently provides quite a few references about stochastic streams distribution on classical hardware architectures [Mascagni, 1999; Traoré and Hill, 2001; Bauke and Mertens, 2007; Hill et al., 2013]. The set of references is even poorer when considering GPU platforms. In fact, restricted parallel hardware architectures like the Single Instruction Multiple Data (SIMD) family, which GPUs belong to, do highly impact the implementation of generators. In addition, we still have to select the best way to allocate random substreams to these manycore architectures.

In this chapter, we bring our contribution to these problems. As long as the theoretical propositions introduced in this chapter result from an initial benchmark of Mersenne Twister for Graphics Processors (MTGP) [Saito and Matsumoto, 2013], we

have chosen to describe this study prior to our theoretical proposals (Section 3.2), as a method to analyze a new PRNG algorithm. That being said, we discuss the theoretical aspects as follows:

- What requirements a distribution technique should meet to adapt to GPUs?
- A survey of the distribution techniques actually fitting GPU architectures;
- The constraints bound to the implementation of a PRNG on GPU;
- A taxonomy of the distribution techniques according to several criteria;
- Some guidelines to choose the right distribution technique for a particular context (architecture, application, PRNG algorithm).

In this manner, we first intend to analyze the features of a particular generator designed for GPU hardware architectures: MTGP. The second purpose of this chapter is to give guidelines to both users and developers of GPU-enabled PRNGs.

3.2 MTGP Benchmark

3.2.1 Introduction

We achieved an analysis of MTGP by using the dynamic creation of this PRNG and facing the resulting generators up to the current most stringent *TestU01* battery of tests: BigCrush. Few weaknesses were identified during these experiments when compared with the original MT. The purpose of our test was to obtain a large set of fine Parameterized Statuses (introduced in Section 2.4.5) allowing the initialization of MTGP without introducing any potential bias with a bad status. Only statuses that passed BigCrush were kept in this study [Passerat-Palmbach et al., 2010].

However, problematic statuses were mostly failing the same two tests. We observed that in some cases with relatively small periods (2^{3217}), 30% of the parameters generated by dynamic creation led to MTGPs with failure to tests 35 and 100 of *TestU01*. Thanks to precisions given by Makoto Matsumoto and Mutsuo Saito, we now know that Test 35 discards the most significant 25 bits from the 32-bit words and then uses the next 5 bits. The reason suggested by the authors to explain why MTGP failed these tests was that there were dependencies among these 5 bits. Failure may occur when the least significant bits are not "tempered" (a fixed linear transformation applied to the sequence). Instead of classical MT, MTGP did not take care of these bits when

tempering. Matsumoto and Saito were extremely reactive and have already corrected the GPU version of MTGPDC to take this into account.

Nonetheless, our study has shown that MTGP was particularly safe with longer periods, according to *TestU01* criteria. Yet, the longer the period, the more space it needs to store its internal state vector used by its algorithm. And this point is currently noticeable because even the best current GPU architectures propose a small amount of fast shared memory. We have selected about 6,000 fine MTGP Parameterized Statuses, with the lowest available period 2^{3217} to reduce their GPU shared memory footprint. They have been provided to Matsumoto and Saito, and are also publicly available as a C source file [1] containing an array of Parameterized Statuses that have successfully passed the BigCrush test suite from TestU01. This section details the whole process of our initial study of MTGP.

3.2.2 Empiric Test of 10,000 Statuses

We previously evoked the DC tool, initially proposed by [Matsumoto and Nishimura, 2000], to create independent PRNGs to use in a parallel environment. Using DC, we generated 10,000 independent parameter sets (Parameterized Statuses), each corresponding to a different MTGP. This step can be very computationally intensive when the algorithm has to look for huge periods generators, such as 2^{19937} for the original MT. According to [L'Ecuyer, 2010], periods contained between 2^{100} and 2^{200} should be sufficient for nowadays stochastic applications. Thus, we decided to manipulate generators of the lowest period allowed by the MTGP DC, which is still 2^{3217}. Moreover, the lower the period is, the fewer memory it consumes to store the internal state vector of the PRNG. With this configuration, the 10,000 parameter sets for MTGP were generated in a single day, using a 256-core Linux cluster.

The second phase consisted in applying the BigCrush test battery to each newly created generator, in order to check their quality. First of all, to easily analyze such an amount of results, we modified the TestU01 library output to enable it to produce lighter results output files. In this manner, we have been able to parse results files using script tools like *Sed* or *Awk* to generate statistics. Moreover, since lots of our computations have taken place on the European computing Grid Infrastructure (EGI), we reduced the quantity of data transferred from this slow bandwidth system. The use of the European computing grid was compulsory in our case, in fact [L'Ecuyer and Simard, 2009] forecasts BigCrush to take about 8 hours of CPU time on an average

[1] `http://fc.isima.fr/~passerat/mtgp/mtgp_3217_statuses_testu01-proof.c`

64-bit processor. We could not afford to perform the equivalent of 80,000 CPU-hours on a single cluster to get our results in a decent time.

Our final aim was to provide verified Parameterized Statuses to allow the simulation community to initialize GPU-enabled PRNGs without introducing any bias in stochastic simulations, according to the current knowledge. A basic selection consisted in keeping only statuses that had perfectly passed all tests of the battery. Unfortunately, this approach eliminated approximately 40% of the statuses. Thus, we tried to determine whether other statuses could be kept with a good confidence level. We set up a more formal analysis to answer this question.

3.2.3 Statistics-Based Analysis

Each test of the TestU01 Bigcrush battery [L'Ecuyer and Simard, 2007] is governed by the H_0 hypothesis, that the successive output values of the RNG are i.i.d. U(0, 1), i.e. are independent random variables from the uniform distribution over the interval [0;1]. These tests are defined by a test statistic γ (which is a function of the numbers to be tested). They compute and report a number, called the p-value of the test, which is contained between 0 and 1. Furthermore, if γ has a continuous distribution, the p-value is U(0, 1) under H_0. At this point, let us consider two precisions from L'Ecuyer and Simard:

1. « If the p-value is extremely small (e.g., less than 10^{-10}, then it is clear that the RNG fails the test, whereas if it is not very close to 0 or 1, no problem is detected by this test. » [L'Ecuyer and Simard, 2007];
2. « Moreover, when a generator starts failing a test decisively, the p-value of the test usually converges to 0 or 1 exponentially fast as a function of the sample size when the sample size is increased further. » [L'Ecuyer and Simard, 2009].

According to these quotations, we decided to consider three p-value types, detailed hereafter:

- p-values contained between [0.001;0.999] are reckoned as correct, (these values are proposed by the TestU01 library);
- those included in the range [0;0.001[U]0.999;1] are counting as suspect;
- lastly, we refined the previous range since we needed to take into account extremely small p-values (less than 10^{-10}), called disastrous afterwards.

As mentioned previously, we ran our tests on 10,000 independent statuses, with regards to MTGP DC. The column chart appearing on Figure 3.1 represents the number of suspect p-values noticed for statuses where no disastrous p-values were obtained. For the sake of readability, only the thirty-two first tests are present on Figure 3.1.

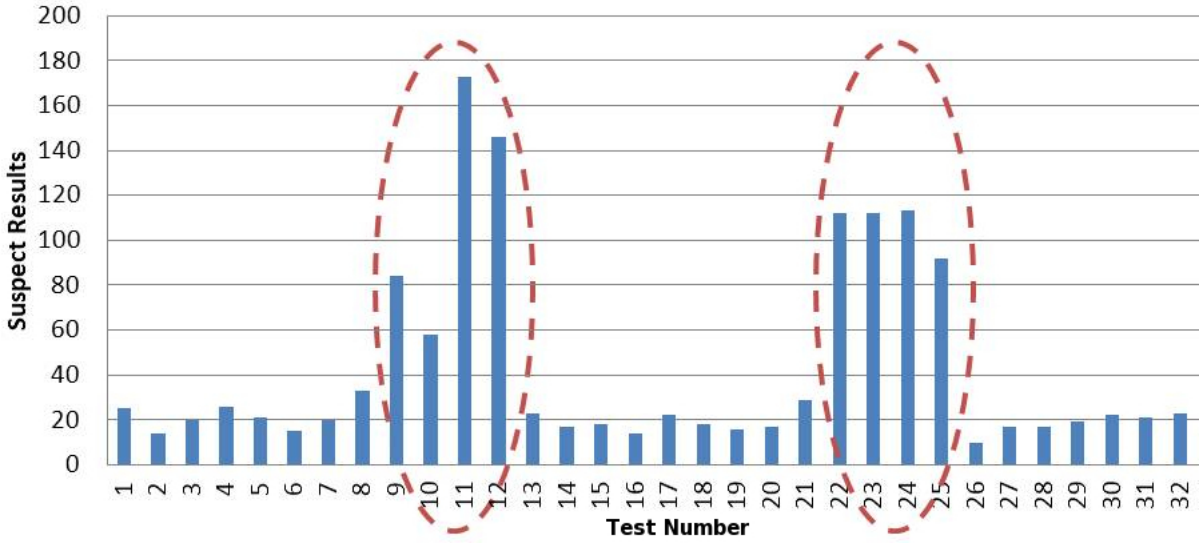


Figure 3.1: Number of suspect results versus test numbers (extract displaying tests 1 through 32)

Figure 3.1 helped us easily identify three test groups. They distinguish from others by recording more than 40 suspect p-values. The interesting point here is that these three groups characterize some aspects of the generator behaviour. In fact, tests belonging to the same group are just differently parameterized versions of the same test. The noticeable tests are described as follows in the reference documentation of TestU01 [L'Ecuyer and Simard, 2009]:

- *smarsa_CollisionOver* (tests 9 to 12), is an overlapping pairs sparse occupancy (OPSO) test introduced in [Marsaglia, 1985];
- *snpair_ClosePairs* (tests 22 to 25), is a m-nearest-pairs (m-NP) test [L'Ecuyer and Simard, 2009];
- *swalk_RandomWalk1* (tests 74 to 79), applies simultaneously several tests based on a random walk of length l over the integers, for several (even) values of l [L'Ecuyer and Simard, 2009].

Under the H_0' hypothesis of a uniform distribution of the p-values over the interval [0;1], the distribution of the number of suspect p-values is binomial with parameters

n = 10,000 and p = 2/1,000. So, we can reject the H_0' hypothesis with a very good confidence level (about 7.10^{-6} for each test that accumulates more than 40 suspect p-values). However, we cannot do the same with H_0. To do so, the γ-statistic used by the test should present a continuous distribution, which is not the case for the considered tests. Concretely, it means that pointing out suspect p-values brings useful information, but no matter the excesses of suspect p-values, such results do not provide a formal statistic proof of the test failure. That is why we consider a test is failed only when it returns disastrous p-values.

Six noticeable tests do not appear in Figure 3.1: four, labelled 70, 71, 80, 81, are linear complexity tests and will be failed by any generator of the MT family (MT, WELL, ...), according to [L'Ecuyer and Simard, 2007]. We increased the execution speed of the test battery by simply disabling those tests that would have systematically produced disastrous p-values. The other two tests, numbered 35 and 100, are the problematic ones. Only these assessments issue disastrous p-values in a non negligible quantity. We noted that about a tenth statuses were failing the 100^{th} Test, while more than a fifth went wrong with the 35^{th} Test. Figure 3.2 gives a graphical representation of the announced proportions.

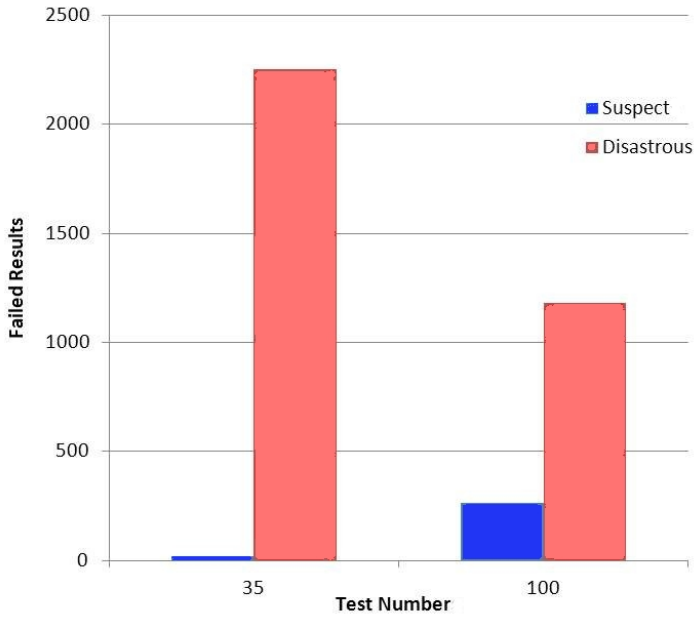


Figure 3.2: Detailed Results for Tests 35 and 100 of the BigCrush battery

Test number 35 is the *sknuth_Gap* test with parameters set to $N = 1$, $n = 3.10^8$, $r = 25$, $Alpha = 0$ and $Beta = 1/32$ [Knuth, 1969]. This test counts, for $s = 0, 1, 2, \ldots$ « the number of times that a sequence of exactly s successive values fall outside the

interval [Alpha, Beta] (this is the number of gaps of length s between visits to [Alpha, Beta]). It then applies a chi-square test to compare the expected and observed number of observations. » [L'Ecuyer and Simard, 2009]. A typical generator miscarrying this test « wanders in and out of [Alpha, Beta] for some time, then goes away from [Alpha, Beta] for a long while, and so on » [L'Ecuyer and Simard, 2007]. However, one should note that the same test is perfectly passed with other Beta values. In view of the analysis we propose, this fact is obviously logical for Beta values lower than the incriminated one (1/32), but it is rather strange not to find disastrous p-values with a higher Beta value (Test 34 sets Beta to 1/16).

Test number 100 is referenced as *sstring_HammingIndep* test with $N = 1$, $n = 107$, $r = 25$, $s = 5$, $L = 1,200$ and $d = 0$. It applies two tests of independence between the Hamming weights of successive blocks of L bits [L'Ecuyer and Simard, 1999]. According to François Panneton, this test measures Hamming-weight dependencies between random values issued by a given generator. It tends to demonstrate that the recurrence does not shuffle bits enough from an iteration to another [Panneton, 2004].

With this first experiment, we have put under the spotlight difficulties that were encountered by the first versions of MTGP 2^{3217}. Assuming that these problems are mostly concentrated on the two properties checked by tests 35 and 100, those producing disastrous p-values, we have focused our further studies on them.

3.2.4 Parameterized Status Influence

3.2.4.1 Seed Status Variation

Let us recall that the only difference between the 10,000 tested statuses were their parameterized parts. They shared the same seed status, arbitrarily filled with 0. Viewing the previous results, we decided to work out whether a successfully passed test was due to either the Parameterized Status, or the Seed status, or both. To do so, we designed a new experiment, sieving a set of 100 Parameterized Statuses alternately associated with 100 randomly chosen Seed Statuses. This initialization technique, called Random Spacing, represented a total of 10,000 combinations put to the proof of the significant tests described in the first experiment. Figure 3.3 shows an extract of the graphical output for Test 35. Red crosses mark a disastrous p-value, while blue ones indicate suspect p-values. An aggregation of red crosses on vertical lines shows that the Parameterized Status on the abscissa failed the considered test for all the seed statuses it was associated to. So, in the case of MTGP, it seems that the sole Parameterized Status

establishes the statistical quality of the output of the generator, independently from the selected Seed Status. This is an important result, which makes any Seed Status initialization safe, as long as the user chooses validated Parameterized Statuses only.

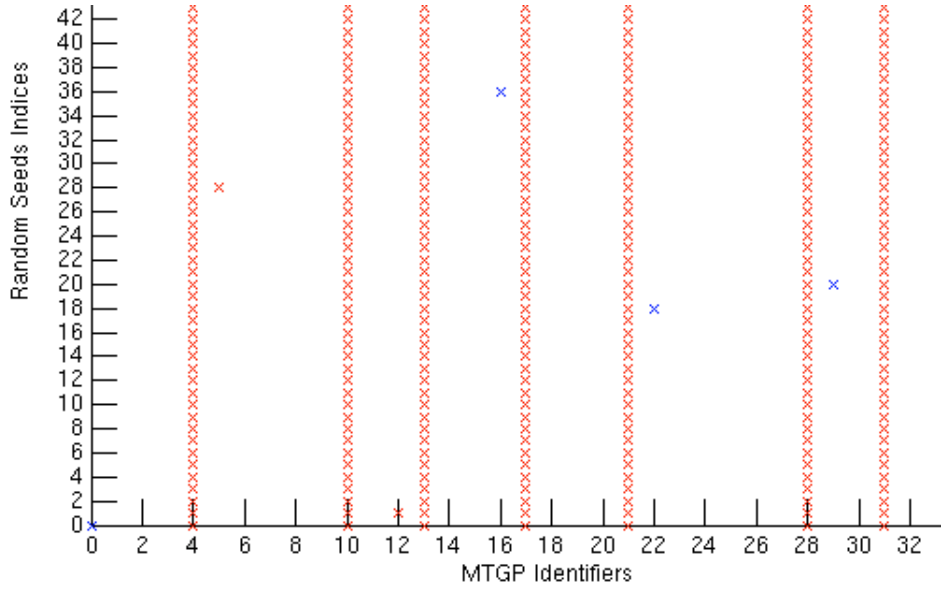


Figure 3.3: Extracts of the results for Test 35: MTGP identifiers versus random seeds indices

3.2.4.2 Period Variation

To comfort our previous assertion, we tried to make other elements of the Parameterized Status vary to observe their impact on the quality of the generator. As long as DC tries to figure out a tempering matrix such as the PRNG produces a well distributed sequence [Matsumoto and Nishimura, 2000], modifying this period should highly impact this property for the newly created statuses. Now, we previously brought forward that Test 35 (*sknuth_gap*) of BigCrush based its judgment on this characteristic. So, we intended to obtain much better results using 1,000 MTGPs of period 2^{23209}, confronted to tests 35 and 100 only. The result is crystal-clear since 99.5% of the statuses passed both tests without any problem. Moreover, we only noticed suspect p-values in the other 0.05%. Obviously, a higher period eliminates sequence distribution issues, but this latter result could hide potential intrinsic weaknesses of the MTGP algorithm. Our last experiment will introduce as a standard a quality-proven PRNG of the same family: the original Mersenne Twister (MT) [Matsumoto and Nishimura, 1998].

3.2.4.3 Algorithm Variation

The original MT is designed with linear-recurrences preventing its recommendation for some particular applications such as cryptography. As far as we know, no other issues are referenced concerning this PRNG. That is why we consider it as a very good standard to compare with MTGP. An interesting point is that both DC algorithms for MT and MTGP are able to produce parameters for the 2^{3217} period. This allows us to work with MTs and MTGPs of the same period. This way, we focus our experiment on the algorithm-dependent parts of the Parameterized Statuses. Once again, we observed the behaviour of each generator when faced to tests 35 and 100. We selected a sample of 1,000 independent PRNGs to compare outputs with our previous benchmarks. Results are even more clear-cut than before: 99.9% of MT-dedicated statuses passed the two tests without any failure, whereas about 64% of MTGP statuses did.

The two previous results tend to show the influence of Parameterized Statuses. Here we have shown that this data structure is tightly bound to the algorithm using it. MTGP seemed to present weaknesses when configured with shorter periods, since MT, running with the same relatively small period, successfully passed tests that MTGP missed. These results are summed up on the column chart displayed in Figure 3.4.

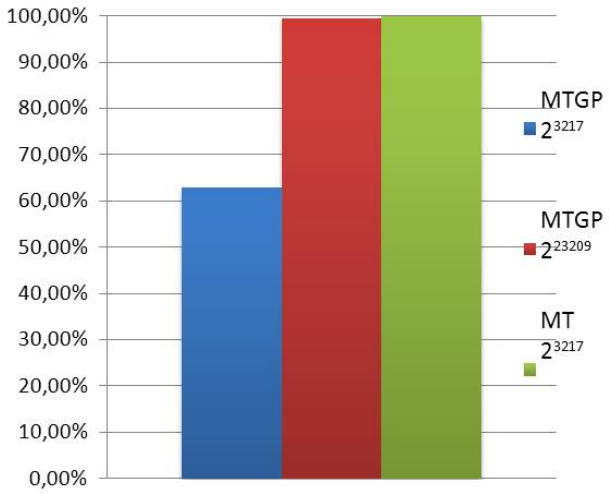


Figure 3.4: Percentage of passed results noticed for tests 35 and 100 depending on the PRNG

3.2.5 Summary about MTGP

This work intended to study a particular generator designed for GPU hardware architectures: MTGP. After an introduction to other recent approaches mentioned in the literature in Chapter 2.4, we provided a description of MTGP and proposed a generic object model representing generators using distinct seeds and parameters in Chapter 2.4.5. To complete our description, we achieved in this chapter some analysis of this PRNG by facing it to the current most stringent test battery: BigCrush from the TestU01 test software library. Weaknesses identified during these experiments have been reduced by comparisons with other configurations of the generator as well as with the original MT.

We have shown that the first implementation of MTGP was safer with longer periods, according to TestU01 criteria, but the longer the period of a PRNG is, the more space it consumes to store the internal state vector used by its algorithm. Nowadays, GPUs memory characteristics do not allow us to waste bytes to store PRNG data without influencing the whole application speed. By selecting only Parameterized Statuses referenced by our study or proofed by an equivalent benchmarking protocol, scientists using MTGP with the lowest available period (i.e. 2^{3217}) can significantly reduce the memory footprint of their hybrid stochastic simulations. In a way to lower the impact on the memory footprint, the authors of MTGP have more recently proposed TinyMT (see Section 2.4.6.2).

As of now, problems regarding MTGP have been solved by their authors. Still, the experimental protocol described in this chapter has proved to be a good way to winnow good Parameterized Statuses for new PRNGs. However, we should not forget that the empirical tests provided by TestU01 consider random sequences individually. They are not a silver bullet to determine good PRNGs from bad ones, and its results should only be considered as elements of a thorough benchmark.

3.3 Requirements for distribution techniques of random streams on GPU

The literature is full of references describing the profile of what a good usage of parallel PRNGs should be. For example, [Coddington, 1996; Hellekalek, 1998b] list requirements that any sequential or parallel PRNG should meet. GPUs are particular parallel architectures, so any PRNG running on this kind of device should, at least,

match 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, [Coddington, 1996] 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 the parallelization principles of GPU-enabled stochastic simulation rely on it. First, it is a necessary, but not sufficient, condition to fulfil in order 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, GPU programming frameworks offer a thread scope rather than a processor scope. Threads in use for GPU programming 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. Beyond 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 [Coddington, 1996] who advocates that *the generator should work for any number of processors*, our second GPU specific requirement for parallelization techniques of random streams is that *it must be usable for any number of GPU threads.*

Returning to the original requirements, we then find in [Coddington, 1996] 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. Please note that this requirement is neither affected by GPU architectures nor by programming frameworks. As a result, we will just recall it without modifying

its expression.

[Coddington, 1996] 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 is no longer ensured through executions. 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 (DOEs) for simulations, where reproducibility is mandatory to isolate the impact of parameters variations on results.

In the case of GPUs, 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 3.5:

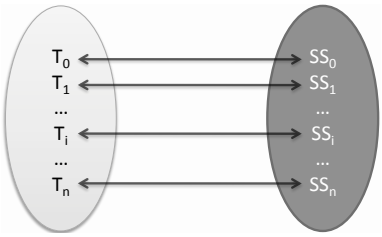


Figure 3.5: Bijective relation between threads and stochastic streams

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:

1. Each thread should dispose of its own random sequence;
2. The parallelization technique must be usable for any number of GPU threads;
3. The parallel random streams produced should be uncorrelated;
4. 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.

3.4 Random streams parallelization techniques fitting GPUs

This subsection presents how the main techniques used to distribute pseudorandom streams between Processing Elements, introduced in Chapter 2.3, can be adapted to GPU architectures, depending on their ability to fulfil the previously introduced requirements. Please note that for the sake of readability, most figures from Chapter 2.3 have been reproduced in the following sections.

3.4.1 Sequence Splitting

Sequence Splitting implies to know 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 on and so forth. Following the previous formalism, we have $Y_i = \{X_{iL}, X_{iL+1}, \ldots, X_{(i+1)L-1}\}$. The repartition of these numbers is described in Figure 3.6.

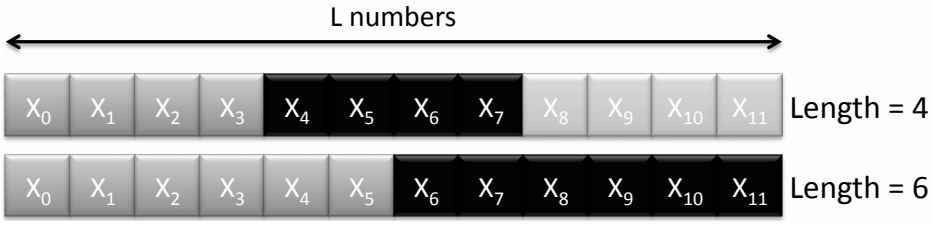


Figure 3.6: Two random streams parallelizations based upon Sequence Splitting with two different sub-sequences lengths

Efficient Sequence Splitting relies on a particular feature of the PRNG called Jump Ahead, or Skip Ahead. Here, we discern two categories of algorithms. Some PRNGs

contain an algorithm able to perform Jump Ahead, thanks to intrinsic properties of the PRNG. Let this technique be considered as Intrinsic Jump Ahead hereafter. This is a relatively recent feature for MT for instance [Haramoto et al., 2008]. This allows us to reach any part of the sequence in equal time, regardless of the destination point. Still the skipping time can vary from one PRNG to another. For instance, the Jump Ahead algorithm of MT skips in the sequence at the cost of some milliseconds whereas the algorithm of MRG32k3a only spends a few microseconds at skipping ahead [Haramoto et al., 2008].

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 $X_{iL+1}, ..., X_{(i+1)L-1}$ 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 [Passerat-Palmbach et al., 2010], 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.

When an intrinsic Jump Ahead algorithm is available for the involved PRNG, Sequence Splitting is a very good approach for GPUs. However, as far as we know there are few GPU ports of algorithms with Jump Ahead features. At the time of writing, we are only aware of an MRG32k3a implementation detailed in [Bradley et al., 2011] and of the recent Tiny Mersenne Twister (TinyMT) [Saito, 2011], available to download but not described in any scientific paper yet.

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, threads will require different computation times to process their own state to jump to. Thus, the SIMD parallelism would be shrunk every time a new pseudorandom stream is created in the algorithm. The overall speedup of the application would consequently decrease. 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.

3.4.2 Random Spacing

The initialization process of Random Spacing consists in building a random status (thanks to random numbers from another generator), and to set it as the seed of the

considered PRNG. Consequently, it fits GPUs well, since this operation can be done in parallel without any constraint. In Figure 3.7, we have sketched the use of Random Spacing to issue a random stream assigned to each of the 3 threads represented. Yet, the risk of overlapping between sub-sequences must be evaluated according to the amount 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 really negligible.

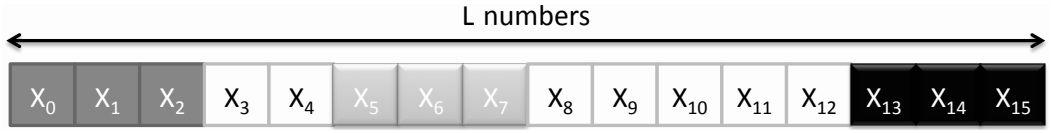


Figure 3.7: Random Spacing creation of three sub-sequences of equal length but differently spaced from each other

3.4.3 Leap Frog

Leap Frog is not quite adapted to split random streams on GPU since it does not satisfy the last constraint expressed in the previous subsection. In fact, if the number of threads changes, the subsequence assigned to each thread will be different. This situation is shown in Figure 3.8. 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. Blocks of threads are scheduled to enable all the threads to be executed. However, the behaviour of the thread scheduler is not deterministic. As a result, we would not be able to ensure the reproducibility of a simulation from a GPU to another, but also from one execution to another. In this case, initializing the PRNG with the same parameters is not sufficient to ensure reproducibility.

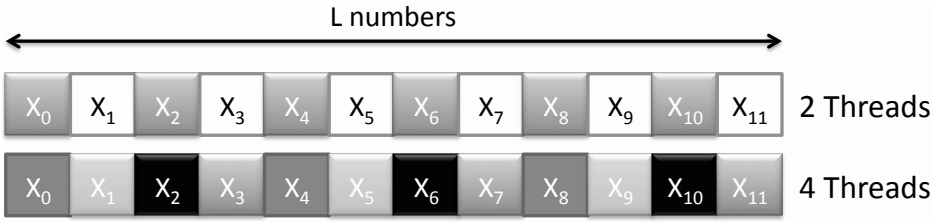


Figure 3.8: Different threads numbers leading to different random substreams through the Leap Frog method

The solution would be to implement the PRNG at a scope where the number of threads is constant. 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. A great number of old GPUs (prior to the Fermi generation when thinking of NVIDIA hardware) did not have enough shared memory to store a PRNG status per warp. However, the newest generations of GPUs offer larger shared memory areas that enables us to use Leap Frog following this idea.

3.4.4 Parameterization

Parameterization displays some constraints making it difficult to port to GPU. Unfortunately, few PRNGs propose it intrinsically. Some, such as LCGs, are even reckoned as bad candidates for Parameterization [De Matteis and Pagnutti, 1988]. In addition, storing Seed Statuses (basically the common seed given by the user to initialize a generator, or the internal state vector of the PRNG) being already problematic, we can scarcely imagine spending vast amounts of memory to store a Parameterized Status per thread. As with Leap Frog earlier, we could overcome this issue by assigning Parameterized Statuses to a higher scope in the thread hierarchy. This approach will be studied in Section 3.5.

3.4.5 Summary

Every technique introduced so far, 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 3.1 states PRNG kinds and parallelization techniques that work well together.

3.5 Implementing PRNGs on GPUs

3.5.1 GPU specific criteria for PRNGs design

As a result of the SIMD parallelism between threads and of their graphics processors legacy, GPUs are not equivalent to a set of standard processors used in parallel. These

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

Table 3.1: Summary of the potential PRNG/Parallelization technique associations

particularities introduce some constraints that need to be satisfied if we do not want to see the overall simulation performance drop significantly. Thus, we will introduce in this subsection the requirements we find compulsory for a PRNG to run efficiently on a GPU architecture.

The main goal targeted when using GPUs is to improve the execution speed of an application. However, GPUs have not been primarily designed to support general computations and are more inclined to perform some arithmetic operations. Nowadays GPUs still display different performances with single and double precision floating point numbers, in accordance with the IEEE 754-2008 standard. For instance, the first generation of NVIDIA supercomputing-dedicated GPU, the Tesla T10, was known to compute double precision floating point operations ten times slower than simple precision operations. Even if the current cutting-edge GPU generation, the NVIDIA Kepler, has considerably reduced the gap between these two precisions (a factor 2 still exists), it is 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 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. The capabilities of these memories, i.e. their capacity and response time, depend on two characteristics.

First, the more threads can reach a memory area, the slower it is. In fact, registers allocated to a single thread are the fastest memory this thread will be able to commu-

nicate with. Close to the same speed, 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. This latter area is far slower than its counterparts (a few hundreds of GPU cycles are necessary for a basic global memory access!).

Second, read-only memories are faster 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.*

The memory hierarchy of GPUs is sketched in Figure 3.9. 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 GPU random number generation algorithm;
2. The algorithm should be designed in a way to avoid global memory accesses.

3.5.2 GPU Memory areas and the internal data structures of PRNGs

We focus here on the implementation level of PRNGs on the GPU, which describes the memory area where the data of the algorithm is located. In [Passerat-Palmbach et al., 2010], we identified three implementation levels, mapped on the CUDA thread hierarchy: threads, blocks of threads and grid of blocks. These strategies directly impact the implementation of the PRNG, so as the parallelization technique coupled with it. Let us introduce them to understand the technical prerequisites about the 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 3.9.

On the one hand, the thread organization is meant to maximize the performances of the application. Threads are bundled into blocks of threads to be assigned to one of the Streaming Multiprocessor (SM) of the GPU. SMs can mostly be considered as the GPU equivalent to CPU cores with vectorization capabilities. The important point is that they have their own thread scheduler. The scheduler champions threads which data are available. Selected threads then run on Streaming Processors (SP): the processing units of SMs.

The matching notion of heterogeneous memories comes into play at this point. Threads can access several memories displaying various capacities, which we will discuss more thoroughly in a further subsection. For now, we just need to keep in mind the hierarchy of memory areas: i.e. the classification of memories based on their response time and visibility from threads. From the fastest to the slowest, threads can access: registers, shared memory, local memory and global memory. This enumeration does not take into account any constant memory since they cannot be written from kernel programs. Thus, they would not be able to store the produced random numbers. While registers and local memory are dedicated to a single thread, shared memory is visible to all the threads within a common block (inside a SM). Every thread can obviously also reach global memory.

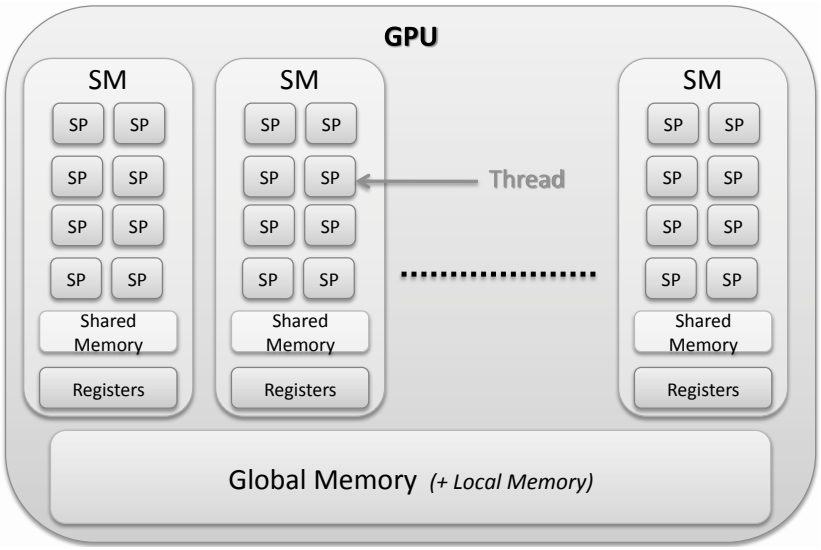


Figure 3.9: Simple representation of the major elements of a GPU

This memory organization highly affects the performances of the PRNG. Considering the first implementation level, i.e.: using a generator per thread, the internal state of the PRNG has to be saved in the local memory of each thread. Most of the time, internal states are formed by several elements contained in arrays. Now, CUDA re-

lated works, like [Kirk and Hwu, 2010], specify that arrays declared within a thread are automatically stored in local memory. Although its name seems to indicate a thread scope, please note that local memory is actually a subset of global memory, and suffers consequently from the same slowness. This area is also allocated to threads when their register set is depleted, since registers are available in limited quantities on GPUs. Usually, new generations of architectures offer a larger amount of registers per thread. Depending on the register occupancy of the application, it can allow small-memory-footprint PRNGs to store all their data either in the register, or in the shared memory area.

Equivalently, with the third cited 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. In [Zhmurov et al., 2010], the authors present three basic generation algorithms working either with a single instance of the PRNG for the kernel or with an instance per thread. The three algorithms exposed are quite basic: Ran2, Hybrid Taus and a Lagged Fibonacci generator. In the same way, [Langdon, 2009] chooses to generate a number per thread in his GPU version of the Park-Miller algorithm. These two approaches make a heavy use of global memory. This has the advantage of being persistent across kernel launches within the same application. It is however important to realize that this area used to be quite slow: it implied a 400 to 800 clock cycle latency because it was not cached [NVIDIA, 2010b] in the first CUDA-enabled architecture. Once again, new generations have improved memory access time. However, the global memory approach will still display a worse latency than its counterparts.

The second implementation scope, the block of threads 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 area. This enables every thread of the block to update it. Shared memory is implemented on-chip and is therefore announced as fast as registers. Thus, PRNGs implemented at a block level will not suffer from the memory latency induced by slow global memory accesses.

We could also imagine a variant of this block of threads scope: assigning a PRNG per warp. The concept of Warps, as introduced by NVIDIA, corresponds to a subgroup of threads dynamically formed by the device at runtime: threads within a warp achieve memory accesses in parallel. Warps are thus the smallest GPU units that are able to process independent code sections. Indeed, given that different warps either run on different SMs, or on the same but at different clock ticks, they are fully independent

to each other. Because of memory latency, warps-schedulers select the warps that have their data ready to process. In the end, the more warps can be scheduled, the better the memory latency can be hidden.

In order to implement a PRNG at a given level, the following requirements must be met: first we need a common memory area accessible by every member of the group. In the case of warps, the shared memory area assigned to their belonging block will perfectly suit. Second, in order to build their own random sequence, processing elements need to be able to distinguish their corresponding PRNG. There is no problem to do so when dealing with a PRNG implemented at either thread, block or kernel level, since CUDA provides us a way to uniquely identify each of these elements. Although warps identifiers are not directly available through CUDA keywords, we have shown in another study how a thread could request the identifier of its parent warp [Passerat-Palmbach et al., 2011a].

The three main implementation scopes detailed in this subsection are sketched in Figure 3.10.

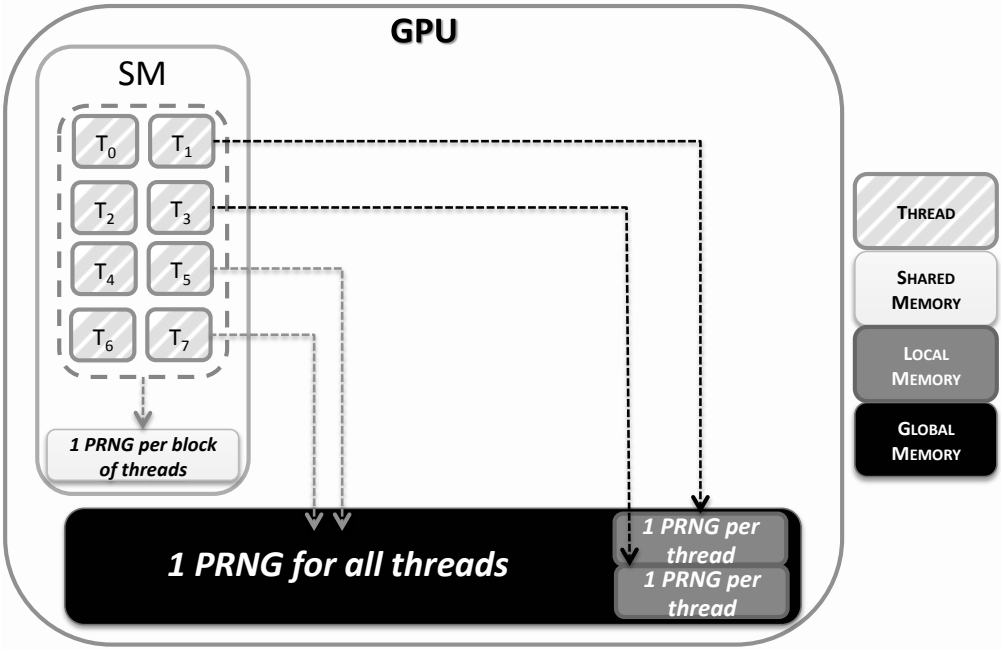


Figure 3.10: PRNGs implementation scopes and their location in the different memory areas of a GPU

As a conclusion, no matter which strategy we choose to implement random number generation facilities on GPU, we will always have to deal with distribution techniques. Such techniques could be hidden in a well-designed library, or directly applied by the simulation developer.

3.6 Discussions and taxonomy of random streams distribution techniques

For techniques such as Sequence Splitting and Random Spacing, we have seen that a common problem is overlapping, but we also have to consider the potential impact of the random initialization on the quality of the underlying PRNG. Recent history has shown that even some of the best RNG algorithms could fail when badly initialized. In the first version of the Mersenne Twister generator, if two initial states were too close with respect to the Hamming distance, then the corresponding output sequences were close to each other. Improvements have been proposed to overcome this problem[2]. For the initialization problem, the remaining technique is to run empirical or statistical tests such as TestU01. In 2008, Romain Reuillon proposed 1 million statuses for the first Mersenne Twister of period $2^{19937} - 1$: he used a RNG with cryptographic qualities to propose independent and well-balanced bit statuses, knowing that when the first MT had a zero-excess initialization status (the problem was corrected in a further 2002 version), it could take a quite long number of draws to recover good statistical properties [Reuillon, 2008a].

In Figure 3.11, we propose a Unified Modeling Language (UML) class diagram of the main parallelization techniques. The latter are basically ordered depending on the use of either a single stream or multiple streams. Then, the taxonomy is refined considering the strategies set up by distribution techniques: for instance, we distinguish techniques issuing blocks of contiguous numbers in opposition to those dealing numbers. Now, we have seen that techniques based upon unique original random streams might take advantage of a jump-ahead algorithm to improve their generation speed. Basically, this feature concerns the Leap Frog and Sequence Splitting approaches. However, we do not consider jump ahead as a relevant criterion to sort distribution techniques. That is why we decided to allow any *UniqueOriginalStream* instance to make use of jump ahead. Finally, please note that we chose not to distinguish a particular hybrid approach. We consider as hybrid any technique combining at least two of its counterparts. This has

[2] `http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/MT2002/emt19937ar.html`

been carried out, thanks to the Composite design pattern on the right side of Figure 3.11. It allows us to describe the Hybrid Approach as a class that represents the combination of at least two other distribution techniques.

In a previous work [Reuillon et al., 2011], we have presented the DistMe software toolkit designed to help with the distribution of large parallel stochastic applications. It is a concrete implementation of the techniques and advice provided all along this chapter and is a concrete implementation of our experience in dealing with pseudorandom numbers for distributed simulations. This paper also presents examples of stochastic simulations for life science research where thousands of billions of pseudorandom numbers have been used. The evolution of this software toolkit (OpenMOLE) has been presented in [Reuillon et al., 2010, 2013]. The preliminary design of the DistMe toolkit was achieved when tackling the distribution of a nuclear medicine application using the largest European computing grid (EGI) [Li and Mascagni, 2003; Maigne et al., 2004; El Bitar et al., 2006]. At that time, we used Sequence Splitting and the dynamic creation of Mersenne Twister algorithms [Reuillon et al., 2011]. Thanks to the EGI computing grid, the equivalent of a few years of computation was achieved in a few days. An interesting example of Leap Frog usage is given in [Janowczyk et al., 2008], and efficient usage of the Jump-Ahead technique is given by simulation software using *Scalable Library for Pseudo-Random Number Generation* (SPRNG) [Mascagni and Srinivasan, 2004], *SSJ* [L'Ecuyer, 2001] and the variants of *rstream* [L'Ecuyer and Buist, 2005].

3.7 Choosing the right distribution technique

To sum up the suggestions stated in this section, Table 3.2 proposes another taxonomy of distribution techniques. On the one hand, we have focused on the environment, where the simulation is supposed to run. On the other hand, we have considered the underlying PRNG algorithm issuing the original stream to be split. This table should help practitioners to figure out which distribution technique they can use, in view of their own environment constraints.

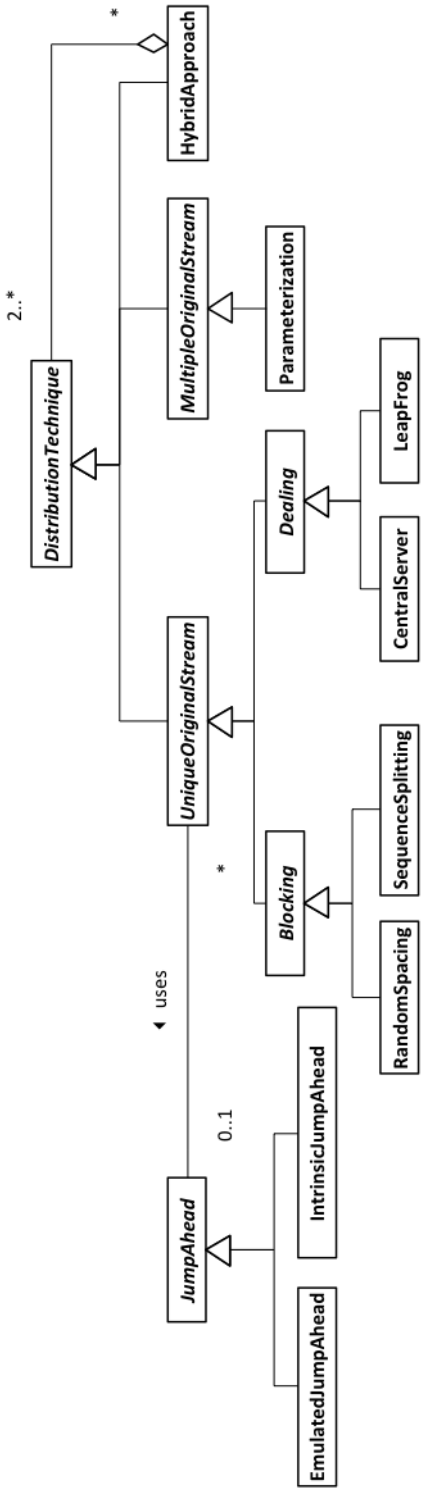


Figure 3.11: Taxonomy of distribution techniques

Distribution Technique	**Where to use it?**	**Where to avoid it?**	**Compliant families of PRNGs**
Central Server	Serious Games	Scientific Applications	Any
Sequence Splitting	Limited-memory environments like GPUs	When long-range correlations in the initial RNG can lead to small range correlations between the potential substreams.	Any Jump-Ahead enabled PRNG (MRG32k3a, Counter based, ...)
Random Spacing	Limited-memory environments like GPUs	When the period of the RNG is smaller than 2^{60} and when a single simulation is consuming more than 10^{11} random numbers.	Large-period PRNGs (WELL, Mersenne Twister, ...)
Leap Frog	Fixed-grained parallelism	• When the Leap Frog is implemented with a Lagged Fibonacci Generator, we have to avoid distributed environments where the number of Processing Elements is not a power of 2. • When the parallelism grain is not constant (disables reproducibility)	Any
Parameterization	Distributed environments	Limited-memory environments (GPUs, ...)	PRNG issued from a parameterization algorithm (MT family, Counter based,...)

Table 3.2: Summary of the potential uses of Distribution Techniques

3.8 Conclusion

A 'good' generator is one where statistical defaults are well hidden, although they are not inexistent because even the 'best' known generators can fail for a particular application or when they are badly initialized. Parallel simulations must first consider the choice of a good sequential generator, but as usual, no statistical or practical test is universal because we cannot prove the statistical soundness of a generator. Test batteries are the most convenient empirical way to ensure that the generator in use fits well with a wide set of common applications. Such batteries can be used to test independently several streams to be used in parallel. In addition, for very sensitive simulations, it can be a good practice to test different generators and partitioning techniques to check the stochastic variability of results and to increase the reliability of simulation results.

In the particular case of using parallel random number streams, additional care should be taken to ensure that we avoid long-range correlations, even though it often implies an additional computing cost. In our opinion, it would be interesting to include random generator partitioning and advanced random streams testing facilities in HPC middleware for the scientific end users. An attempt to do so was proposed by Romain Reuillon in his PhD thesis with the DistMe, DistRNG and DistTest toolkits [Reuillon et al., 2008].

This chapter introduced the main problems that will be encountered by a simulation practitioner trying to port a stochastic simulation to 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. In order to take advantage of the existing PRNG parallelization techniques on GPU, we defined a set of requirements that should be met by any PRNG implementing a distribution technique on GPU.

Then, we studied the applicability of widespread distribution techniques on GPU, and determined which techniques best matched GPU constraints. These requirements and chosen techniques sum up the experience accumulated in our research team concerning GPU-enabled stochastic simulations. They resulted in a taxonomy of the distribution techniques and conclusions about their suitability in various concrete use cases and parallel environments.

Finally, we focused on PRNG implemented on GPU to give another set of guidelines regarding the specificity of the architecture of GPUs. These guidelines take into account

both the thread scheduling, and the memory hierarchy.

In the next chapter, we will make use of good software engineering practices to implement the guidelines within software libraries targeting GPUs and manycore CPUs. These developments match the theoretical advice provided in this chapter as well as good software practice which some HPC tools sometimes lack.

Chapter 4

Design and Implementation Proposals for Modern HPC Frameworks

> “ *In this case, as opposed to the scrupulous method of plain good taste and scientific grooming, the trick had been worked by exaggerating defects, she'd made them ornamented by admitting them boldly.*
>
> — Truman Capote, *Breakfast at Tiffany's*

Contents

4.1 Introduction

This chapter exposes 3 designs and implementations resulting from the guidelines and experiences about the distribution of pseudorandom streams in parallel described in Chapter 3. Let us quickly introduce these 3 developments:

- Shoverand, a PRNG library for CUDA-enabled GPUs;
- ThreadLocalMRG32k3a, a counterpart to *ThreadLocalRandom*, from the Java Development Kit 7, using MRG32k3a [L'Ecuyer, 1999];
- TaskLocalRandom, an extended version of ThreadLocalMRG32k3a handling not only Java threads but also Java tasks from Java thread pools.

4.1.1 Purpose of Shoverand

As we have seen in the previous parts of this manuscript, it is very important to deal carefully with pseudorandom numbers distribution when working with parallel environments such as GPUs. Still, we cannot expect any user to be aware of every theoretical consideration that will prevent his simulation results from being biased. Thus, we introduced Shoverand [Passerat-Palmbach et al., 2011a], a framework that provides Pseudorandom Number Generation (PRNG) facilities to CUDA-enabled GPU applications.

Shoverand combines several aspects to ease developments of stochastic-enabled applications on GPU. First, its API is quite similar to what can be encountered when using high-level CPU languages like C++ or Java. Second, Shoverand's main goal is to handle the distribution of stochastic streams automatically without any intervention from the user. Finally, our framework also targets PRNG developers: indeed, Shoverand only integrates third-party PRNGs and focuses on unifying their interface. To do so, we integrate compile-time constraints that check whether the algorithm meets our guidelines.

4.1.2 ThreadLocalMRG32k3a and TaskLocalRandom

The latest release of the Java Development Kit (JDK 7) offers a couple of new tools to enhance the already existing concurrent package. Mainly, a new framework called Fork/Join appears [Lea, 2000]. It provides an easy-to-use MapReduce implementation running in parallel thanks to a pool of worker threads. Such inputs, added to the already present tools allowing the distribution of the computing load across several threads, should attract more and more simulation practitioners to Java development. These users will also bring their own concerns bound to parallelization in their domain of expertise. Thus, simulationists working on stochastic simulations will ask for a tool to help them to partition a random source in a parallel Java environment.

Java 7 tries to tackle this problem in providing facilities to partition a pseudorandom stream across various threads thanks to the new class *ThreadLocalRandom*. This class is in charge of safe pseudorandom number generation across Java threads. Section 4.3 studies the pros and cons of this approach, and introduces ThreadLocalMRG32k3a [Passerat-Palmbach et al., 2012c], an alternative to *ThreadLocalRandom* that shows better results in terms of generation speed and statistical quality. ThreadLocalMRG32k3a respects the same Application Programming Interface (API) as *ThreadLocalRandom*, thus enabling clients to use it in place of its JDK counterpart at no cost.

As any other Java Thread Pool, ForkJoin exploits threads as workers and manipulates the tasks that will be run on the workers. In *ThreadLocalRandom*, pseudorandom number generation is handled at a thread level. As a consequence, a scientific application taking advantage of a Java Thread Pool to parallelize its computation will suffer from a bad pseudorandom stream partitioning due to the behaviour of *ThreadLocalRandom*. Section 4.4 introduces TaskLocalRandom [Passerat-Palmbach et al., 2013c], a task-level alternative to *ThreadLocalRandom* that solves this partitioning problem

and assigns an independent pseudorandom stream to each task run in the thread pool. TaskLocalRandom is compatible with existing Java thread pools such as Executors or ForkJoin. It is an extension of the previously described ThreadLocalMRG32k3a.

4.2 ShoveRand

4.2.1 Introduction

In this section, 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 API to users, regardless of the RNG they choose. 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;
- Present the design choice that helps users and developers to follow the guidelines stated in Chapter 3.

4.2.2 A Model-Driven Library to Overcome Known Issues

In the light of the previous discussions, we spotted lacks in different domains involved in RNG libraries design: first, the embedded RNGs and second, the design of the API. These shortages showed the importance of defining a novel framework for RNG implementations on GPU. To enable developers to use RNGs easily, we believe 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 on which our library relies.

4.2.2.1 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) [Saito and Matsumoto, 2013], benchmarked back in Chapter 3.2 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 [Hill, 2010; L'Ecuyer, 2010] and can be a drawback when we consider the memory footprint of the generator. As a matter of fact, the larger 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). DC issues independent parameter sets, called Parameterized Statuses hereafter. It 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 fulfils 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 4.2.4.1.

4.2.2.2 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 GPU. This leads to heterogeneous propositions of GPU-enabled RNGs. The latter can even sometimes be of very good quality but a bit tough to use for inexperienced users. Thus, our goal is to design this generic RNG framework and furnish an implementation embedding quality-proven 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 GPU-enabled RNGs implementations. We intend to furnish an empty shell that will require a small amount of work to embed a new RNG. In 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 our needs. Hopefully, most of the RNGs share common characteristics that we were able to factorize in few classes. Before any description of the machinery of our library model, let us introduce its schematised version through the UML class diagram in Figure 4.1:

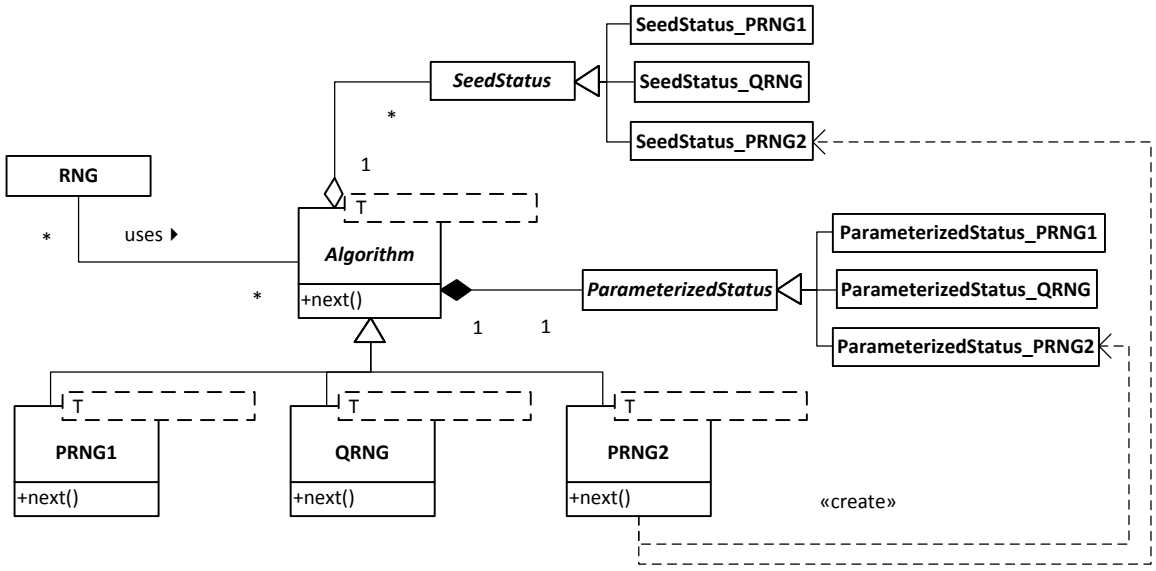


Figure 4.1: Parameterized Version of the Model

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 of four base classes for the library: *RNG*, *Algorithm*, *ParameterizedStatus* and *SeedStatus*. Relations are 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. Thanks to their genericity, that the concrete implementations of these two *Status* classes will be finely tuned for the *Algorithm* employing them. You can notice that this relation is explicitly drawn for *PRNG2* only on the class diagram in Figure 4.1, for the sake of readability. 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 in the figures) of the generated random numbers. Common data types are 32-bit or 64-bit integers, or floating-point numbers of single or double precision. Thus, parts of the implementation can, again, be tuned specifically for a data type. The library is also abstracted from the underlying programming framework used to develop applications that will run on GPU. Indeed, the core of the library is not bound to a particular GPU device code technology, allowing developers to write RNGs implementations in their favourite 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. However, at the time of writing, OpenCL does not support the object-oriented constructs that would allow us to implement such a model.

These features of the meta-model are intended to guide developers in the way they embed their PRNG algorithms into Shoverand. The homogeneous API exposed to users directly results from these upstream design guidelines. In conclusion, formulating guidelines for developers serve the end-users. We will see how this meta-model is implemented in CUDA/C++ in the next section.

4.2.3 Meta-Model Implementation

We intend to offer a straightforward API, enabling users to call a RNG without wondering indefinitely which parameters they 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. It can be interesting to quickly replace the RNG used in an application by another one relying on different principles. This allows comparing the output results of a given application, depending on its randomness source. With a uniform API, enhanced by generic programming, the only thing to change is 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.

4.2.3.1 Policy-based class design

Several pseudo-random number generation algorithms can be implemented in ShoveRand through different classes. The *RNG* class actually gets its behaviour from these classes. This situation largely recalls the Strategy design pattern [Gamma et al., 1995], which intends to dynamically modify the behaviour of the class instances according to the Strategy they are combined with. This feature simply relies on virtual functions calls. Until recently, GPUs disabled strong polymorphism, i.e. virtual functions. Now, GPU architectures support the implementation of virtual functions tables, but this feature remains costly in terms of execution time. Yet, virtual functions are required to enable polymorphism. Hopefully, [Alexandrescu, 2001] 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 behaviour. 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 the implementations classes of a policy, called policy classes. They guide the behaviour of these classes to match the initial purpose of the policy. 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.*

4.2.3.2 Strong static typing

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

First, when *RNG* is defined for a particular (T; Algorithm) pair, 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. For instance, NVIDIA's cuRand library exposes functions that take any of the status of any implemented generator as parameter. On the other hand, Shoverand's strong typing feature 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 [Siek and Lumsdaine, 2000] and implemented in the Boost Concept Check Library (BCCL), concept checking enables us to statically check the interface of a given class. This mechanism verifies of the correctness of the interfaces of the policy classes. In the previous part, we have stated a list of constraints 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 4.1.

```
1  // concept requirements
2  BOOST_CONCEPT_USAGE(RNGAlgorithm) {
3    // require Algo<T>::init()
4    al_.init();
5    // require T Algo<T>::next()
6    value_ = al_.next();
7    // require Algo<T>::ss_ to be of
8    // SeedStatus<Algo> type
9    same_type(ss_, al_.ss_);
10   // require Algo<T>::ps_ to be of
11   // ParameterizedStatus<Algo> type
12   same_type(ps_, al_.ps_);
13 }
```

Listing 4.1: The interface of the policy is checked through Boost Concept Check Library

As a result, when users try to provide an *Algorithm* implementation to instantiate a particular *RNG*, the compiler checks whether the input *Algorithm* class meets all the requirements in the list. Beyond the fact of proposing a generic uniform interface to RNGs, this mechanism ensures that future implementations respect the standard we propose. These constraints are present for information purpose. Developers become aware of some lacks in their implementations since they are faced with clear compiling errors, displayed regardless of the utilization of all the methods int their classes. Instead of letting misconception errors lurk in the shadows until someone calls a missing method, errors can be detected at an early stage of development. To conclude, this feature is a full part of the developer-friendly aspect of our library, similarly to the straightforward uniform API.

Policies and concept checking have been integrated in the meta-model. Figure 4.2 shows a class diagram based upon Figure 4.1, but where policies and concept checking have replaced the original Strategy design pattern:

4.2.4 PRNGs embedded in Shoverand

At the time of writing, Shoverand embeds several PRNGs. All these algorithms have been selected according to their intrinsic properties. We first consider their statistical properties in a sequential environment, because a PRNG could not cope with the requirements of parallel environments if its sequential version was poor. Conse-

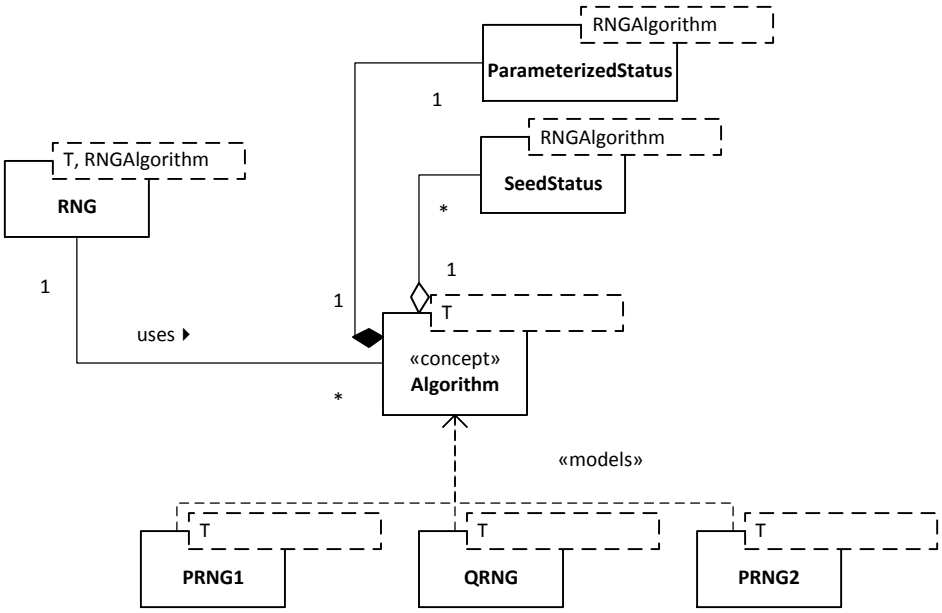


Figure 4.2: Meta-Model Describing the ShoveRand Framework

quently, every PRNG wrapped in Shoverand must satisfy the most stringent testing battery currently available, namely BigCrush from TestU01 [L'Ecuyer and Simard, 2007]. PRNGs that pass all those tests are referred to as "Crush-resistant" in [Salmon et al., 2011]. While being Crush-resistant cannot ensure a perfect randomness of the considered pseudorandom stream, it is a satisfying property of which few PRNGs can be proud.

Additionally, the retained algorithms must support a reliable technique to distribute numbers in a parallel environment. We have previously surveyed such techniques in [Hill et al., 2013], but only some of them can be applied to a GPU platform [Passerat-Palmbach et al., 2012a]. The chosen ones are then ideal candidates to be ported to GPU, if not available yet, and moreover to be integrated in Shoverand. We detail hereafter the PRNGs that are currently, or will soon be, embedded in Shoverand.

4.2.4.1 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 pseudorandom numbers on GPU to get them back to the host side afterwards. However, we want to generate numbers that can be directly consumed by the application currently running on the GPU. Thus, we had to slightly modify

Saito's original proposition to change its behaviour regarding generated numbers. The original version was able to generate three numbers per GPU thread, reducing this way memory transfer latency between host and 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 pseudorandom numbers from three 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. Now, the MTGP algorithm can be used 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 MTGP, 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 pseudorandom numbers:

```
1 __global__ void testShoveRandMTGP(int* outputData) {
2
3   shoverand::RNG< int, MTGP > rng;
4
5   outputData[threadIdx.x] =
6           rng.next() * rng.next();
7 }
```

Listing 4.2: MTGP Integrated in ShoveRand

4.2.5 Case study: generating pseudorandom numbers in a CUDA kernel with Shoverand

In this section, we describe a major aspect of Shoverand: its user-friendly interface. We will see that on both host and device sides, our API is very expressive while remaining very concise. Shoverand competes with two major counterparts in the CUDA world, both coming from an NVIDIA initiative, named Thrust [Hoberock and Bell, 2010] and cuRand [NVIDIA, 2012]. These two libraries are also providing random number generation features but vary from Shoverand on several points that we will compare in this section.

4.2.5.1 Host side: Initialization phase

From the end-user's point of view, Shoverand requires an initialization phase in order to allocate its internal data structures on the device and perform some initializations. As a matter of fact, depending on the chosen PRNG, initialization might involve external data to be read from a parameter file, as it is the case with MTGP for instance.

We previously saw that we needed to consider the distribution technique and the PRNG algorithm as a pair. As a consequence, distribution techniques vary from one PRNG to another in Shoverand, but their initialization phase require the same data, which is basically the number of CUDA blocks that the kernel using the PRNG will spawn. This data being stored in the memory of the device prior to the kernel call, no superfluous parameter needs to be passed to the kernel. This feature allows users not to have their hands tied when designing their kernels, since Shoverand does not impact the prototypes kernels like other libraries do.

As a result, the host side initialization phase boils down to a single call to a static method named *init()*. This method must be provided by every PRNG implementation to satisfy Shoverand's rules.

4.2.5.2 Device side: Computation phase

Using the device side of Shoverand is even simpler than the host side. You only have to create an instance of the PRNG you want to use and let the constructor of its class do the rest. Device side initializations are performed behind the scenes by the constructor, so that users have nothing to do. Then, random numbers are picked up by calling the *next()* method on the previously created object. This process is really intuitive for

end-users used to random number generation facilities offered by high-level languages such as Java.

4.2.5.3 Comparison with Thrust and cuRand

Thrust and cuRand are two projects developed by NVIDIA fellows. While cuRand is part of the CUDA SDK, Thrust is an external open-source library that can be downloaded from an Internet repository. In the paper originally introducing Shoverand [Passerat-Palmbach et al., 2011a], we had surveyed these two libraries and identified their major drawbacks that led us to design Shoverand. The following lines investigate the changes brought by the state-of-the-art versions of Thrust and cuRand.

cuRand is NVIDIA's solution to random number generation on GPU. In [Passerat-Palmbach et al., 2011a], we mentioned that cuRand suffered from the poor statistical quality of the PRNGs it embedded. The last version of this library partially solves this problem by following the advice we made in our previous paper. Now, cuRand embeds renowned high-quality PRNGs such as MRG32k3a [L'Ecuyer, 1999] and MTGP [Saito and Matsumoto, 2012]. These two PRNGs are known as "Crush-resistant" in the literature.

On the other hand, cuRand's API remains poor and forces users to add extra parameters to the prototypes of every kernel taking advantage of the library. The C API is not generic and loses its consistency when non-default options are enabled: for instance, the initialization step of MTGP is achieved through a dedicated call in cuRand. This call is totally irrelevant when used with another PRNG. This approach is then not convenient when you want to quickly swap PRNGs to study the impact of various random sources on a given application. Listing 4.3 shows a code snippet of what the initialization and utilisation of cuRand might look like (adapted from the CUDA toolkit documentation[1]).

```
1 __global__ void setup_kernel(curandStateMRG32k3a *state)
2 {
3     int id = threadIdx.x + blockIdx.x * 64;
4     /* Each thread gets same seed, a different sequence
5        number, no offset */
6     curand_init(0, id, 0, &state[id]);
7 }
8
```

[1] http://docs.nvidia.com/cuda/curand/index.html#topic_1_3_6, *last access 7/29/13*

```
9  __global__ void generate_kernel(curandState *state)
10 {
11     int id = threadIdx.x + blockIdx.x * 64;
12     unsigned int x;
13
14     /* Copy state to local memory for efficiency */
15     curandState localState = state[id];
16     /* Generate pseudo-random unsigned ints */
17     for(int n = 0; n < 10000; n++) {
18         x = curand(&localState);
19
20          ...
21     }
22     /* Copy state back to global memory */
23     state[id] = localState;
24
25      ...
26 }
```

Listing 4.3: Example of use of the cuRand API

Thrust is an open source GPU-enabled general purpose library developed by NVIDIA fellows. This project aims at providing a GPU-enabled library equivalent to some classic general-purpose C++ libraries, like STL or Boost. Classes are split through several namespaces, such as Thrust::random. The latter contains all classes and methods related to random numbers generation on GPU. Thrust::random implements three PRNGs, each through a different C++ class template. We find a Linear Congruential Generator (LCG), a Linear Feedback Shift (LFS) and a Subtract With Carry (SWC), which are stated as not adapted to High Performance Computing.

Still, Thrust offers a nice API that mirrors the API of Boost. The *random* namespace provides user-friendly features very close to Shoverand's abilities. For instance, neither explicit initializations nor parameters are required in order to benefit from random number generation facilities in a kernel. Listing 4.4 exposes a device function making use of Thrust::random to pick up pseudorandom numbers (this code snippet is adapted from the online Thrust documentation[2]).

[2] `https://github.com/thrust/thrust/blob/master/examples/monte_carlo.cu`, *last access 7/29/13*

```
__host__ __device__
float operator()(unsigned int thread_id)
{
  unsigned int seed = hash(thread_id);

  // seed a random number generator
  thrust::default_random_engine rng(seed);

  // create a mapping from random numbers to [0,1)
  thrust::uniform_real_distribution<float> u01(0,1);

  // take N samples in a quarter circle
  for(unsigned int i = 0; i < N; ++i)
  {
    // draw a pseudorandom number
    float x = u01(rng);

    ...
  }

  ...
}
```

Listing 4.4: Example of use of the Thrust API

As a conclusion, we have on the one hand cuRand, a library that is improving after having been criticized by the research community for the statistical characteristics of its embedded PRNGs, and that exposes a restrictive API; and on the other hand, we have Thrust and its nice "à la Boost" API, yet powered by poor quality PRNGs. Shoverand is based upon the good achievements from these two libraries: it exposes a user-friendly API while integrating only Crush-resistant PRNGs.

Listing 4.5 shows off how to pick up pseudorandom numbers from Shoverand. In this code snippet, we consider both the initialization on the host side and the random number generation on the device side:

```
using shoverand::RNG;
using shoverand::MRG32k3a;

// kernel using Shoverand
```

```
5  __global__ void fooKernel(float* ddata) {
6
7          RNG < float, MRG32k3a >          rng;
8
9          ddata[blockDim.x * blockIdx.x + threadIdx.x] = rng.next();
10 }
11
12  ...
13
14 RNG< float, MRG32k3a >::init(block_num);
15
16 fooKernel <<< block_num, thread_num >>>(d_data);
17
18 RNG< float, MRG32k3a >::release();
```

Listing 4.5: Example of use of Shoverand

Comparing Listing 4.5 with Listings 4.3 and 4.4, we see that the API of Shoverand is closer to the elegant API provided by Thrust, than to the verbose API of cuRand.

4.2.6 Case study: embedding a new PRNG into Shoverand

Shoverand is not only a library but also a framework that allows PRNG developers to insert their own proposals as long as they follow some rules. In order to help them in their task, let us recall that Shoverand employs a mechanism called concept checking that lets us express constraints in Shoverand's code that will be checked at compile-time for all the classes that inherit from ours. Such a mechanism forces developers to match our interface without having to introduce costly runtime techniques like polymorphism and consequently virtual methods.

Thanks to the Boost Concept Check Library (BCCL), we are able to design constraints that will make any compilation attempt fail if they are not met. In Shoverand, we force every PRNG algorithm class to inherit from our base RNG class. Then, each user-defined subclass must define at least 3 methods: *init()* and *release()*, which deal with parameters allocation and initialization from the host side, and *next()*, which picks up the next random number within a kernel, on the device side.

In the same way, BCCL also permits us to verify that developers have provided two members to their class: the Seed and Parameterized Statuses. These two members

respectively represent the current state of the PRNG and its initial parameters. Fig. 4.3 sketches a UML class diagram of the expected content of a PRNG class to be embedded in Shoverand:

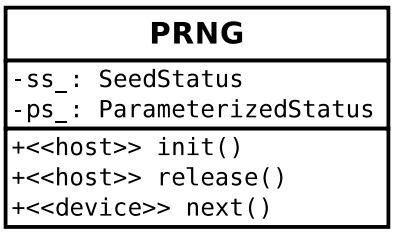


Figure 4.3: UML class diagram of the expected interface of a PRNG in Shoverand

4.2.7 Summary

In our opinion, cuRand and Thrust, the two most widely spread libraries to generate pseudorandom numbers on GPU are not satisfying for scientific applications. 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 and Matsumoto's work: MTGP, introduced in [Saito and Matsumoto, 2013], which is a GPU-enabled parallel PRNG. It was originally written to use the 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 GPU. Part of this work implied to modify the behaviour of this PRNG to make it **generate** a single random number per thread that could be directly consumed by the program currently running on the device. We have also embedded other fine generators from Pierre L'Ecuyer's team, such as MRG32k3a [L'Ecuyer et al., 2002a].

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 design stochastic applications independently of the underlying generator. Other future components will also take advantage of this statically checked skeleton. We plan to implement the distributions features that come with the Boost.Random library. The interesting point here is that ShoveRand will simplify the implementation of distributions because of its concept checking capabilities.

Currently developed in CUDA/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. Unfortunately, at the time of writing, OpenCL is not able to support object-oriented facilities yet.

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

4.3 ThreadLocalMRG32k3a

4.3.1 Introduction

Java 7 introduces the *ThreadLocalRandom* class, a tool that intends to enable developers to deal with pseudorandom numbers in parallel on a single shared memory computer, without having to figure out how to distribute numbers among the available processing elements. The question we have for scientific applications is the following: can *ThreadLocalRandom* serve as a random source with the statistical quality required by stochastic simulations? Although this development is a good initiative that is worth being integrated in Java, we will see that the current implementation still has some major drawbacks for scientific purposes.

The present section will:

- Study *ThreadLocalRandom*'s intrinsics to figure out whether its output is satisfying regarding stochastic simulations needs;
- Present already existing libraries that could serve as alternatives to *ThreadLocalRandom*;
- Introduce ThreadLocalMRG32k3a, our proposal based upon the MRG32k3a Pseudorandom Number Generator (PRNG) algorithm from Pierre L'Ecuyer [L'Ecuyer, 1999];
- Compare ThreadLocalMRG32k3a to *ThreadLocalRandom*, and consider its potential evolutions.

4.3.2 ThreadLocalRandom

4.3.2.1 Implementation concerns

Officially released with JDK 7, the *ThreadLocalRandom* facility was developed within the *jsr166y* initiative by Doug Lea. *ThreadLocalRandom* tries to solve the complexity regarding the use of random sources correctly in parallel applications. Each thread owns a *ThreadLocalRandom* instance, allowing every one of them to be independent from the others to pick up random numbers. Still, the most important point behind this technique is that it is supposed to distribute pseudorandom streams safely among threads.

ThreadLocalRandom inherits from *java.util.Random*, thus sharing its interface. Every thread must call a method named *current()* before calling the classical *nextXXX()* methods to pick up a pseudorandom number which type is indicated by the *XXX* suffix.

ThreadLocalRandom makes use of the Random Spacing technique (see Chapter 2.3.1.3) to distribute pseudorandom streams across threads. This technique consists in initializing an identical PRNG instance in each thread with a different Seed Status [Passerat-Palmbach et al., 2010], the latter being randomly chosen by another algorithm. By doing so, each thread owns an highly independent pseudorandom sequence, provided the PRNG algorithm has a long enough period, and is not subject to long-range correlations [De Matteis and Pagnutti, 1988].

Random Spacing is implemented in *ThreadLocalRandom* through its constructor. The constructor of *ThreadLocalRandom* calls the constructor of *java.util.Random*, then sets a Boolean to true. This depicts that the initialization step has been done and cannot be performed again. *ThreadLocalRandom* must then rely on the constructor of the *Random* class to set its initial seed. Until JDK6, the constructor of *Random* used to automatically perform a call to *setSeed()*, the method in charge of the Random Spacing initialization of the seed. However, this is not true anymore with JDK7. Consequently, any PRNG class that extends Random and relies on it to call *setSeed()*, will see its Seed Status remain uninitialized. According to [Gosling et al., 2005], the seed of each thread is thus set to zero, as any class member of the long type would be when non explicitly initialized. As a result, every thread will pick up the same pseudorandom sequence in such a case.

We have already spotted a similar problem in a Mersenne Twister implemented in

Java. We have proposed a corrective patch that solves it[3]. Calling *setSeed()* in every thread could easily solve this problem. Unfortunately, the *setSeed()* public method, which would normally allow setting the seed of the PRNG of a thread to a new value, is locked by the previously mentioned boolean. Such a feature is important to prevent users to involuntarily harm pseudorandom streams independence between threads by setting several seeds to the same value. However, this also prevents us from adapting the class behaviour, and for example to force a call to *setSeed()* directly in the constructor of the subclass. Moreover, this solution relies on the user-awareness of the problem, which goes against the initial purpose of *ThreadLocalRandom* to hide random streams distribution to the user.

The problem was finally solved in the second update of the JDK7 by changing the constructor of *Random* in order to take into account a potential use of *setSeed()* by subclasses. This change is confusing in two ways. Not only does it break encapsulation, one of the elementary concepts of the object-oriented paradigm, but it also appears as a lack of good software engineering. It is indeed not recommended to adapt an implementation according to an already existing source code. Instead, it is safer to rely on the specification only. In our case, the *Random* class documentation issued by Oracle makes no mention of a potential call to the *setSeed()* method by the constructor of *Random*. As a consequence, we cannot blame Oracle for this bug, but rather advise developers to focus on the official documentation only, especially when they are working on such sensitive aspects of the implementation.

Another weakness in the implementation of *ThreadLocalRandom* lies in the impossibility to reproduce the same pseudorandom sequences throughout several runs of the application by default. Scientific applications such as stochastic simulations need to ensure reproducibility between executions for their results to be checked or for debug purposes. When *ThreadLocalRandom* is used by default, it does not satisfy this need because it relies on the constructor of *Random* to set its internal seed. In that case, the seed is set to the current system time. This could be interesting for games but not for a scientific software. This problem can be fixed by basing the initial seed on a unique identifier for each thread, so that for a given identifier, a thread will always be assigned the same stochastic stream. The assignment of unique identifiers to threads is discussed in Section 4.3.4.2.

Still, although Java provides a thread identifier at runtime through the *Thread.currentThread().getId()* call, this identifier is not reliable since it is global for

[3] `http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/VERSIONS/JAVA/PATCH/MTRandom.patch`, *last access 7/29/13*

the whole JVM. The identifier will vary with the number of threads created before the thread requesting an identifier. Therefore, *ThreadLocalRandom* must be extended with its own thread identifier to make it safe in terms of pseudorandom stream distribution and reproducibility. We have proposed such an extension in [Passerat-Palmbach et al., 2012c].

4.3.2.2 Statistical Quality Discussion

The underlying PRNG of *ThreadLocalRandom*, is a well-known and widely studied LCG from Knuth [Knuth, 1969] (it also rules the output of the POSIX *drand48* C function for example). This LCG belongs to the family of algorithms that cannot pass the most stringent statistical test battery at the time of writing: TestU01 [L'Ecuyer and Simard, 2007] (see Section 2.2 for more information about this test battery). According to [L'Ecuyer, 2010], LCG generators should be discarded from scientific applications since their structure is not adapted to many modern applications. The problem is even bigger when parallel and distributed computing is considered. In addition, the period proposed by *ThreadLocalRandom* is relatively small for modern scientific applications: it is 2^{48} numbers long. Pierre L'Ecuyer suggests that for modern applications periods should be at least 2^{100} numbers long [L'Ecuyer, 2010].

In regards to the parallel utilization of *ThreadLocalRandom*, we can barely imagine that such a bad generator [Ferrenberg et al., 1992; Hellekalek, 1998b,a] could behave better in a parallel environment. Thanks to TestU01 parallel filters [L'Ecuyer and Simard, 2007], we can easily create a random sequence formed by the combination of any number of input sequences from different *ThreadLocalRandom* initializations. However, as stated in [Salmon et al., 2011], it is impossible to perform a complete coverage of all possible logical sequences, because many strategies can be set up to distribute both threads and random streams across parallel computational units. Nonetheless, some samples are particularly representative of how most users will use random sequences, and we will study them in Section 4.4.5.2.

4.3.3 Related Works

Several attempts to provide a user-friendly interface to generate random numbers in parallel environments can be found in the literature. Here we recall the major proposals that can compete and replace *ThreadLocalRandom* in scientific applications. We only consider frameworks that provide ways to automatically distribute pseudorandom

streams through threads without the user's help.

As we have seen previously, the standard Java library only ensures thread safety through synchronized methods when accessing the random number generation features of the *java.util.Random* class. This approach is not satisfying in the world of High Performance Computing (HPC): in addition to not ensuring reproducibility of simulations because of thread scheduling and of scaling problems, it impacts performance of parallel stochastic applications because of the sequential bottleneck implied by the synchronization guarding random facilities. This method to partition pseudorandom sequences is known as Central Server in the literature [Hill et al., 2013].

JAPARA [Coddington and Newell, 2004] was proposed by Coddington and Newell in 2004 to tackle this lack in Java libraries. They bring up a Java API to support parallel generation of random streams. JAPARA proposes that every Processing Element (Java threads in that case) handles its own pseudorandom stream. In doing so, only the initialization phase is synchronized, and a referenced partitioning technique is then used to distribute the underlying pseudorandom streams. JAPARA comes with three PRNGs implemented, each coupled with a distribution technique that matches its intrinsic characteristics. The user only has to select the PRNG he wants to employ, and then rely on the framework to ensure independence between the different streams assigned to the threads. Furthermore, JAPARA allows the user to save and restore the current state of a PRNG, thus permitting to checkpoint a simulation.

After having first proposed a random number package with splitting facilities [L'Ecuyer and Côté, 1991], l'Ecuyer's team proposed an object-oriented pseudorandom number generation package in 2002 [L'Ecuyer et al., 2002a]. This was achieved in the rstream library [L'Ecuyer and Leydold, 2005] that implements a single MRG32k3a PRNG, which independent streams are partitioned from an original stream thanks to the Sequence Splitting technique (see Chapter 2.3.1.2). A declination of rstream comes with the SSJ (Stochastic Simulation in Java) [L'Ecuyer et al., 2002b] framework as the pseudorandom streams parallelization utility of the library. It provides a greater set of PRNGs (including the famous Mersenne Twister [Matsumoto and Nishimura, 1998] for instance), and a compliant set of distribution techniques.

The latest Java random number generation framework that has retained our attention is DistRNG [Reuillon, 2008b; Reuillon et al., 2011]. While its API does not diverge from the two other proposals described in this section, DistRNG focuses on correct partitioning of random streams. To do so, this framework handles XML generic statuses that model any PRNG state. Every Processing Element is initialized with a different XML status that needs to be built upstream. DistRNG displays a fine choice

of statistically sound PRNGs according to the TestU01 reference testing library.

In conclusion, this section has shown that several satisfying proposals of APIs for parallel pseudorandom number generation can be found in the literature. Consequently, users have many reliable solutions at their disposal if they want to take advantage of statistically sound pseudorandom sequences in their Java applications. Moreover, most of these solutions can replace *ThreadLocalRandom* features but require modifications on the application source code to meet their functioning requirements.

4.3.4 MRG32k3a Implementation

In this section, we present the Java implementation we made of the MRG32k3a PRNG, described by Pierre L'Ecuyer in [L'Ecuyer, 1999]. Several features of this algorithm retained our attention, from its internal data structure to the results it displays when faced to today's most stringent testing batteries.

4.3.4.1 The choice of MRG32k3a

Talking about its internal properties, MRG32k3a is really suited to parallelization among small computational elements such as threads, because its lightweight data structure only stores 6 integers to handle its state. It means that introducing this PRNG in already existing Java applications will have roughly no impact on their memory footprint. The algorithm itself is quite short, relying on simple operations only to issue new random numbers. The parameters chosen for MRG32k3a are such that it has a full period of 2^{191} numbers. This period is fairly enough since L'Ecuyer suggests that periods between 2^{100} and 2^{200} are highly sufficient even for large-scale simulations. MRG32k3a has been designed to produce independent streams and sub-streams from its original random sequence, thanks to its parameters that enable safe Sequence Splitting. Thus, the internal parameters split the initial sequence into 2^{64} adjacent streams of 2^{127} random numbers, themselves divided into sub-streams containing 2^{76} elements.

The ability to issue independent streams is very important when tackling the safe distribution of random numbers across parallel computational elements. The Sequence Splitting approach at the heart of MRG32k3a suggests an obvious partition of the original sequence by assigning each computational element a stream or a sub-stream, depending on the application eagerness for random numbers. As long as we are focusing on parallel applications that are Java threads based, the parallel grain is limited to how many threads a single manycore machine can handle. This figure depends on the

underlying architecture hosting the Java platform, but we do not expect having to deal with more than 2^{64} parallel threads in the near future, which is the total number of independent streams bearing 2^{127} random numbers that each MRG32k3a can provide.

The last important point in our opinion is that this generator displays a great statistical quality, according to its TestU01 results related in [L'Ecuyer and Simard, 2007]. MRG32k3a passes all the tests of BigCrush, the most stringent and complete testing battery coming with TestU01, and is so referred to as a "Crush-resistant" PRNG in [Salmon et al., 2011]. While being Crush-resistant cannot ensure a perfect randomness of the considered pseudorandom stream, it is a satisfying property of which few PRNGs can be proud. Furthermore, PRNGs stated as bad according to TestU01 criteria have led to incorrect simulation results in the past [De Matteis and Pagnutti, 1988; Ferrenberg et al., 1992; Maigne et al., 2004], and even good PRNGs can miss some tests [Reuillon, 2008b; Salmon et al., 2011]. Thus, as we did not want to take any risks with our PRNG choice as a replacement of the LCG of *ThreadLocalRandom*, we focused on Crush-resistant PRNGs such as MRG32k3a.

4.3.4.2 Implementation Details

We have designed ThreadLocalMRG32k3a so that it can be used as an alternative to *ThreadLocalRandom*. Thus it displays the very same interface as its counterpart. The methods contained in our class are all dedicated to produce various kinds of random outputs: from integers to double precision floating point values, so as *ThreadLocalRandom* performs. In the same way, we also reused the *current()* method introduced in *ThreadLocalRandom*: it actually aims to provide its independent instance of ThreadLocalMRG32k3a to each thread calling it. The *current()* method is the core of ThreadLocalMRG32k3a in a sense that the call hierarchy it implies highly differs from the original behaviour of *ThreadLocalRandom*.

Every thread must call the static method *current()* in order to retrieve its own ThreadLocalMRG32k3a instance. The method basically acts like a singleton that builds a *ThreadLocal* instance parameterized with the PRNG class, ThreadLocalMRG32k3a in our case. *ThreadLocal* is a generic Java class appeared in JDK 2 that provides easy copy-on-access facilities to concurrent threads. When a thread first accesses a *ThreadLocal* object, the latter gets an instance especially built for it that does not require synchronized accesses with other threads. Typical applications of this mechanism are thread-based counters such as thread identifiers for example.

Our implementation first takes advantage of this technique to ensure reproducibility

between executions. Let us recall that stochastic simulations need to be reproduced for debug purposes or their results to be checked. When *ThreadLocalRandom* is used by default, it does not satisfy this need because it relies on the constructor of *Random* to set its internal seed. As this default constructor uses the current system time as seed, we changed its behaviour in basing the seed initialization on the thread unique identifier. Then, for a given identifier, a thread will always be assigned the same stochastic stream. Although Java enables us to figure out a thread identifier at runtime through the *Thread.currentThread().getId()* call, we cannot rely on this identifier since it is global for the whole JVM. Because of that, the identifier issued by this call depends on the number of threads created prior to the callee. Therefore, we have stored a handcrafted unique identifier within ThreadLocalMRG32k3a, thanks to a synchronized atomic counter handled through the *ThreadLocal* mechanism.

Please note that the *ThreadLocal* mechanism only operates at thread level and is not aware of any task concept introduced by top-level frameworks such as Fork/Join or Executors. Thus, reproducibility cannot be expected when either *ThreadLocalRandom* or ThreadLocalMRG32k3a is used by tasks from these frameworks.

Now that we are able to assign a stream to each thread, we need to determine how these streams are actually handled within our MRG32k3a implementation. We have seen previously that this PRNG had been designed to partition its original sequence into streams and sub-streams. We have chosen to give an independent stream to each thread, so that they can all benefit of their own independent 2^{127} numbers long pseudorandom sequence. As long as streams are contiguous in the original sequence, the beginning state of each independent stream is located every 2^{127} elements in the original sequence. Hopefully, a Jump Ahead algorithm is detailed in [L'Ecuyer, 1999] that enables us to advance the state of the original sequence at almost no extra cost, no matter how much elements we skip. Thus, if a thread has been assigned an identifier k, the Seed Status of its ThreadLocalMRG32k3a instance is initialized by the constructor to X_n, with $n = 2^{127} * k$. The latter situation is summed up in Figure 4.4.

4.3.4.3 Example of use

From a Java developer point of view, picking up random numbers from ThreadLocalMRG32k3a is as simple as using the original Java Random API as exposed in Listing 4.6.

```
1 ThreadLocalMRG32k3a myRNG = ThreadLocalMRG32k3a.current();
2
3 Thread tid = new Runnable {
4     public void run() {
5         for (int i = 0; i < numbersCount_; ++i) {
6             myRNG.next();
7         }
8     }
9 }
```

Listing 4.6: Example of use of ThreadLocalMRG32k3a

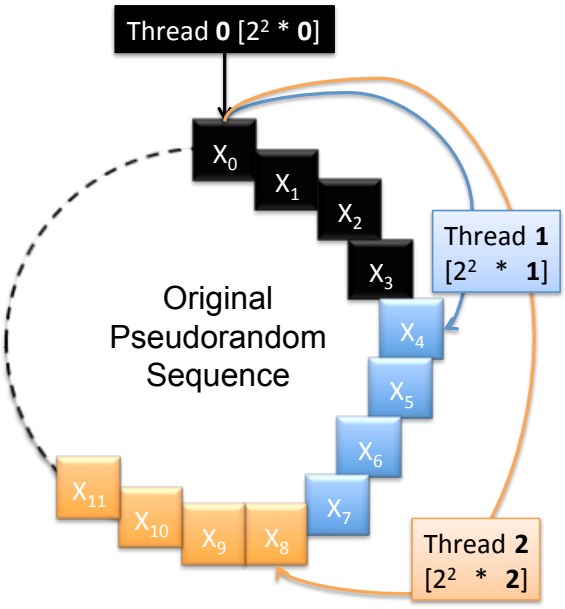


Figure 4.4: 3 Threads performing respective Jump Ahead on an original pseudorandom sequence, according to their unique identifier. Streams are here limited to 2^2 elements each.

The implementation detailed in this section makes ThreadLocalMRG32k3a the equivalent of *ThreadLocalRandom* concerning API and features. However, our proposal is more suited to parallelize scientific applications where statistically sound random sources are necessary. Since Section 4.4 will introduce an extension of ThreadLocalMRG32k3a, called TaskLocalRandom, we will delay the detailed performances analysis until Section 4.4.5.

4.3.5 Summary

This section has studied the recent *ThreadLocalRandom* proposal shipped with JDK 7 that intends to provide independent random streams for parallel Java applications. Having stressed the importance of using statistically sound PRNGs and partitioning techniques, we have asserted that Crush-resistant generators were in our opinion the only category of generators that should be trusted for scientific applications development. Considering this criterion, we have evaluated *ThreadLocalRandom*, as having a satisfying design but a poor implementation.

Meanwhile, this study surveys the most spread libraries that have the same purpose as *ThreadLocalRandom*, but display improved quality. We strongly recommend some of them, like SSJ or DistRNG, to replace *ThreadLocalRandom* as much as possible.

In addition, we propose in this work ThreadLocalMRG32k3a as another alternative to *ThreadLocalRandom*. Our proposal respects the same API as *ThreadLocalRandom*, but it relies on MRG32k3a, a well-known Crush-resistant PRNG. Not only does ThreadLocalMRG32k3a displays a far better statistical quality than its JDK counterpart, it is also much more suited for stochastic simulations, given that it issues a reproducible output by default.

As explained in Section 4.3.4.2, the implementation of ThreadLocalMRG32k3a does not take into account the notion of tasks introduced by some Java frameworks such as Fork/Join. Hence, Section 4.4 will introduce an extension of ThreadLocalMRG32k3a enabling it to behave correctly when used within Java tasks.

4.4 TaskLocalRandom

4.4.1 Introduction

In section 4.3, we have seen that Java 7 had introduced the *ThreadLocalRandom* class. This tool enables developers to deal with pseudorandom numbers in parallel on a single shared memory computer, without having to figure out how to distribute numbers among the available Processing Elements. The question for scientific applications is as follows: can *ThreadLocalRandom* serve as a random source with the statistical quality required by scientific applications?

As its name suggests, *ThreadLocalRandom* is designed to perform at a thread level.

However, threads are now mostly used as worker threads in Java thread pools, following the introduction of tasks frameworks since JDK 5. Worker threads were designed to get rid of the overhead bound to threads creation. An application creating lots of threads will be slowed down by frequent thread spawns. To overcome this issue, threads are created once and for all, and are then assigned tasks to process. Such permanent threads are gathered in thread pools over the lifetime of the application.

Due to architecture considerations, we usually use as many worker threads as there are available Processing Elements in the system. Processing Elements can take the shape of physical cores or threads depending on the underlying architecture that exploits the Java Virtual Machine. For instance, modern Intel CPUs integrate a feature called HyperThreading that enables a physical core to refine its parallelism capabilities by handling threads at the hardware level. In such a case, the number of Processing Elements will denote the number of physical threads. In order not to limit the parallelism granularity to this physical threads boundary, the notion of tasks has been introduced. Tasks are purely equivalent to Threads in terms of development, since they implement the same Java *Runnable* interface. They only differ from threads in that they are queued within worker threads, and thus being scheduled when their number is greater than the number of workers. This behaviour is depicted in Figure 4.5. Tasks scheduling allows designing a finely grained parallel algorithm that will scale up smoothly on platforms with the number of worker threads.

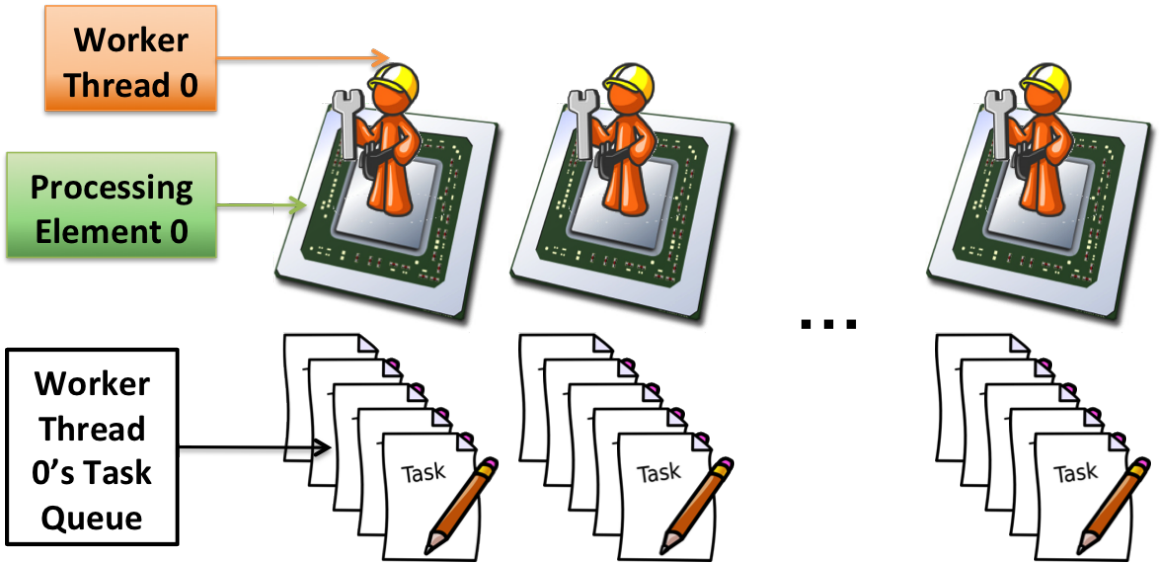


Figure 4.5: One worker thread per Processing Element is created. It is assigned a queue of tasks to process.

Tasks make pseudorandom stream distribution from *ThreadLocalRandom* inefficient, since tasks are not taken into account by the class. As a consequence, a novelty brought by the latest release of the JDK 7 cannot handle one of the most common way to leverage threads in Java concurrent applications! Although *ThreadLocalRandom* is stated as "particularly appropriate when multiple tasks (for example, each a *ForkJoinTask*) use random numbers in parallel in thread pools"[4], it cannot be considered as safe in terms of pseudorandom stream distribution since all the tasks run by the same worker thread will share the same pseudorandom stream.

The present work will consequently tackle the correct distribution of pseudorandom streams in parallel Java applications harnessing the power of tasks frameworks. To do so, we will:

- Discuss the capabilities of *ThreadLocalRandom* when used in a Task framework context;
- Introduce TaskLocalRandom, our proposal based upon the MRG32k3a Pseudorandom Number Generator (PRNG) algorithm from Pierre L'Ecuyer [L'Ecuyer, 1999];
- Compare TaskLocalRandom to *ThreadLocalRandom*, and consider its potential evolutions.

4.4.2 ThreadLocalRandom Plunged into Tasks Frameworks

Java tasks frameworks are now widely spread across Java applications exploiting concurrency. Introduced in JDK 5 through the *ExecutorService* class, these tools are thread pools that create threads for the whole lifetime of an application. These threads are then used as workers that will pick up tasks from queues created by the task framework and execute their content, instead of creating new threads. By doing so, the application no longer suffers from the overhead induced by frequent thread creations. The power of this approach is that it relieves developers from low-level thread management without impacting the application or requesting new knowledge.

The tasks queued to be executed by worker threads are nothing more than instances implementing the *Runnable* interface. This latter interface is already used to implement handcrafted concurrent Java applications: they contain the workload to be performed

[4] `http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ThreadLocalRandom.html`, *last access 7/28/13*

by threads when they are used without any task framework. This simplicity explains the wide adoption of tasks frameworks amongst the Java community.

However, the internal mechanisms of *ThreadLocalRandom* make it unable to handle tasks. Most of the features provided by *ThreadLocalRandom* to distribute pseudorandom streams amongst threads lie behind the *current()* method, which builds a *ThreadLocal* instance parameterized with the PRNG class, as we have seen in Section 4.3. The *ThreadLocal* mechanism only operates at thread level and is not aware of any task concept introduced by top-level frameworks such as Fork/Join. Thus, reproducibility cannot be expected when *ThreadLocalRandom* is used by tasks from these frameworks. We will describe our proposal to solve this matter in Section 4.4.4.1.

4.4.3 Related Works

None of the frameworks described in Section 4.3.3 integrate the task notion. They call for further developments if they are to be used within a task execution framework.

4.4.4 TaskLocalRandom Implementation

4.4.4.1 Assigning Independent Pseudorandom Streams to Different Tasks

Provided that we are able to uniquely identify tasks (this aspect will be tackled in Section 4.4.4.2), an independent pseudorandom sequence can be assigned to each of them. Similarly to the implementation of MRG32k3a, we have chosen to assign each task its own independent 2^{127} numbers long pseudorandom sequence. Again, this implementation relies on the Jump Ahead facilities intrinsically provided with MRG32k3a. Thus, if a task has been assigned an identifier k, the Seed Status of its TaskLocalRandom instance is initialized by the constructor to X_n with $n = 2^{127} * k$. This situation had been depicted up in Figure 4.4.

4.4.4.2 The Challenge of Uniquely Identifying Tasks

The main struggle at the heart of TaskLocalRandom is to provide a unique task identifier, which the Java language does not support at the time of writing. As explained previously, *ThreadLocalRandom* benefits of the *ThreadLocal* mechanism from the Java SDK. *ThreadLocal* relies on JNI (Java Native Interface) calls, which means its im-

plementation is directly tied to the underlying Operating System (OS) that supports the JVM. Each OS deals with threads on its own way, using native APIs. However, these native APIs do not provide a common concept of threads. It seems consequently difficult to implement a mechanism equivalent to *ThreadLocal* at a task level.

Two approaches can be considered to avoid this lack: either each task can autonomously distinguish itself from the others using a particular algorithm, or a central element in the system needs to uniquely identify each task.

In the first case, the most spread algorithm used to provide unique identifiers without the help of a central element is called UUID [Leach et al., 2005]. It was designed for the purpose of online Internet services and is now frequently exploited in programming techniques to distinguish elements. For instance, Java uses it to allot a unique version number to classes that support serialization. Several algorithms are referenced by the RFC (Request for Comments) standard to produce UUIDs. UUIDs issued by the *UUID* class from the Java SDK are from class 4. It means that the underlying algorithm used here is itself powered by a PRNG. The actual PRNG algorithm is not explicitly mentioned, but it is stated as cryptographically secure by the documentation. More pragmatically, a 128-bit identifier issued by *UUID* would only reach 50% of chance to overlap with another one if 1 billion of UUIDs had been picked up every second for 100 years. Consequently, this approach is reliable enough when it comes to generate unique random identifiers.

Still, UUIDs would directly represent the identifier of the task in our case. Let us recall that the latter identifier is also at the heart of the Jump Ahead algorithm of the MRG32k3a PRNG, which allows it to assign independent random streams to each task. Unfortunately, this Jump Ahead algorithm only accepts 32-bit integers in input to determine the amount of streams to jump over. In order to preserve the uniqueness characteristics of UUIDs, we cannot imagine to shrink them from 128-bit to 32-bit without introducing a risk of collision between two UUIDs. As a result, UUIDs are not a satisfying approach to uniquely identify tasks in our case.

The other option to achieve unique task identification is to request the identifiers atomically to a central element. This way to get identifiers has the drawback to create a bottleneck at the task creation, when each new task will claim its own identifier. Although this assertion is technically true, it is important to consider its impact in a more pragmatical way. To do so, let us figure out the typical number of tasks that might be created at some point in an application.

Tasks are typically created prior to any execution launched in workers. Still, we

can imagine that tasks are spawned in parallel in order to fasten this initialization stage. Then, the maximum number of tasks created at a given time cannot exceed the number of worker threads leveraged by the application. We know that the number of workers created is bound to the number of physical threads available on the machine. Any greater number of worker threads would quickly make the performance of the application drop. Thus, the number of worker threads, and consequently the maximum number of tasks potentially created at a given time will remain in the range of the number of physical threads hosted by all the cores of a machine. At the time of writing, this number can grow up to hundreds of physical threads in the cutting-edge HPC hosts. In Java, the number of physical threads available is obtained through a call to *Runtime.getRuntime().availableProcessors()*.

That being said, we have studied the execution time of several numbers of sequential calls to the *getTaskId()* method of our own *Runnable* implementation. The number of calls were chosen to match the typical number of physical threads contained in nowadays systems, but also to extrapolate any potential leap ahead in the number of physical threads available in the future. Table 4.1 sums up the results of this small experiment, executed on an old Intel Core 2 Duo running at 2.8GHz.

Number of calls to *getTaskId()*	32	64	128	1,024	1,024,000
Execution time (ms)	0.00397	0.00773	0.01601	0.05129	0.10813

Table 4.1: Computation time of several sequential calls to the *getTaskId()* method

As results in Table 4.1 show, even a great number of calls to *getTaskId()* will not introduce an overhead in applications making use of TaskLocalRandom. Eventually, the potential synchronization that could appear when the first tasks are started will quickly vanish due to scheduling considerations. All the worker threads will thus scarcely request for a new task identifier at the very same time. In conclusion, the central-element approach is satisfying since it fulfils our needs without impacting the computation time of the application.

4.4.4.3 Implementation Details

We have designed TaskLocalRandom for it to be used as an alternative to *ThreadLocalRandom*. It displays the very same interface as its counterpart. The methods contained in our class can produce two kinds of random outputs: double precision floating

point values and integers. These are the two kinds of data types that are handled by the original MRG32k3a implementation described in [L'Ecuyer et al., 2002a]. Double precision numbers are natively produced by the algorithm, which manipulates 64-bit floating point values at its heart in order to take advantage of hardware-implemented operations on modern CPUs.

In contrast with *ThreadLocalRandom*, TaskLocalRandom does not inherit from *java.util.Random*, which contains superfluous methods directly bound to the underlying LCG of this class. Still, methods in TaskLocalRandom respect the same interface than *java.util.Random* so that a minimal compatibility is maintained, without harming our design.

Although TaskLocalRandom might sound similar to the implementation from [L'Ecuyer et al., 2002a], only the interface is mimicked. Let us recall that TaskLocalRandom is task-aware. It can then be employed safely by users in order to produce highly independent pseudorandom sequences within the tasks of their Java parallel application.

The Java implementation of the central element described in Section 4.4.4.2 is achieved through a new abstract class called *RandomSafeRunnable*. This class implements the *Runnable* interface that is traditionally used to describe the behaviour of tasks and threads in Java concurrent applications. *RandomSafeRunnable* stores the identifier of the new task using a single instance of the class *AtomicInteger*, available in the *java.util.concurrent.atomic* package. After being initialized to 0 prior to any task creation, the constructor of *RandomSafeRunnable* performs a call to the thread-safe *getAndIncrement* method from the *AtomicInteger* object. The result of this call acts as the unique task identifier for the lifetime of the task represented by an instance of *RandomSafeRunnable*.

Please note that the way TaskLocalRandom is implemented also ensures that a new task will not keep the identifier of a formerly completed task. In such a case, tasks making use of *ThreadLocalRandom* would have been assigned the same identifier by the JVM. This would have led different tasks to exploit the same pseudorandom stream.

In order to concretely assign a unique pseudorandom sequence to each task, the Jump Ahead algorithm evoked in Section 4.4.4.1 is called with the identifier of the task as a parameter. As a result, each task now uniquely identified is assigned the stream corresponding to its identifier. Streams are labelled from the starting point of the original MRG32k3a pseudorandom stream.

4.4.4.4 Presentation of the API

From a Java developer point of view, picking up random numbers from TaskLocalRandom is as simple as using the original Java Random API as exposed in Listing 4.7.

```
1  public class Foo extends RandomSafeRunnable {
2
3    @Override
4    public void run() {
5
6      TaskLocalRandom rng = new TaskLocalRandom(this);
7
8      for (int i = 0; i < 5; ++i) {
9        System.out.println("Task[" + this.getTaskId() +
10         "] from Thread[" + Thread.currentThread().getId() +
11         "] {" + i + "} = " + rng.next());
12     }
13   }
14
15   public static void main(String[] args) {
16
17     ExecutorService executor = Executors.newFixedThreadPool(
18           Runtime.getRuntime().availableProcessors() );
19
20     for (int i = 0; i < 100; ++i) {
21       Runnable task = new Foo();
22
23       executor.execute(task);
24     }
25
26     executor.shutdown();
27     while (!executor.isTerminated()) {}
28   }
29 }
```

Listing 4.7: Presentation of the API of TaskLocalRandom

The implementation detailed in this section makes TaskLocalRandom the equivalent of *ThreadLocalRandom* in regards to its API and features. However, our proposal is more suited to parallelize scientific applications where statistically sound random sources are necessary, and it also fulfils the requirements needed by Java tasks frameworks. That being said, let us compare the performances of *TheadLocalRandom* and TaskLocalRandom.

4.4.5 Results

In this part, we compare three aspects of the initial *ThreadLocalRandom* with our proposal TaskLocalRandom: their memory footprint, their numbers throughput and their statistical quality. Then, TaskLocalRandom is faced with the other software tools of the literature introduced in Section 4.3.3. Please note that since TaskLocalRandom is an extension of ThreadLocalMRG32k3a, the performances of the former also characterize those of the latter. These two developments share common parts, like the PRNG algorithm for example. They mainly vary from their capacity to handle tasks.

4.4.5.1 Memory Footprint and Speed

ThreadLocalRandom wraps a LCG that uses only one integer to store its internal state, whereas MRG32k3a needs at least 6 integers. TaskLocalRandom also relies on an extra task identifier to provide reproducibility as required by stochastic simulations. Thus, *ThreadLocalRandom* is more efficient in terms of memory footprint.

Considering speed, it is hard to isolate accurately the methods involved in random number generation across several threads. That is why we based our comparison on the data produced by the VisualVM[5] profiler to figure out which algorithm was the most efficient. These results shows that TaskLocalRandom is about twice as fast as *ThreadLocalRandom*, requiring about 0.5 ms to pick up a random number whereas *ThreadLocalRandom* requires about 0.8 ms. Therefore, our Java wrapper does not impact the original fastness of the MRG32k3a algorithm. MRG32k3a is actually announced faster than the LCG used by *ThreadLocalRandom* in [L'Ecuyer, 1999].

[5] `http://visualvm.java.net/`

4.4.5.2 Statistical Quality

We have already discussed the statistical quality of LCGs, but in our case, the LCG at the heart of *ThreadLocalRandom* is used in parallel thanks to the Random Spacing distribution technique. When parallelizing an application, data processing is spread among the available computational elements following a particular pattern: the whole range of input data will be regularly sliced to feed each computational element. This configuration is also encountered for pseudorandom numbers: each thread or task receives its own pseudorandom stream and uses it to process its part of the data. The data of the corresponding sequential process would be equivalent to a concatenation of all the data chunks, but also of the pseudorandom streams used. As a result, knowing the parallelization techniques used for both random numbers and input data, we could recreate the computation scenario that would have taken place in a sequential environment. This allows us to check the corresponding random sequence resulting from the concatenation of the subsequences. Although two or more pseudorandom sequences considered independently can produce bad statistical results, their combination can behave differently when faced with the same statistical tests [L'Ecuyer, 1988].

We know that it is nearly impossible to examine every possible combination, thus we decided to focus on the most obvious technique to process input data: assign an equally sized subset from the original data to each task. This situation is sketched in Figure 4.6. Please note that for the purpose of this test, we fall back to standard Java threads so that *ThreadLocalRandom* can compete fairly with TaskLocalRandom. TaskLocalRandom can actually handle pseudorandom streams distribution across both threads and tasks, the latter being impossible for *ThreadLocalRandom*. Still, this parameter does not impact the results of our experience.

To simulate this situation, we have faced the two PRNGs to TestU01. The random stream studied by the testing battery was provided by combining the substreams of a given number of threads. In Table 4.2, each PRNG is tested using combined streams resulting from what would be the concatenated random sequence of 16 to 64 threads.

Table 4.2 shows that using MRG32k3a instead of the LCG implemented in *ThreadLocalRandom* is particularly relevant when considering the statistical output of both generators. Here, we see that none of the 180 configurations of *ThreadLocalRandom* tested can pass the TestU01 Bigcrush testing battery, whereas TaskLocalRandom does not generate any failed output. This figure backs our PRNG choice for the underlying algorithm of TaskLocalRandom.

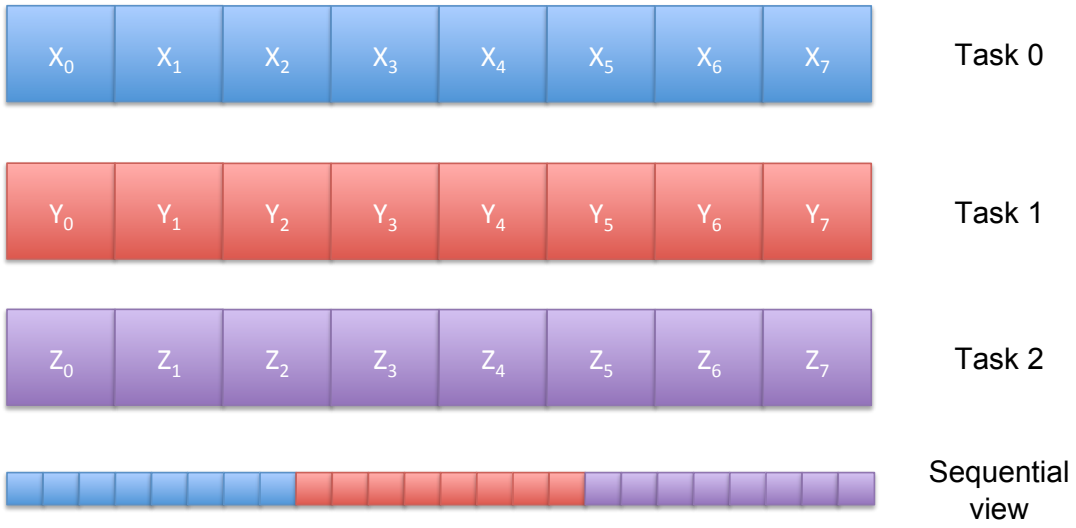


Figure 4.6: Substreams allotted to 3 different tasks and the corresponding pseudorandom sequence from the point of view of a sequential process

Class	16 threads	32 threads	64 threads
ThreadLocalRandom	all	all	all
TaskLocalRandom	none	none	none

Table 4.2: BigCrush failed results for *ThreadLocalRandom* and TaskLocalRandom used by 16, 32 and 64 threads. Each test configuration was initialized with 60 different seed-statuses

4.4.5.3 Comparison of Java PRNG Libraries

This section recalls the major characteristics of all the Java PRNG facilities that we described in Section 4.3.3. Table 4.3 aims at comparing these characteristics with those of TaskLocalRandom. We particularly focus on the ability of each library to automatically deal with pseudorandom streams distribution. The different criteria involved in the comparison are as follows:

- How many PRNG algorithms are embedded in the library?
- Does the library automatically handle pseudorandom streams distribution across threads?
- Does the library automatically handle pseudorandom streams distribution across tasks?

	Number of embedded PRNGs	**Automatic** distribution across threads	**Automatic** distribution across tasks
JAPARA	2	No	No
SSJ	11	No	No
DistRNG	9	No	No
ThreadLocalRandom	1	**Yes**	No
TaskLocalRandom	1	**Yes**	**Yes**

Table 4.3: Ability of Java PRNG facilities to deal with threads and tasks

As we can see in Table 4.3, TaskLocalRandom is the only library that automatically takes into account pseudorandom streams distribution at a task level, while the others would force the developer to determine a distribution scheme through the tasks of his concurrent application.

4.4.6 Discussion

In this Section, we proposed a Java implementation of L'Ecuyer's MRG32k3a that behaves correctly when used with Java tasks frameworks. However, simulation practitioners often expect to challenge their stochastic models with different random sources. In this way, providing a wider set of PRNGs is relevant for the simulation community. This complete framework would obviously display an API identical to TaskLocalRandom. In this section, we review the algorithms that we plan to include in future versions of this work.

Having already considered a Sequence Splitting partitioning technique with MRG32k3a, we chose to focus another highly reliable distribution technique: Parameterization (see Chapter 2.3.2). While Sequence Splitting intends to slice an original random sequence in several independent random streams, Parameterization tackles the problem differently. PRNGs employing Parameterization own a parameter that can distinguish one instance of a given PRNG from another. This unique parameter then contributes to issue highly independent random streams that can be assigned to different processing elements, such as tasks.

In view of the state-of-the-art PRNGs employing the Parameterization technique, mentioned in Chapter 2.4.6, we could embed TinyMT and Threefry/Philox in TaskLocalRandom.

In the case of TinyMT, part of this development work will be close to what has already been achieved with the implementation of MRG32k3a. However, this PRNG might show less flexible than MRG32k3a since its Parameterized Statuses need to be precomputed by the Dynamic Creator algorithm. DC relies on several C++ libraries, and would thus be difficult to reimplement in Java in a portable way. Thus, to provide a full Java concurrent implementation, not only we need to implement the algorithm, but also to ship precomputed statuses with it. The point is to find a tradeoff between a sufficient amount of Parameterized Statuses and a reasonable memory footprint, so that the sole PRNG does not bloats the whole application. Each task will then receive an instance of "TaskLocalTinyMT" initialized by a different status. Since the data structure representing a status weights no more than a hundred of bytes, delivering lots of ready to be used Parameterized Statuses should be possible.

Considering Salmon's algorithms Philox and Threefry [Salmon et al., 2011], they appear to be better suited than TinyMT (or any other member of the Mersenne Twister family) to target a smooth integration in Java tasks frameworks. Their parameters are formed by a single key that can be set at runtime according to each task's unique identifier. Thus, they could be useful when Mersenne Twister-like PRNGs might not fit some applications that cannot afford wasting any memory space to store the state and the initialization parameters of the PRNG of each task.

4.4.7 Summary

This work has studied the recent *ThreadLocalRandom* proposal shipped with JDK 7 to cope with Java task frameworks such as Executors and Fork/Join. Having evaluated *ThreadLocalRandom*, as having a satisfying design but a poor implementation in the previous Section, we have focused on its utilization within worker threads in this Section.

Undoubtedly, *ThreadLocalRandom* is intrinsically unable to deal with tasks executed within Java Thread Pools: it assigns the same pseudorandom stream to all the tasks handled by the same worker thread. We have detailed why this behaviour was obviously unsuitable when considering scientific applications.

As a result, we propose in this Section TaskLocalRandom as another alternative to *ThreadLocalRandom*. Our proposal respects the same API as *ThreadLocalRandom*, but it relies on MRG32k3a, a well-known Crush-resistant PRNG. TaskLocalRandom displays not only a far better statistical quality than its JDK counterpart but it is also much more suited for scientific applications, given that it issues a reproducible output

by default. TaskLocalRandom is a bit greedier than *ThreadLocalRandom* in terms of memory consumption, but it completely outperforms its counterpart in both speed and statistical quality. According to our measures, TaskLocalRandom is about twice as fast as *ThreadLocalRandom* and passes all the tests of BigCrush: the most stringent testing battery from TestU01.

The major input brought by TaskLocalRandom lies in its cooperation with the *RandomSafeRunnable* abstract class that we also introduced in this study. This pair of classes enables a correct distribution of pseudorandom streams among tasks. It is, to our knowledge, the sole PRNG facility that can be used safely within a Java task framework such as *Executors* or *Fork/Join*.

Among the simulation community, it is a safe practice to check the results of a stochastic simulation using several PRNGs which rely on different internal mechanisms. This is why we now plan to implement other Crush-resistant PRNG algorithms such as TinyMT, Threefry or Philox that display statistical properties equivalent to MRG32k3a. This effort would allow simulation practitioners to compare the results of their simulations when fed with different random sources. This way, simulationists could change the PRNG they use in the blink of an eye, and still benefit of correct pseudorandom streams distribution across their Java tasks.

4.5 Conclusion

In this chapter, we have implemented the guidelines from Chapter 3 in frameworks targeting CUDA-enabled GPUs and parallel Java applications.

ShoveRand, the CUDA-enabled development, is now fully functioning and offers features equivalent to what can be found in its counterparts: *cuRand* and *Thrust*. It deals with pseudorandom sequence partitioning across threads better than its counterparts. It integrates some of the PRNGs best suited for GPU platforms mentioned from Section 2.4.5 to 2.4.6.2. Now the question of its evolutions comes, and it still has several aspects that can be improved. For instance, the ability to ensure safe partitioning of pseudorandom sequences across multiple GPUs is a major concern in our opinion.

Regarding the Java libraries, ThreadLocalMRG32k3a and TaskLocalRandom both address issues spotted in the JDK 7. While *ThreadLocalRandom*, the original class from the JDK, displayed bugs preventing a correct distribution of the pseudorandom streams in its initial versions. Now that this problem is solved in the JDK, the LCG at the heart

of *ThreadLocalRandom* remains a weakness for any scientific application that would make use of this class. This is why, we believe our developments bring an input for such applications. Moreover, TaskLocalRandom is, to our knowledge the only library that takes Java tasks into consideration when it comes to safely distribute pseudorandom numbers. As an extension to ThreadLocalMRG32k3a, TaskLocalRandom will be the only one of our two Java tools to be maintained in the future.

Now changing hats as simulation practitioners instead of toolsmiths, the next Chapter will present a stochastic simulation models where we have used these tools and guidelines. The model studies how polymers fold in a cell and runs on CUDA-enabled GPUs. It is a stochastic model that uses the ShoveRand library described in this chapter.

Chapter 5

Application: Parallel Monte Carlo Simulation of a Polymer Folding Model on GPU

> “ *A leader leads by example not by force.*
>
> — Sun Tzu, *The Art of War*

Contents

5.1 Introduction

Two major design choices are available for code parallelization. One possibility is to adapt the existing application to fit a parallel template, which mainly consists in breaking loops to spread their workload across several parallel computing elements, be they threads or processors. This approach tends to be limited by the so called Amdahl's law [Amdahl, 1967; Hill and Marty, 2008] that computes the maximum speed-up of an application, taking into account the part of it that can effectively be parallelized. Another possibility consists in redesigning the application from scratch, to make the most of the target parallel platform. In this case, the application is more likely to benefit from performance gain laid down by the Gustafson's law [Gustafson, 1988]. Both situations present advantages: adapting an existing application will make it easier to write; redesigning an application from scratch often opens new parallelization perspectives.

When dealing with particular hardware accelerators such as GPUs, which are constrained by their underlying architecture, it is usually more efficient to provide a dedicated parallel implementation. Such an approach allows this implementation to better fit the requirements of the platform. In the case of GPUs, applications are expected to favour computing intensive algorithms rather than those containing a high number of branch instructions.

Parallelization of an existing code, even from scratch, can be severely limited by the sequential nature of the underlying algorithms, which have been originally thought for Turing-like architectures. In this context, it may be useful to redesign the algorithmic principles themselves, just as new algorithmic schemes have emerged from the possibility of quantum computing.

For the sake of software compatibility, simulations are often parallelized without much code rewriting. Performances can be further improved by optimizing codes to use the maximum power offered by parallel architectures. While this approach can provide some speed-up, performance of parallelized codes can be strongly limited *a priori* because traditional algorithms have been designed for sequential technologies.

Thus, additional increase of performance should ultimately rely on some redesign of algorithms.

In this chapter, we study a Polymer folding model that has been redesigned from an original sequential implementation to leverage the computing power of parallel architectures. We will present the associated Possible Futures Algorithm that has been specifically crafted to improve the results of this model. As a stochastic model running on GPU, this application will also show a concrete utilization of ShoveRand, our pseudorandom stream distribution library presented in Chapter 4.2.

5.2 A classical Monte-Carlo simulation of polymer models

The proper functioning of biological cells requires a specific organization, both in time and space, of genes that are expressed. The details of this organization are better known thanks to the advanced techniques of molecular biology. These techniques allow pinpointing the simultaneous location of several genes and also developing comprehensive lists of pairs of nearby genes in space [Lieberman-Aiden et al., 2009]. However, the multiplicity and diversity of the forces involved lead to a very partial understanding of the mechanisms responsible for the spatio-temporal organization. For this purpose, the modelling of chromosomes using unique polymer chains is very useful. The organization mechanisms are studied using the fundamental principles of polymer physics. The numerical implementation of the models enables us to study these phenomena *in silico*.

Numerical simulations are based on the stochastic evolution of monomers that make up the polymer chains. Three techniques are commonly used: molecular, Brownian and Monte Carlo dynamics. The latter allows to quickly reach thermodynamic equilibrium and is used to study the behaviour of large systems evolving in times typically-related to the cellular cycle (*i.e.* the minute/hour). Despite these advantages, the digital implementation of polymer models is inherently costly, especially for conditions of high confinement such as those in nuclei of cells. Algorithms based on conventional CPUs do not allow to simulate the behaviour of entire chromosome of eukaryote or prokaryote cells. The major problems are the relaxation time of the polymers that scale in the best case up to the square of the lengths of chromosomes, and the systematic rejection of certain movements during simulations because of collisions generated.

These approaches are highly parallelizable and fortunately a very significant gain in speed can be seen on parallel architectures with shared memory. A preliminary reflection led us to believe in the feasibility of an approach making several pieces of chromosomes evolve in parallel. The compatibility of these changes are regularly checked by testing collisions between monomers. There is currently no parallelization technique specific to polymers. However, the ability to massively parallelize the calculation of collisions during the simultaneous stochastic dynamics of different edges of the chromosome suggests that such an approach is viable.

The collaboration between our team and the MEGA (Modelling Genome Architecture and Engineering) team of the ISSB (Institute of Systemic Biology and Synthesis) aims at combining the skills at software integration and High Performance Computing present in our team, to those of the MEGA team at modelling the effects of spatio-temporal structure of genetic networks, in order to obtain performance gains. To do so, we chose to use GPUs prior to any implementation, so that this simulation model served as an application of our emerging works on GPU. In addition, collaboration between our two teams resulted in a funding by the CNRS to buy advanced equipments.

In this Section, we present the redesign of a traditional model of polymer folding [Junier et al., 2010], which has been extensively used in the field of chromosome modelling [Langowski and Heermann, 2007]. These models are known to suffer from poor efficiency when considered in situations that are similar to *in vivo* because of the strong confinement of chromosomes. So far, "simple" code parallelization has mainly been set up using tools such as OpenMP [Dagum and Menon, 1998] or MPI [MPI Forum, 1993]. More recently, parallelization of dynamical models have been achieved on GPU architectures, demonstrating the usefulness of such technology for the modeling of long polymers [Reith et al., 2011]. In the same spirit, our algorithm is mainly meant to exploit massively parallel processors, more particularly NVIDIA's CUDA-enabled GPUs.

Our new algorithm is based on a two-step procedure:

1. Generation of several potential future states;
2. Selection of the next state as a random choice amongst the set of valid futures.

We hence call this approach the "Possible Futures Algorithm" (PFA). PFA can be compared to the branch prediction feature available in modern CPUs, which enables CPUs to load the instructions following the branches of a conditional statement, prior to knowing the result of this statement. This avoids offloading the pipeline mechanism, and consequently waste precious instruction cycles [Lee and Smith, 1984; Smith, 1998].

While the initial algorithm only accepts $\sim 5\%$ of the proposed states in certain configurations, our solution maintains a satisfying acceptance rate over 60% in the same conditions, regardless of the input parameters of the model.

The description is organized as follows. First, we describe the initial chromosome folding model on which our parallel declination is based. Next, we point out the limitations that led us to design a new model. We then introduce our new parallel chromosome folding model and the PFA approach. Finally, we discuss some implementation choices and study the performances of the first implementation of our algorithm.

5.3 Description of the original model

Large-scale properties of polymers have been efficiently captured thanks to simplified models that ignore the details of both the polymer composition and the exact nature of the surrounding solvent [de Gennes, 1988]. In biology, various models have been used to model the behavior of DNA molecules and of chromosomes [Strick et al., 2003]. Chromosomes have been modeled as simple flexible chains that cannot overlap with themselves (self-avoidance effect) – see Figure 5.1. These so-called worm-like chains provide a coarse-grained description of protein-coated DNA that is simple enough so that, using Monte-Carlo simulations, they can be used to investigate the folding properties of rather long biomolecules [Langowski and Heermann, 2007]. Thus, they have been used to address both the problem of the structuration of chromosomes *in vivo* [Fudenberg and Mirny, 2012] and, more recently, the interplay between structuring and function [Junier et al., 2012].

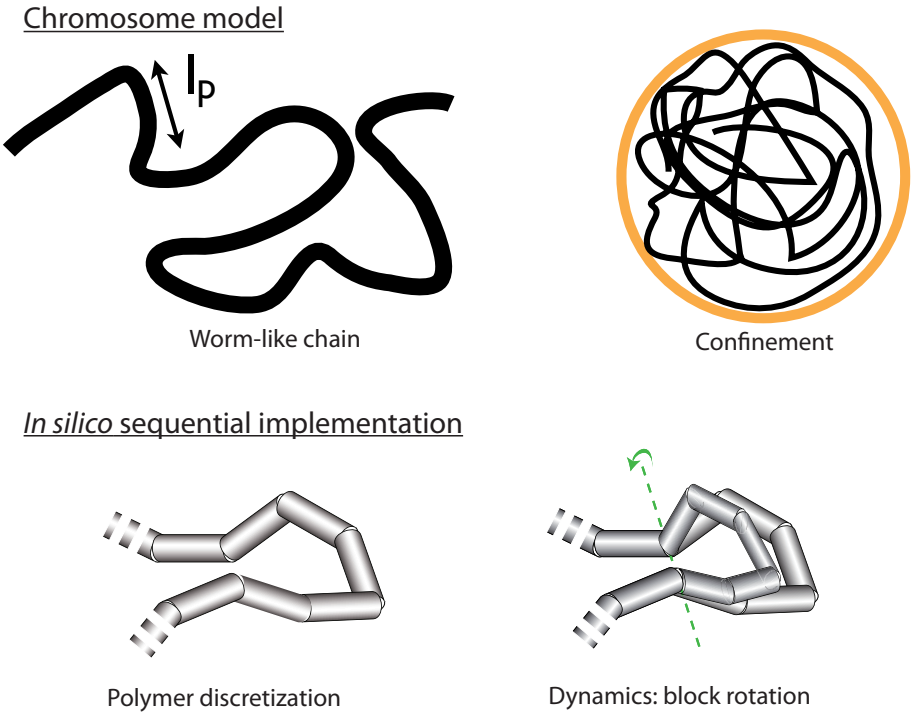


Figure 5.1: Worm-like chain model of chromosomes. On the top is indicated the original continuous model, which is characterized by the length beyond which the chromosome looses the memory of its directional order (persistence length l_p). The actual situation to be modelled is a situation where the chromosome is confined to a small volume (on the right). To be simulated on a computer, this model is discretized into cylinders (bottom). In this discretized version, two consecutive cylinders can rotate around their common joint but have to pay some energy to do so because as they are semi-flexible (captured by the persistence length). In the simulations, we use five cylinders per persistence length (figures adapted from [Junier et al., 2010]).

The worm-like chain is a continuous polymer model that needs to be discretized to be simulated on a computer. To this end, the polymer chain is divided into contiguous cylinders. Two consecutive cylinders are jointed together so that they can rotate freely around their joint (Figure 5.1), which is used to make the polymer conformations evolve dynamically. To mimic the persistence of the directional order of the polymer chain (see Figure 5.1), the junctions between each cylinder (hereafter called joints) carry an energy called "bending energy". This energy is exchanged with the solvent using a Monte-Carlo algorithm (see Algorithm 5.1) and depends on the angle between the two cylinders connecting the joint. We further consider chromosomes that are confined into cells that are more or less narrow, so that our approach can address the computational

properties of chromosomes in confined geometries [de Gennes, 1988]. In the following, we call "cell" the embedding volume of the chromosomes during the simulations, *i.e. in silico*. In the most general case, this may not correspond to the nucleus in which chromosomes are embedded *in vivo*.

In summary, a chromosome *in silico* can be viewed as a long string of contiguous "cylinders". The chromosome can be circular (as in bacteria) or linear (as in humans). Its statistical properties are the result of its jiggling motion due to the interaction with the surrounding solvent. In these models, the solvent is not taken explicitly. Instead, we used a standard stochastic algorithm, also called a Monte-Carlo procedure, to make the chain move step by step. Each of these steps is conceptually simple, and altogether, they form the informal Algorithm 5.1.

Select a section of consecutive cylinders of random size (called "block" hereafter);
Perform a rotation of the block by a random angle around the axis that intercepts the first and last joint of the block;
if *The rotated block does not overlap with the rest of the polymer* **then**
    Compute the new bending energy of the joints at the edges of the block;
    Accept the rotation with some probability depending on the variation of the bending energy;
Increase time +1;

Algorithm 5.1: Monte Carlo procedure involved in the simulation process

We used the standard Metropolis rate for accepting the rotation depending on its energy: *i.e.* if the new bending energy is lower, the transition is accepted; otherwise, the transition is accepted with the probability $e^{-\Delta E/k_B T}$ where ΔE stands for the increase of energy and $k_B T$ for the thermal energy characterizing the solvent (k_B is the Boltzmann constant and T is the temperature). This dynamics is known to satisfy the required properties to simulate the thermodynamic behaviour of polymer models. In practice, it favours movements whose energy variation is dictated by the value of the temperature T. As a result, the chromosome is prompted to a state of equilibrium between the energies of the conformations and the number of equivalent conformations at a given energy, also called entropy.

This stochastic algorithm slowly makes the chromosomes evolve following a combination of random and physical constraints, over hundreds of millions of runs. The limiting factor for this evolution happens when most rotations are rejected due to the overlapping conditions. This occurs very often when the embedding cell is small, as in biological situations. In particular, simulations of large chromosomes become ex-

tremely time-consuming, which makes this type of algorithms poorly efficient to tackle in detail the folding problems of multiple chromosomes *in vivo*.

5.4 Limitations of the sequential model

5.4.1 The collisions bottleneck

The original algorithm picks up a "block" at random on the chromosome, and randomly rotates it around an axis. This process is repeated all along the execution of the simulation. By definition, it is therefore sequential. As a consequence, for string confinement constraints, *i.e.* for small cell sizes, it is difficult for the algorithm to find valid configurations towards which to evolve. Thus, because valid configurations are hard to find through random attempts, a large number of (wasted) steps is required to provide a new valid configuration. In situation of string confinement, 95% of the generated rotations can produce collisions. It is consequently a major bottleneck that needed to be tackled in order for the model to scale up with chromosomes sizes. A first possibility would be to reduce the amplitude of the rotations. However, in this case, motions of the polymer are very small and both a large number of blocks and a large number of tries are necessary to thermalize the system (*i.e.* to make it visit all most likely conformations). Thus, in any case, the use of parallel motions can facilitate the dynamical evolution of the polymer.

5.4.2 The Possible Futures Algorithm (PFA)

We propose a parallel simulation model which considers two parallel optimizations. The first one consists in handling several blocks in parallel, instead of only one block as in the sequential algorithm. Processing blocks in parallel is possible in our case because it does not break any physical law at the heart of the model. In particular, it does not break the so-called ergodicity, *i.e.* the unbiased sampling of the configuration space. This new parallel rotation step is presented in Figure 5.2.

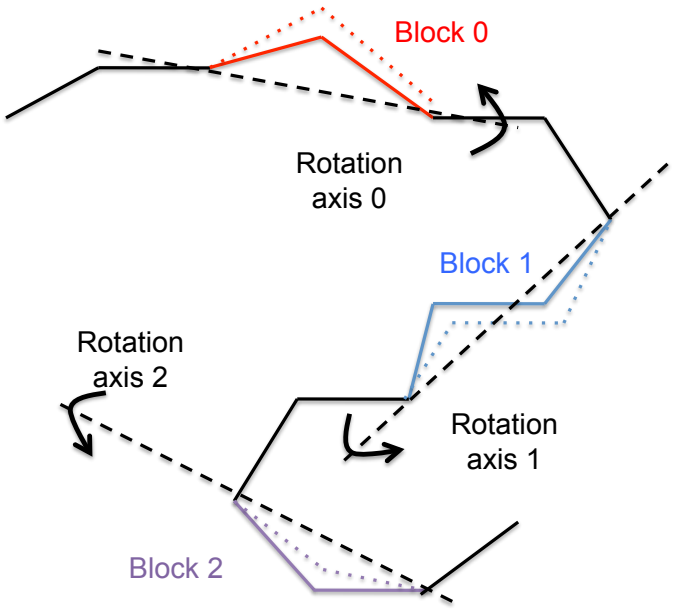


Figure 5.2: A parallel rotation step for 3 blocks

The second optimization consists in considering for each single block a set of several possible rotations to maximize the chances of producing a correct rotation. We call each of these rotations a "possible future" of a block. In Figure 5.3, we can see a chain of cylinders called "Base". It represents the chromosome as it is at the beginning of an iteration. Yellow cylinders belong to different blocks labelled *Block 0* through *Block 2*. Starting from blocks defined on the base chromosome, the algorithm will generate several possible futures for each block, applying each time a different rotation angle to the block. In Figure 5.3, three futures are generated for each of the three blocks defined.

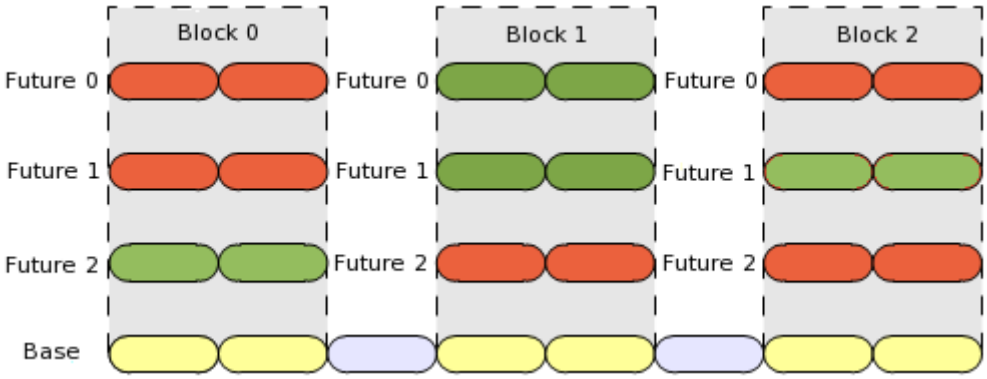


Figure 5.3: Example of generation of three possible futures for three different blocks

Cylinders in grey do not belong to any block, and therefore remain static during this iteration. It is important to note that at each step, the current state of a given block is also viewed as a possible future. Consequently, the fact of not rotating a block is valid, and is always a possible solution for the dynamical evolution of the polymer (which is here a no-motion).

5.5 Parallel model

The Possible Futures Algorithm induces extra computation time, since futures need to be created before each step of the simulation, and merged as a complete new state for the whole chromosome at the end of each simulation step. However, such an organization reminds of a frequently used pattern in parallel computing: map/reduce [Dean and Ghemawat, 2008]. This pattern is particularly adapted to GPU computing and some frameworks can even simplify its implementation on GPU [He et al., 2008]. In our case, the map stage consists in generating and processing new futures. This stage is processed in parallel, while the reduce stage is performed sequentially to build the resulting state of the chromosome at the end of the current simulation step.

We describe now the consecutive steps allowing the parallel algorithm to generate and transform possible futures, before combining them as a whole new state for the chromosome. In view of the massive number of repeated parallel tasks that result from the possible future approach, we have chosen to exploit GPU accelerators to run this algorithm.

5.5.1 Generate possible futures

First, the map stage must be initiated sequentially by generating random blocks in the chromosome. Blocks are equally sized arrays of consecutive joints from the chromosome. Each block owns a different, non-overlapping chain of joints. Blocks also differ from the random angle describing the rotation which will be applied to them. Blocks cannot overlap to prevent the same cylinder from being involved in two different rotations.

To generate futures, a “master” thread is run alone on the GPU to define random blocks and to generate random rotation angles. Then, a burst of threads is launched to clone the original blocks and change their rotation angles, thus changing them into possible futures of the original block. Upon completion, possible future blocks are

ready to be processed in parallel by thousands of GPU threads.

Calling F the number of possible futures per block and N the number of blocks, one can see that this stage operates in parallel the equivalent of what the sequential algorithm used to do over $F \times N$ time steps. The potential performance gain is hence on the order of $F \times N$, since while the sequential simulation checks after each step that the block can be moved, the parallel implementation fully transforms each possible future without wasting time on intermediate verifications.

5.5.2 Select valid futures

Once all the futures have been transformed and represent a potential future state of their base block, it is necessary to filter them: *i.e.* dismiss those that are not valid. The algorithm calculates the bending energy carried by the joints at the edges of a future block. To be stated as valid, the resulting bending energy of a future must be lower than the one of its original block. Should the new bending energy be greater than the original one, then the considered future has gained energy. In this case, there is a probability that the future may be valid, determined by a random draw according to the physical properties of chromosomes. In any other case, the physical properties of the future make it unsatisfying, and it is consequently withdrawn. The remaining futures must still meet two other requirements, referred to as static and dynamic conditions, to be compatible with their environment and to be potentially included in the global solution.

Static conditions are met when all the cylinders of the future are within the virtual cell in which the chromosome must remain confined. Moreover, none of the cylinders of the future can collide with parts of the chromosome that did not evolve during this simulation step. By doing so, we eliminate all futures that are not physically possible.

For dynamic conditions to be valid, a second evaluation determines which futures are suitable with respect to the other futures. A future is suitable as long as it does not collide with any of the other valid futures generated at this step of the simulation. Accepted futures are displayed in green in Figure 5.3, while dismissed futures appear in red.

To sum up, we have introduced extra operations in delaying the selection of valid futures, but this stage offers many potential solutions to increase the acceptance rate of rotations that make the chromosome evolve. This stage justifies on its own the use of a massively parallel architecture such as GPUs to speed-up the computation of the

algorithm. Let's now study the operations which compose the reduction stage of a simulation step.

5.5.3 Determine compatible futures

We consider a two-round reduction in our algorithm. The first round models the problem as a graph to determine the compatible futures. The vertices of the graph represent possible futures, and the presence of an edge between two vertices represents the compatibility between two possible futures.

5.5.4 Compose the global result

The second reduction step now aims at composing a global result from the compatible futures identified by the graph. From this graph, hundreds of random cliques are built from the whole set of possible futures at hand. A clique is a subset of vertices from a graph such that for every two vertices in the clique, there exists an edge connecting the two. In our case, a constant number of vertices is simply picked up at random in the graph to build equally-sized cliques, with the particular care to always draw a vertex from each group of possible futures.

We now compose a global result by finding one maximum clique from the set of cliques issued previously. A maximum clique owns as many possible block futures as there are blocks in the original chromosome. Thus, any maximum clique forms a solution to our problem and its nodes are the new state of each block defined at the beginning of the simulation step. Please note that should several maximum cliques be found during the first round of the reduction, one would be chosen at random, since they all represent equivalent satisfying solutions to make the chromosome evolve.

The elements from Figure 5.3 are displayed as a graph in Figure 5.4. It shows that the possible future 0 from Block 1 is suitable in terms of physics criteria, but is not compatible with any possible future from Block 2 (although this future is compatible with the current state of the chromosome, for the sake of simplicity, it is not shown in Figure 5.4).

In Figure 5.4, we see a clique compounded of 3 vertices: *{Block0.Future2, Block1.Future1, Block2.Future1}*. This means these three are inter-compatible futures, and together form a global consistent future for the chromosome. To validate the whole operation, each possible future block is stored in the base chromosome, replacing its original counterpart.

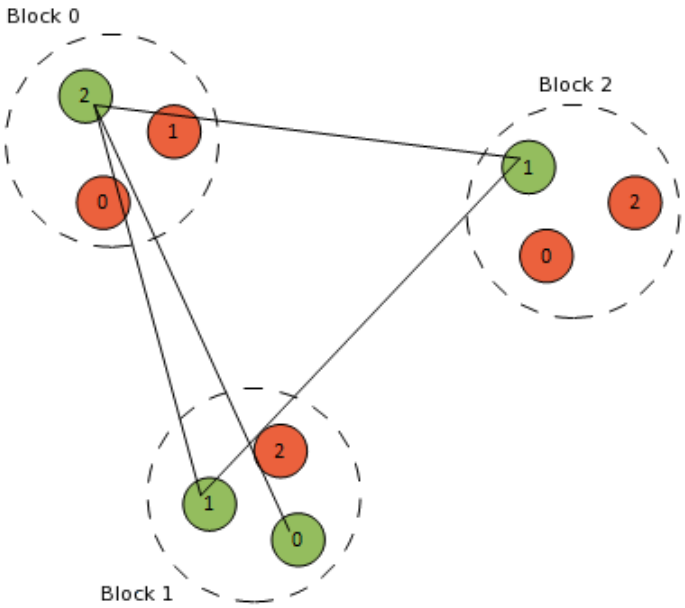


Figure 5.4: Compatibility graph of the possible futures

5.6 GPU implementation choices

5.6.1 GPUContext: avoid memory transfers between host and device

GPU applications often suffer from the excessive latency induced by memory accesses. It is strategic to optimize them as much as possible in order to obtain the best performance (memory transfers from the host to the device, or memory accesses from threads on the device) [Ryoo et al., 2008].

In our case, we have decided to cut any communication between the host and the device, except synchronizations. GPUs cannot have all their threads synchronized at one point, at the time of writing, due to hardware constraints. On the other hand, some stages of the simulation need the data to be in a particular state before they

can achieve their part of the computation. Although some research work claim to synchronize all the threads of a GPU [Feng and Xiao, 2010; Xiao and Feng, 2010], this is only true in restricted configuration where the device is under-utilized. Concretely, we know that blocks scheduling prevent a block to be preempted from a Streaming Multiprocessor (SM) before all its threads complete. Then, if all the threads of a block are implementing a synchronization barrier, waiting for the last thread of the whole kernel to reach it is impossible. What happens is that if all the SMs are occupied by blocks at a given time, they cannot be preempted so blocks waiting for execution will never be scheduled, since the running blocks will wait indefinitely for all the threads to complete. This situation usually freezes the device and consequently ends the execution abruptly.

Good practices of GPU programming suggest to use as much threads, and so blocks, as possible to keep the device busy. In doing so, it is very likely that the number of blocks created by a kernel will outnumber the number of SMs available. The only reliable way to synchronize all the threads on a GPU, regardless of the thread configuration of the kernel, is thus to wait for the completion of the kernel that has launched them. Each simulation step is consequently divided into several parallel stages, launched consecutively on the GPU. Such a design enables us to finely tune the parallelism grain of the simulation: at each stage the parallelism grain is the most adapted to the kernel that is to be launched.

As a result, communications are shrunk to their utter minimum from the host to the device. Pointers to the data transferred to the device memory are stored in an instance of the class *GPUContext*, which as passed as a parameters to each kernel. Simulation data are transferred back to the host from the GPU only when a checkpoint of the state of the simulation is reached.

5.6.2 Approximate cylinders to avoid branch divergence

In the original sequential model [Junier et al., 2010], the elements that compose the chromosomes are represented by cylinders with rounded edges, as shown in Figure 5.5. In this diagram, the point *mid* is the middle of the cylinder, and *e1* the direction vector that indicates the orientation of the cylinder. *rin* and *rfin* are two points which denote the edges of the cylinder.

This representation has the advantage of being accurate according to the observed physics, but on the other side, it highly complicates the calculation of collisions. Indeed, computing an intersection between two cylinders is rather slow: the corresponding

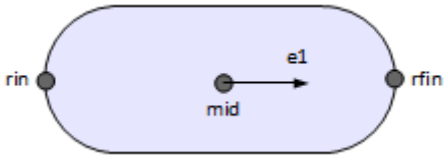


Figure 5.5: Cylindrical element that is part of the chromosome

code is about 200 lines and contains many conditions, which makes it less suitable for execution on GPUs.

Instead, cylinders can be approximated by groups of spheres, as illustrated in Figure 5.6. Points *sphere0*, *sphere1* and *sphere2* are the centres of the three spheres composing the cylinder. Points *joint0* and *joint1* are the edges of the cylinder.

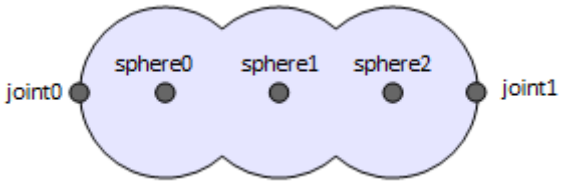


Figure 5.6: Pseudo-cylindrical approximation of a segment

Thanks to this representation, calculating the presence of a collision between two cylinders consists in calculating the collisions between the three spheres of the first cylinder and the three spheres of the second cylinder. Calculating collisions between two spheres is a trivial operation, as shown in Listing 5.1.

```
1
2  bool Sphere::collision (const Sphere & otherSphere)
3 {
4   Vector3 vector = this->position - otherSphere.position;
5
6   // All the spheres have a radius RADIUS.
7   // We compare the squared lengths to avoid an expensive square root.
8   return vector.squaredLength() - (RADIUS * RADIUS);
9 }
```

Listing 5.1: Calculation of collisions between spheres

Three spheres (rather than two or four) were chosen because it allows a very good approximation of the original cylinders and collisions observed. Moreover, using more

than three spheres does not increase the accuracy of the approximation in a significant way.

5.6.3 Pseudorandom number generation

Our new Polymer Folding model makes a full use of the facilities of pseudorandom stream distribution provided by ShoveRand. We will present the three steps needed to get pseudorandom numbers within the parallel kernels of the simulation.

The first thing to do is to set the PRNG algorithm to be used all along the simulation. This has to be done once and for all in the code. Listing 5.2 shows MRG32k3a is set as the generator of the simulation in the *GPUContext* class introduced in Section 5.6.1.

```
1 class GPUContext
2 {
3   public:
4
5   /** Random number generator type */
6   typedef shoverand::RNG<double, shoverand::MRG32k3a> rng_type;
7
8    ...
9 };
```

Listing 5.2: Configuration of ShoveRand to use MRG32k3a

Prior to any utilization of ShoveRand in a kernel, the library has to be initialized by describing the configuration of the kernel to be called in the *init()* method. As presented in Listing 5.3, the initialization step and the kernel launch are two consecutive operations. We only need to pass the number of blocks of the kernel to initialize ShoveRand properly. The call to *init()* is synchronous so that no extra call is needed between *init()* and the kernel launch to ensure the initialization phase is complete when the kernel starts.

```
1 void Simulation::step() {
2
3    ...
4
5   GPUContext::rng_type::init(blockCount);
```

```
6   Kernel::generateBlockFutures<<< blockCount, Constants::GPU::
        ThreadsPerBlock >>>(_deviceContext);
7
8   ...
9 }
```

Listing 5.3: Initialization of ShoveRand with the number of blocks to be used

Now that everything is up and running, picking pseudorandom numbers in the kernels only consists in calling the *next()* method in the code of the kernel. The mechanisms of ShoveRand work behind the scene to retrieve the pseudorandom stream assigned to the calling thread and issue the current pseudorandom number of this highly independent sequence. This process is depicted in Listing 5.4.

```
1  void Kernel::generateBlockFutures(GPUContext* context)
2  {
3    int uniqueId = threadIdx.x + blockIdx.x * blockDim.x;
4
5    ...
6
7    GPUContext::rng_type rng;
8    rotationAngle = (rng.next() * 2 - 1) * Constants::Simulation::
         MaxRotationAngle;
9
10   ...
11 }
```

Listing 5.4: Pseudorandom number drawn through ShoveRand

5.7 Results

Thanks to the log files generated by the application, which contain the 3D coordinates of each joint, we can obtain a 3D representation of chromosomes using third-party software such as *gnuplot*. Figure 5.7 is the output drawing generated by *gnuplot* for a circular chromosome after 100 million iterations. Such a number of iterations will draw from 44.10^9 up to 98.10^{12} pseudorandom numbers all along the execution.

All the results presented in this section have been obtained through simulations run on machines equipped with the following configuration: Intel Core i7-2600k 3.40GHz

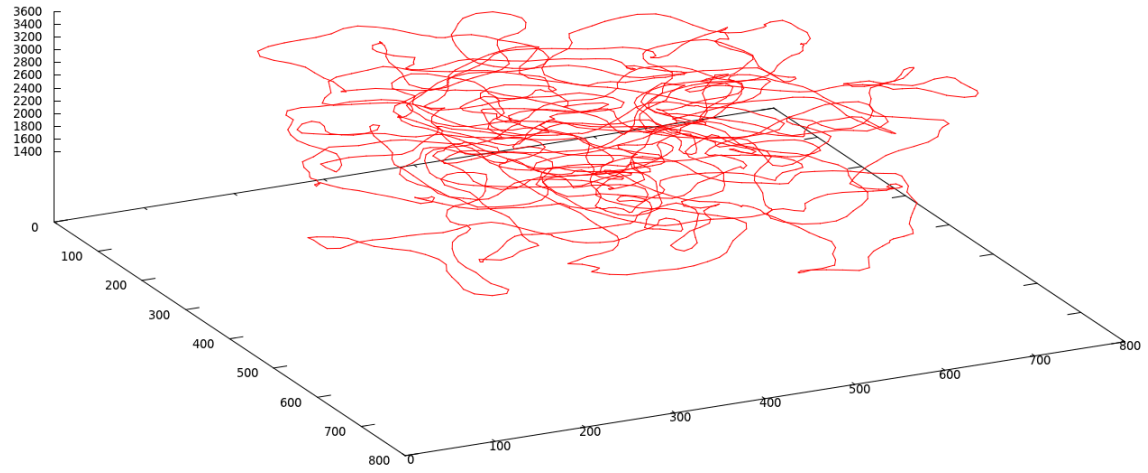


Figure 5.7: Gnuplot 3D representation of a chromosome

CPUs and NVIDIA GeForce GTX590 GPUs. These Fermi-architecture GPUs belonging to the GeForce range own 512 CUDA cores and are ECC-disabled on the contrary to the Tesla range. This configuration is not the best available in our cluster but it represents most of its machines composing. We wanted runs from the Design of Experiments (DoE) to be homogeneous and thus to be executed on the same kind of hosts. In Section 5.7.2, these performance will be compared with the cutting-edge K20c GPU.

5.7.1 Efficiency of the parallelization approach

The first metric that is interesting to study from the new parallel model is the efficiency of the possible futures approach regarding the acceptance rate. Figure 5.8, shows the efficiency of the PFA approach in increasing the acceptance rate. As we can see, the acceptance rate increases with the number of possible futures involved in each simulation step.

For clarity's sake, the acceptance rate depicted in Figure 5.8 is obtained by averaging the acceptance rates of several configurations benefiting successively from 4, 8, 64 and 128 possible futures per block. The configuration test set is as follows:

- number of blocks $\rightarrow \{1; 3; 7; 15\}$
- maximal block size $\rightarrow \{32; 64; 256; 510\}$

Eventually, it appears that the acceptance rate becomes trickier to improve when the number of blocks increases. Indeed, the number of cliques grows exponentially

with the number of blocks. Thus, it is more and more difficult to find a maximal clique amongst the set of randomly generated cliques.

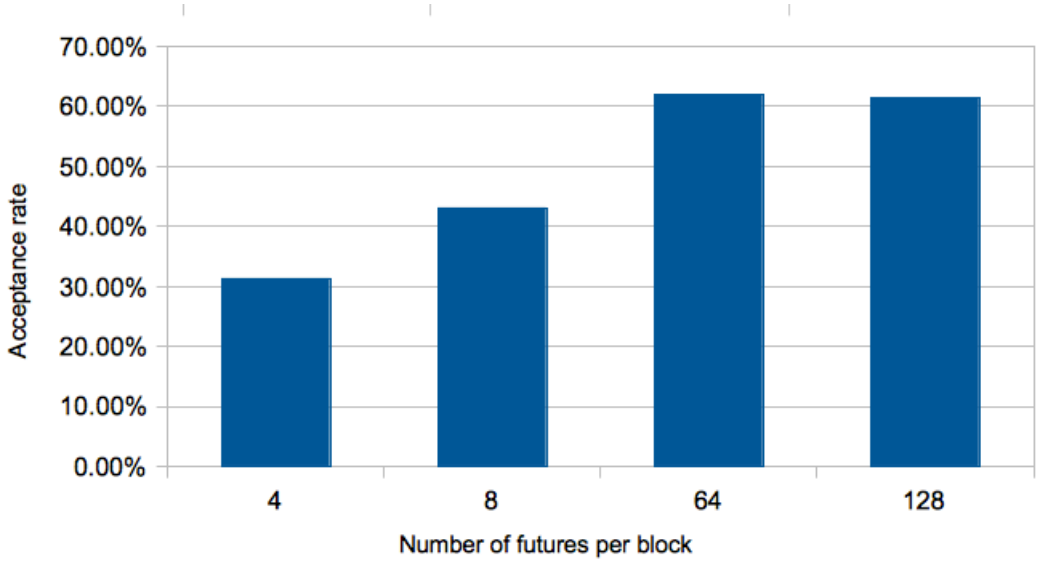


Figure 5.8: Increase in the acceptance rate with the number of possible futures

5.7.2 Performance on the cutting-edge Kepler architecture K20 GPU

Parallel architectures and especially GPUs evolve very quickly. In the particular case of NVIDIA GPUs, a new generation is shipped every two years. Thus, an application running on this architecture must take advantage of these evolutions and display improved performance when run on the forefront GPUs. Having run our benchmark on Fermi-architecture GTX 590 GPUs, we have tested some identical configurations on a recent Kepler-architecture K20 GPU [NVIDIA, 2012]. Figure 5.9 depicts the speed-up obtained at no cost, only by switching from the GTX to the K20 GPU.

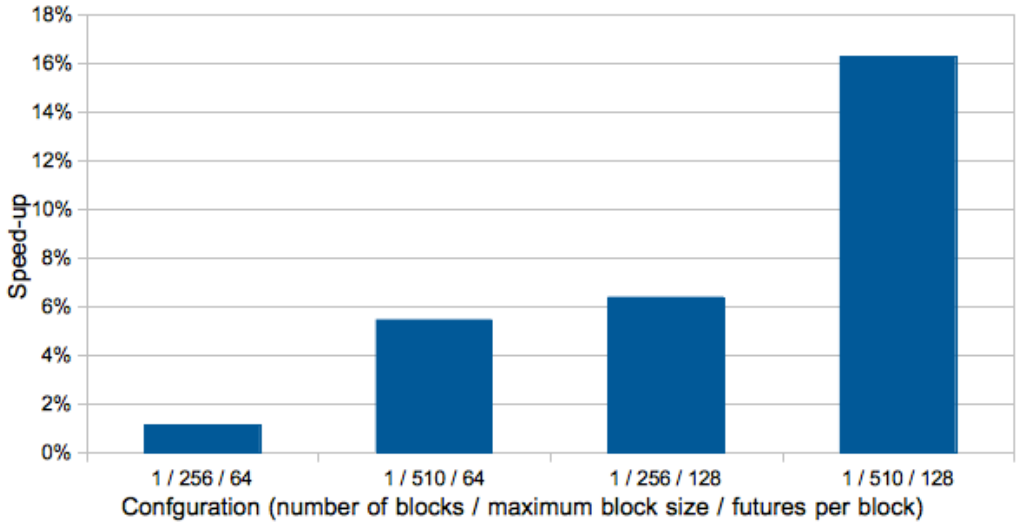


Figure 5.9: Improved performance of the model when run on a Kepler architecture K20 GPU

5.7.3 Impact of the ECC memory

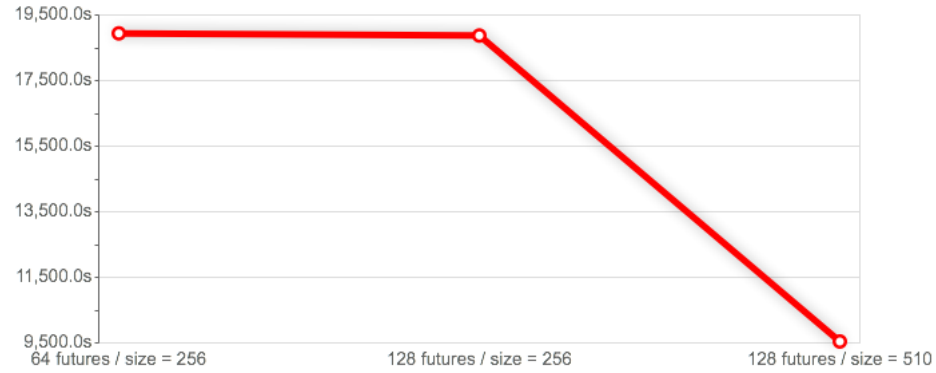
We have compared the performance of the three following configurations of the model, with a single block of cylinders:

- 64 futures per block, block of 256 cylinders
- 128 futures per block, block of 256 cylinders
- 128 futures per block, block of 510 cylinders

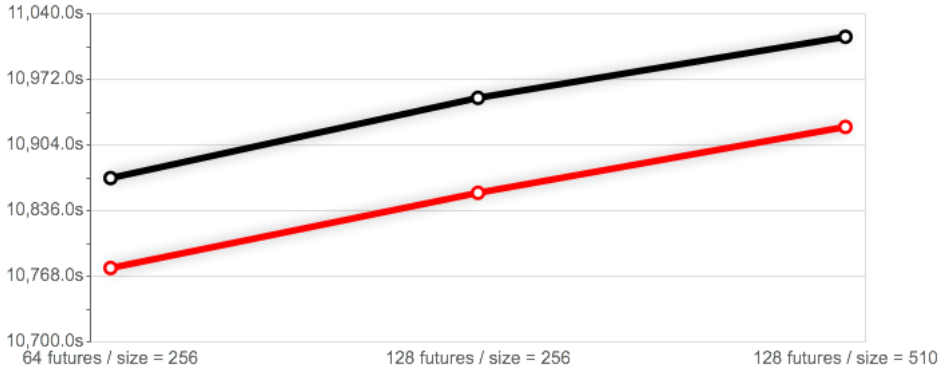
For this experience, we successively enabled the ECC memory on the device and the host, when it was available, to figure out the actual impact that this technology had on our application. The results that appear in Figure 5.10[1] were obtained through runs on 3 different GPUs: a Geforce GTX 590, a Tesla C2050 and a Tesla K20c.

The first thing to notice is that the non-scientific GeForce GTX 590 is clearly outperformed by its two counterparts from the Tesla range. It even produces a misleading value for the run with 128 futures per block of 510 cylinders: the execution time for this configuration is actually lower than the two others, but it is only because the compute capabilities of the device do not allow it to find a maximum clique among the too many possible future blocks created by this configuration. The model gets stuck

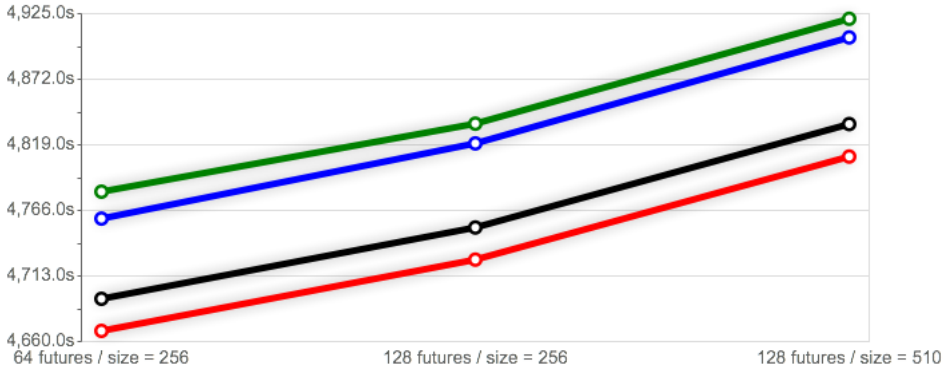
[1] *Figure generated using RGraph (`http://www.rgraph.net/`), under the terms of the Creative Commons 3.0 license.*



(a) GeForce GTX 590 (Fermi)



(b) Tesla C2050 (Fermi) - *Top (**black**) is ECC disabled on device, bottom (**red**) is ECC enabled*



(c) Tesla K20c (Kepler) - *From top to bottom: **Green** is ECC enabled on both host and device, **blue** is ECC enabled on host only, **black** is ECC enabled on device only and **red** is no ECC at all*

Figure 5.10: Comparison of the execution times of 3 configurations of the Polymer Folding Model on 3 different GPUs

and no new valid configuration is found, but in no way it is faster than the two other configurations.

Concerning the Tesla devices, as expected and already seen in Section 5.7.2, the Tesla K20c completes the simulation more than twice as fast as the Tesla C2050. The same shape is preserved between the two line charts, showing the ability of our CUDA implementation to scale from one generation of GPU to a new recent.

Finally, the impact of enabling ECC memory on the device is nearly insignificant as it only induces a less than 1% penalty on the execution time. Meanwhile, running the same configurations on the same device but on an ECC-enabled host is about 2% slower than on a non-ECC host. Then, our Polymer Folding Model is not sensitive to ECC memory, meaning it performs enough computations not to be slowed down by its memory accesses, or transfers to and from the host. These figures are noteworthy on the way to exascale computing, where hardware errors will become more and more common, while reliable technologies such as ECC memories will be a major concern in the design of High Performance Computing systems. Applications that are not memory-bound will appear more suitable to scale with these future hardware resources.

5.8 Conclusion

In this section, we have described a parallel model of chromosome folding and its implementation on GPU, using the NVIDIA CUDA technology. The interested reader will find a free and open-source version of both sequential and parallel implementations in the git repository of the French Institute of Complex Systems[2].

The purpose of this model is to make chromosomes evolve in the confined space of a cell, by applying successive rotations on random chunks of the chromosome. This new parallel model is based on an initial sequential one, which encounters difficulties when the cell space becomes more confined. The acceptance rate of the simulated rotations then quickly falls, making the sequential model evolve slower.

In order to increase the acceptance rate of simulated rotations, especially when the space becomes more and more confined, we designed a Possible Futures Algorithm (PFA). This approach proposes different new configurations for each block at each step, which have to be checked for inter-compatibility. All the futures can be generated and processed in parallel, thus inducing a massively parallel workload of repetitive tasks.

This approach has shown satisfying results in terms of acceptance rate. While the original model was likely to dismiss about 95% of the rotations, thus resulting in an

[2] `https://forge.iscpif.fr/projects/dna/repository`

acceptance rate as low as $\sim 5\%$, the Possible Futures Approach displays a satisfying acceptance rate of more than 60%. Such an acceptance rate enables the chromosome to evolve to a new configuration after almost each simulation step.

We now plan to challenge this parallel model with an implementation of the sequential one. Such a benchmark will have to be conceived very carefully, as long as the two models operate in two different ways and the same metrics cannot be easily compared between the two models. For instance, a simulation step does not perform the same actions in the two models. That is why we focused on the acceptance rate in this study. Still, the parallel implementation should display far better performance than the sequential one thanks to its Possible Futures Approach. The more the chromosome becomes confined within the cell space, the more the parallel algorithm will be efficient. In fact, the efficiency of the parallel implementation is due to its ability to maintain a high acceptance rate throughout the execution, whereas the sequential model will see its acceptance rate decrease with the various stages and the confinement.

While most of our results were obtained using Fermi architecture GPUs from NVIDIA, we highlighted improved performance on the cutting-edge Kepler architecture K20 GPUs. The next stage to push further the implementation and get improved performance would be to split the computation across several devices, in a multi-GPU approach. Still, it is difficult to forecast whether this technique will give a significant boost to the performance of our application. Other triggers can be pulled to improve the performance of a parallel application. We have seen that one thing to usually care for was the algorithm, that could be redesigned to work in parallel. This has already been the case with this model. However this step is time consuming, and the new implementation involves to choose a parallel platform along with a programming language to exploit it. Making a wrong choice at this step can lead to poor results in terms of speed up. Sometimes, a platform can slightly improve the performance, and thus hide the benefits of another one, more appropriate, that could have performed even better. In Chapter 6, we will study whether automatic parallelization techniques can help the simulation community build prototypes targeting various parallel platforms. The point is to obtain indications on the behaviour a given application would have on each of these potential hosts.

Chapter 6

Inputs of Automatic Parallelization Techniques for the Design of Parallel Simulations

> “ *For the most wild, yet most homely narrative which I am about to pen, I neither expect nor solicit belief.*
>
> — Edgar Allan Poe, *The Black Cat*

Contents

6.1 Introduction

At the manycore era, the keyword is parallelization. While designing parallel applications from scratch can be quite tricky, parallelizing sequential applications often involves a large refactoring. Moreover, the awaited speed-up is not magically obtained. Depending on the parallel-likeness of the application, results can be very disappointing. Then, the question is: how much time shall we invest on trying to parallelize a given application?

Depending on their underlying models and algorithms, applications will display greater speedups when parallelized on some architectures than others. For example, model relying on cellular automata algorithms are likely to scale smoothly on GPU devices or any other vector architecture [Caux et al., 2011; Topa and Młocek, 2012].

The problem is sometimes, developers champion a given architecture without trying to evaluate the potential benefits their applications could gain from other architectures. This can result in disappointing performances from the new parallel application and a speed-up far from the original expectations. Such hasty decisions can also lead to wrong conclusions where an underexploited architecture would display worse performance than the chosen one [Lee et al., 2010]. One of the main examples of this circle of influence are GPUs, which have been at the heart of many publications for the last 5 years. Parallel applications using this kind of platform are often proved relevant by comparing them to their sequential counterparts. Now, the question is: is it fair to compare the performances of an optimized GPU application to those of a single CPU core? Wouldn't we obtain far better performances trying to make the most of all the cores of a modern CPU?

On the other hand, many platforms are available today, and in view of the time and workload involved in the development of a parallel application for a given architecture, it is nearly impossible to invest much human resources in building several prototypes to determine which architecture will best support an application. As a matter of fact, parallel developers need programming facilities to help them build a reasonable set of parallel prototypes targeting a different parallel platform each.

This proposition implies that developers master many parallel programming technologies if they want to be able to develop a set of prototypes. Thus, programming

facilities must also help them to factor their codes as much as possible. The ideal paradigm in this case is Write Once, Run Anywhere, that suggests different parallel platforms understand the same binaries. To do so, a standard called OpenCL (Open Computing Language) was proposed by the Khronos group in 2008, and has lastly been updated to version 1.2 [Khronos, 2011]. OpenCL aims at unifying developments on various kinds of architectures like CPUs, GPUs and even FPGAs (Field Programmable Gate Arrays). It provides programming constructs based upon C99 to express the actual parallel code (called the kernel). They are enhanced by APIs (Application Programming Interface) used to control the device and the execution. OpenCL programs execution relies on specific drivers issued by the manufacturer of the hardware they run on. The point is OpenCL kernels are not compiled with the rest of the application, but on the fly at runtime. This allows specific tuning of the binary for the current platform.

OpenCL solves the aforementioned obstacles: as a cross-platform standard, it allows developing simulations once and for all for any supported architecture. It is also a great abstraction layer that lets clients concentrate on the parallelization of their algorithm, and leave the device specific mechanics to the driver. Still, OpenCL is not a silver bullet; designing parallel applications might still appear complicated to scientists from many domains. Indeed, OpenCL development can be a real hindrance to parallelization. We need high-level programming APIs to hide this complexity to scientists, while being able to automatically generate OpenCL kernels and initializations.

Another widespread cross-platform tool is Java and especially its virtual machine execution platform. The latter makes of Java-enabled solutions a perfect match for our needs, since any Java-based development is “Write Once, Run Anywhere”.

To sum up, OpenCL and Java both appear like relevant tools to quickly build a various set of parallel prototypes for a given application. This study therefore benchmarks solutions where one or the two of the these technologies were used to build prototypes of parallel simulations on manycore CPU and GPU architectures. We concentrate on simulations that can benefit from data-parallelism in this study. As a matter of fact, the tools that we have chosen are designed to deal with this kind of parallelism, as opposed to task-parallelism. We intend to show that automatically built parallel prototypes display the same trend in terms of speed-up than handcrafted parallel implementations.

6.2 High-Level APIs for OpenCL: Two Philosophies

Many attempts to generate parallel code from sequential constructs can be found in the literature. For the sole case of GPU programming with CUDA we can cite great tools like HMPP [Dolbeau et al., 2007], FCUDA [Papakonstantinou et al., 2009], Par4all [Amini et al., 2011] and Rootbeer [Pratt-Szeliga et al., 2012]. Other studies [Karimi et al., 2010], as well as our own experience, show that CUDA displays far better performance on NVIDIA boards than OpenCL, since it is precisely optimized by NVIDIA for their devices. However, automatically generated CUDA cannot benefit of the same tuning quality. That is why we rather consider OpenCL code generation instead of CUDA. The former having been designed as a cross-platform technology, it is consequently better suited for generic and automatic code production.

6.2.1 Ease OpenCL development through high-level APIs

6.2.1.1 Standard API C++ Wrapper

The OpenCL standard not only describes the C API that any implementation should meet, but also the C++ wrapper that comes along with it. This wrapper provides both C++ bindings of the OpenCL calls, and also two restricted declinations of the Standard Template Library (STL) from genuine C++: the string and vector classes. These two classes have been rewritten for the purpose of the OpenCL wrapper as a subset of their counterparts but are mostly compliant. The idea is to get rid of the bloated classes of the STL, which *std::string* is nothing but the best example. As long as they display an interface close to the original classes, the OpenCL versions, stored in the *cl* namespace, can be swapped with the matching STL class thanks to a single macro.

Another C++ mechanism nicely leveraged by the OpenCL C++ wrapper is exceptions. OpenCL errors are handled in a low-level way with error codes to be compared to 0 to assess whether the previous call is completed successfully. Obviously this technique is not quite adapted to develop with higher level languages, because it inflates source codes with non-effective lines. Moreover, it forces developers to be very careful and to explicitly check the returning code of their invocations, and, in case of problems, to retrieve the corresponding error. In our case, this painful process is handled by the exception mechanism that forces developers to catch the exception and treat it consequently.

The remaining elements of this binding are dedicated to ease the OpenCL API for developers. The most significant example in this way is the double call pattern foreseen in our short introduction to OpenCL in section 1.5.3.2. Using the traditional OpenCL C API, one needs to perform two successive calls to the same function to first figure out the number of results to be stored in an array provided for this purpose, before actually filling it through a second call requesting the exact number of elements to store in the array of results. The C++ wrapper here highly facilitates the process since a single call is needed to obtain the results, while *cl::vector* is used instead of dynamically allocated arrays.

Listing 6.1 shows an example of the syntax of the wrapper. Although efforts can be noticed to provide a simple API, the result remains quite verbose since Listing 6.1 only lists GPU devices and creates a dummy kernel from a source file in the context of the discovered device.

```
1  try {
2     // Place the GPU devices of the first platform into a context
3     cl::vector<cl::Platform> platforms;
4     cl::vector<cl::Device> devices;
5
6     cl::Platform::get(&platforms);
7     platforms[0].getDevices(CL_DEVICE_TYPE_GPU, &devices);
8     cl::Context context(devices);
9
10    // Create and build program
11    std::ifstream programFile("kernel.cl");
12    std::string programString(
13          std::istreambuf_iterator<char>(programFile),
14          (std::istreambuf_iterator<char>()));
15    cl::Program::Sources source(
16          1,
17          std::make_pair(programString.c_str(),
18          programString.length()+1));
19    cl::Program program(context, source);
20    program.build(devices);
21
22    // Add kernel to progam
23    cl::Kernel fooKernel(program, "foo");
24
```

```
25        // Create kernel
26        cl::vector<cl::Kernel> allKernels;
27        program.createKernels(&allKernels);
28    }
29    catch(cl::Error e) {
30        std::cerr << e.what() << std::endl;
31    }
```

Listing 6.1: GPU devices listing and kernel creation using the C++ wrapper API (adapted from [Scarpino, 2011])

6.2.1.2 QtOpenCL

Used to provide nice bindings for C++ development tools going from database drivers to concurrency, the Qt library developed by Nokia [Nokia, 2010] now also offers its own OpenCL wrapper as a set of extensions named QtOpenCL. At the time of writing, we would like to insist on the fact that QtOpenCL is neither included in the stable Qt 4.7 release, nor in the future 4.8 version according to Nokia's roadmap. However, QtOpenCL is freely available for download as an extension and is compatible with Qt 4.6.2 and Qt 4.7 frameworks.

As an OpenCL wrapper, QtOpenCL aims to break down three main barriers that we already identified as OpenCL drawbacks: the initialization phase, the memory management and the execution of kernels. To assist users in the initialization phase, and particularly to compile the external OpenCL source code, QtOpenCL handles automatically OpenCL source files reading through a single method. Memory management receives as much consideration so that buffers can be created more simply. Moreover, their object oriented interface allows users to call methods such as *read()* directly on buffers to retrieve results from a hardware accelerator.

The third point of the QtOpenCL intentions targets kernel executions. This aspect is handled "à la Qt" thanks to the *QFuture* class, well-known by developers using Qt for their CPU developments involving concurrency.

QFuture is an event returned when a kernel is run through the *QtConcurrent::run* method, and allows to wait for the kernel to complete. In addition, *QFuture* is compatible with the *QFutureWatcher* class that uses the signal/slot mechanism which Qt is based upon. The latter mechanism is an implementation of the Observer design pattern [Gamma et al., 1995] that causes a routine to be called when a particular event

occurs. In our case, the event signals the completion of the associated kernel and can rise an update of the Graphical User Interface (GUI) of the application for example.

Listing 6.2 presents how to enqueue a dummy kernel that takes a vector of integers as a parameter and outputs another one, using QtOpenCL.

```
1  // context creation
2  QCLContext context;
3  if ( !context.create() ) {
4          std::cerr << "Error in context creation for the GPU" << std::
               endl;
5          return 1;
6  }
7
8  const int vectorSize = 1024000;
9  QCLVector<int> inVector = context.createVector<int> ( vectorSize );
10 QCLVector<int> outVector = context.createVector<int> ( vectorSize );
11
12 for (int i = 0; i < vectorSize; ++i) {
13         inVector[i] = i;
14 }
15
16 // kernel build
17 QCLProgram program = context.buildProgramFromSourceFile ( "./kernel.cl
       " );
18 QCLKernel kernel = program.createKernel ( "foo" );
19
20 // enqueue and run kernel
21 kernel.setGlobalWorkSize ( vectorSize );
22 kernel ( inVector, outVector );
```

Listing 6.2: GPU context creation kernel enqueueing using QtOpenCL

6.2.1.3 PyOpenCL

PyOpenCL is a research initiative developed at the same time as its CUDA counterpart PyCUDA [Klöckner et al., 2012]. Both approaches rely on a concept called GPU Run Time Code Generation (RTCG) that intends to solve common problems encountered when harnessing hardware accelerators. In PyOpenCL, this translates in a flexible code

generation mechanism that can adapt to new requirements transparently. In the end, programmers can summon kernels as if they were methods from the program instance they have just built.

Apart from those dynamic features, PyOpenCL displays the classical inputs from an OpenCL wrapper developed in a high level language such as Python. This language is well-known for its concision and simplicity, and PyOpenCL directly benefits from this characteristic. This is achieved widely because of Python dynamic typing system that delegates an important part of the work to the interpreter. Finally, OpenCL source code reading takes advantage of the capacity of Python to handle files, so that a program can be read from source and built using no more than four lines, as it can be seen in Listing 6.3.

```
1 programFile = open ( 'foo.cl', 'r' )
2 programText = programFile.read ( )
3 program = cl.Program ( context, programText )
4 program.build ( )
```

Listing 6.3: GPU program building using PyOpenCL

6.2.2 Generating OpenCL source code from high-level APIs

6.2.2.1 ScalaCL

ScalaCL is a project, part of the free and open-source Nativelibs4java initiative, led by Olivier Chafik [Chafik, 2011b]. The Nativelibs4java project is an ambitious bundle of libraries trying to allow users to take advantage of various native binaries in a Java environment.

From ScalaCL itself, two projects have recently emerged. The first one is named Scalaxy. It is a plugin for the Scala compiler that intends to optimize Scala code at compile time. Indeed, the functional constructs of Scala might run slower than their classical Java equivalents. Scalaxy deals with this problem by pre-processing Scala code to replace some calls by more efficient ones. Simply, this plugin intends to transform Scala loop-like calls such as map or foreach by their while loops equivalents. The main advantage of this tool is that it is applicable to any Scala code, without relying on any hardware.

The second element resulting from this fork is the ScalaCL collections. It consists in a set of collections that support a restricted amount of Scala functions. However, these

functions can be mapped at compile time to their OpenCL equivalents. A compiler plugin dedicated to OpenCL generation is called at compile time to normalize the code of the closure applied to the collection. Functional programming usually defines a closure as an anonymous function embedded in the body of another function. A closure can also access the variables from the calling host function. The resulting source is then converted to an OpenCL kernel. At runtime, another part of ScalaCL comes into play, since the rest of the OpenCL code, like the initializations, is coupled to the previously generated kernel to form the whole parallel application. The body of the closure will be computed by an OpenCL Processing Element (PE), which can be a thread or a core depending on the host where the program is being run. Listing 6.4 presents a simple closure that computes the cosine function to every element of an array. Only the execution of the body of the closure is deported to the hardware accelerator targeted by OpenCL.

```
1  import scalacl._
2  import scala.math._
3
4  implicit val context = new ScalaCLContext
5
6  val r = (0 to 1000000).cl
7  val rangeArray = r.toCLArray
8
9  // Runs asynchronously on the GPU via OpenCL
10 val mapResult = rangeArray.map ( v => cos(v).toFloat )
```

Listing 6.4: Computing cosine of the 1,000,000 first integers through ScalaCL

The obvious asset of ScalaCL is its ability to generate OpenCL at compile time. It means that we could enhance ScalaCL by adding an extra step to tune the issued OpenCL kernel at compile time and take advantage of a GPU vendor specific extensions for instance.

6.2.2.2 Aparapi

Aparapi [AMD, 2011] is a project initiated by AMD and recently freed under an MIT-like open source licence. It intends to provide a way to perform OpenCL actions directly from a pure Java source code. This process involves no upstream translation step, and is then wholly carried out at runtime. To do so, Aparapi relies on a Java Native Interface (JNI) wrapper of the OpenCL API that hides complexity to developers.

Concretely, Aparapi proposes to implement the operations to be performed in parallel within a *Kernel* class. The kernel code takes place in an overridden *run()* abstract method of the Kernel class that sets the boundaries of the parallel computation. At the time of writing, only a subset of Java features are supported by Aparapi, it means that *run()* can only contain a restricted amount of features, data types, and mechanisms. Concretely, primitive data types, except *char*, are the sole data elements that can be manipulated within Aparapi's kernels.

For years, Java has been used to managing concurrency problems thanks to a *Thread* class whose implementation makes use of the native thread of the underlying Operating System where the Java Virtual Machine (JVM) runs. Thread implements the *Runnable* interface that only consists in providing a *void run()* method that will be called when the thread is launched. This mechanism is widely adopted in the Java world, and most of the frameworks offering an abstraction layer to *Thread* employ it in order to remain compliant between each other. At first, it seems that Aparapi follows the same path given that it designates a *run()* method to contain the parallel code. However, Aparapi's source code does not involve *Runnable* at any moment. A simple example of using Aparapi is set up in Listing 6.5 to square an array of 8 integers.

```
1  final int [] inArray = new int[] {1, 2, 3, 4, 5, 6, 7, 8};
2  final int [] outArray = new float[inArray.length];
3
4  Kernel kernel = new Kernel ( ) {
5          public void run() {
6          outArray [ getGlobalId() ] = inArray [ getGlobalId() ] * inArray [
               getGlobalId() ];
7          }
8  }
9
10 kernel.execute(inArray.length);
```

Listing 6.5: Squaring an array of integers using Aparapi

This design choice seems rather awkward when taking into consideration that the parallel tasks are assigned to a pool of CPU worker threads, the Java Thread Pool (JTP), when no accelerator can be found on the host platform. In fact, several implementations of a JTP are now shipped with Java Standard Development Kit (SDK) like Executors or Fork/Join, and have proved to be both efficient and user-friendly. Software design fosters reutilization as much as possible and being compliant with standard tools not only fastens development but it also strengthens it.

Additionally, it is interesting to note that Aparapi is not fully cross-platform, albeit being written in Java. The JNI bindings used to perform the calls to the OpenCL API make the Java package of Aparapi bound to native binaries. Thus, Aparapi cannot be shipped as a single package and depends on the ability of its underlying platform to run native code. This aspect can become a problem in terms of simplicity of use by clients, since it forces them to have a C++ tool-chain installed on each platform they want to run Aparapi on, so that the native part of the library can be recompiled for their particular platform. Still, native code dependency is not a major drawback for Aparapi given that there are few chances that a platform cannot execute the compiled-side of the JNI part of the library. This means that Aparapi can mostly be considered as a cross-platform tool that requires little effort from the client to be effective.

The first versions of Aparapi came with a slight limitation: AMD constrained the library to run on its devices only! Systems where AMD devices could not be found used to fall back to the JTP instead. Thanks to the open-source licence that protects this tool, a third-party developer was able to remove that latter constraint. In fact, execution is locked on a particular set of devices, here AMD GPUs, only by comparing the OpenCL platform identifier to the string identifying an AMD OpenCL platform... The removal of this software restriction makes Aparapi a potential high-level API to automatically build parallel declinations of applications on any OpenCL-enabled platform. The availability of the source code is also really helpful in view of some of the shabby implementation choices made by the original developer: for instance, Aparapi always targets the device bearing the identifier 0 in a system! If this aspect is frustrating when running Aparapi on a system owning multiple OpenCL-enabled devices, it becomes really annoying when the OpenCL compilation failed because the first device is not available (locked, or disabled).

6.2.3 A complete solution: JavaCL

JavaCL is an open source initiative that targets a smooth integration of OpenCL in Java applications. Like ScalaCL that we have studied in the previous section, JavaCL is part of the Nativelibs4java project developed by Olivier Chafik. The JavaCL library displays two interesting aspects that we will describe hereafter.

The first input from this library is to ease OpenCL development from the host side. In order to be easily integrated in Java code, the OpenCL 1.1 API [Khronos, 2011] has been fully wrapped in Java classes. Every element required to perform OpenCL computations can now be handled directly in Java. This allows us to write nothing

but the kernel using the C OpenCL API.

JavaCL is issued as a single *jar* file containing all the dependencies needed to be executed on any platform, included native libraries in charge of being the bridge between Java and the OpenCL platform. This way, it is perfectly independent from its underlying platform, and can be easily shipped by several ways. The most convenient packaging form is as a Maven repository in our opinion. Maven [Apache Software Foundation, 2002] is a build tool targeting Java applications that allows the projects it constructs to declare dependencies located in remote repositories. When a dependency is encountered for the first time, the Maven client installed on the host system building the project downloads the dependency, and stores it in a local repository, so that it can be reused for a further build.

Functions intending to query an OpenCL installation have been designed to provide the most information in a single call. This completeness leads to an invocation pattern well-known by OpenCL developers: the double call pattern foreseen in section 1.5.3.2. JavaCL answers this shabby design through its API that automatically issues a filled array as a result.

When platforms have been identified, it is time for an OpenCL application to list their belonging devices and to create a context combining some of them. As long as the main purpose of a High Performance Computing application is to speed up computation time, developers usually select the most efficient devices to form the OpenCL context in which their application will run. Once again, JavaCL takes care of this step by providing a *createBestContext* method that creates a context with the device containing the most work units.

The two previous features focused on increasing the ease of use of the OpenCL API. Moreover, JavaCL also enhances OpenCL development thanks to the Java language capabilities. Let us now examine the characteristics that make JavaCL really more than a simple OpenCL wrapper.

As we have seen in our introduction to OpenCL, the latter stores data that are to be treated on the device into buffers. The main concern with these buffers is that whatever the data they contain, they are represented by the sole *cl_mem* type. This is a potential source of harm since the compiler has no way to check that you are passing a wrong buffer type as parameter to a kernel for example. On the other side, Object Oriented languages such as Java are used to encapsulating data in dedicated containers. JavaCL does so by providing a different class for each data type that contains a buffer. Thanks to Java generics, a *CLBuffer* class can be parameterized with

the primitive data type the buffer actually contains. Not only expressive containers is a nice feature for developers who have to correct a source code they did not write, but it is a particularly good point that helps compilers find errors. Strong static typing prevents errors encountered when buffers from any type are accepted as parameters, which can appear dramatic when programming hardware accelerators such as GPUs or FPGAs. To sum up, the JavaCL API allows the compiler to check little programming errors at compile time, before they turn into malicious bugs at runtime.

Being compiled at runtime, within the host program, OpenCL sources might lead to compiling errors as any program would. However, due to on-the-fly compiling, errors are a bit more tedious to take into account than in a usual program. Developers have to explicitly request for the compilation error log through the classical double call syntax inherent in the query system of the OpenCL API. This is not particularly suited to solve problems efficiently, and one has to make sure to correctly handle the output of the compiler anytime he builds a kernel through the C API. In the Java world, runtime errors are traditionally handled by exceptions. The exceptions mechanism is cleverly adapted by JavaCL so that any problematic kernel build will rise a *CLBuildException* that can be caught, and whose content can be printed easily.

In the same way, enqueued tasks errors are also wrapped in JavaCL exceptions. An exception hierarchy has been designed to cover the scope of potential errors issued by an OpenCL-enabled application. Explicit exceptions matching the most spread OpenCL errors can be cast throughout the lifetime of the application; for example, memory-bound errors corrupting the command queue are covered by *CLException.InvalidCommandQueue*. Once again, this JavaCL feature relieves the developer from dealing with involved errors codes, while the way Java handles exceptions ensures that errors will be reported at one time or another.

Last but not least, the most promising part in our opinion is the Generator. All the previous features introduced in this section either wrapped OpenCL calls or enhanced the API with type and error checking capabilities for example. The interesting point with the JavaCL generator is its ability to parse a pure OpenCL source file containing one or several kernels, and to issue a corresponding Java source file. The latter contains a class that associates a method per kernel present in the source file. In order to smoothly integrate in the host application, the automatically generated Java class is named after the source file name, whereas each of its methods bears the name of a kernel parsed in the file. Two aspects need to be distinguished to fully understand the inputs the generator brings to OpenCL development.

First, it widely contributes to simplifying OpenCL development: while JavaCL

wrappers already make OpenCL on-the-fly compilation comfortable, the Generator allows developers to skip this step by reading the source code at compile time and generating a corresponding class whose methods can be called to launch the associated kernel.

Second, applications safety is again enforced given that kernel parameters are now typed. Traditionally, kernel parameters need to be set once and for all before enqueueing the kernel. JavaCL brings a first improvement to this process by allowing parameters to be all passed at once as a list of Objects. However, this behaviour might lead to runtime errors since the type and order in which parameters are passed cannot be verified at compile time, each of them being identified by a reference to an Object. Here, the Generator enables an earlier detection of this kind of error, since the prototype of the method acting as a Proxy of the kernel to be enqueued will only match the right type and order of the parameters. Listing 6.6 displays the execution of an instance of the kernel *MyKernel*, which corresponding class was generated upstream by the JavaCL Generator.

```
1  CLContext context = JavaCL.createBestContext();
2  CLQueue queue = context.createDefaultQueue();
3
4  final int vectorSize = 36864;
5  Pointer<Integer> inPtr = allocateInts (vectorSize);
6
7  for (int i = 0; i < vectorSize; ++i) {
8     inPtr.set(i,i);
9  }
10
11 CLBuffer <Integer> inVector = context.createIntBuffer(Usage.Input,
       inPtr);
12
13 // Create an OpenCL output buffer
14 CLBuffer<Integer> outVector = context.createIntBuffer(Usage.Output,
       width*height*8);
15
16 // Read the program sources and compile them
17 MyKernel kernel = new MyKernel(context);
18
19 CLEvent addEvt = kernel.foo(queue, inVector, vectorSize, vectorSize /
       192);
```

```
20
21 // Blocks until the kernel finishes
22 Pointer<Integer> outPtr = outVector.read(queue, addEvt);
```

Listing 6.6: Squaring an array of integers using JavaCL

Finally, the combination of an enhanced wrapping API and a generation mechanism proves that JavaCL provides a more thorough solution than the other APIs studied here.

6.2.4 Summary table of the solutions

In Table 6.1, we compare the APIs studied in this work according to the availability of three features: high-level wrapper, code generation facility and cross-platform portability of the resulting binary that will run on the host.

API	Wrapper	Code-generator	Cross-platform
C++ Wrapper	Yes	No	No
QtOpenCL	Yes	No	No
PyOpenCL	Yes	Yes	Yes
ScalaCL	Yes	Yes	Yes
Aparapi	Yes	Yes	No
JavaCL	Yes	Yes	Yes

Table 6.1: Comparison of the studied APIs according to three criteria

Table 6.1 states that although C++ is one of the most spread language in the HPC community, it suffers from poor APIs to enhance OpenCL.

6.3 Automatic Parallelization of a Gap Model using Aparapi

At the end of Chapter 5, we evoked the importance of choosing the right platform to parallelize an application. We proposed to automatically build parallel prototypes targeting several platforms thanks to high level tools. In this section we study the benefits of parallelizing part of an application using OpenCL code generated through

Aparapi. We previously depicted these two technologies as interesting in the context of automatic parallelization of applications. The said application is a Forest Gap Model simulation described in [Passerat-Palmbach et al., 2012a], while the model itself is more thoroughly described in Appendix A,

We have chosen to employ Aparapi because thanks to its ability to transform Java code into OpenCL code, it is more suitable for a smooth integration within an already existing Java simulation. In this respect, a proof of concept was published by an AMD software engineer to demonstrate the features of Aparapi [Joshi, 2012]. This process will help us determine whether such an automatic parallelization approach is relevant when dealing with legacy simulation models.

The Gap Model depicts the dynamics of trees spawning and falling in a French Guiana rainforest area. The purpose of this model is to serve as a basis for a future agent-based model rendering the settling of ant nests in the area, depending on the trees and gaps locations. At each simulation step, a random number of trees will fall, carrying a random numbers of their neighbours in their fall. In the same time, new trees are spawning in the gaps area to repopulate them.

Here, we focus on the bottleneck of the model: a method called several times at each simulation step that represents about 70% of the whole execution time. Although the whole model is stochastic, the part we consider in this work is purely computational. The idea is to figure out the boundaries of the gaps in the forest. The map is a matrix of boolean wherein cells carrying a *true* value represent parts of gaps, while any other cell carries a *false* value. According to the value of its neighbours, a cell can determine whether it is part of a gap boundary or not.

6.3.1 Profiling

Prior to any parallelization attempt, it is a good point to use a profiler in order to determine which part of the simulation model is the most time consuming. Indeed, due to the complexity of our model, it would have been difficult to design a fully parallel prototype. Thus, we needed a profiling step to figure out what were the critical parts of the model. We used the Netbeans profiler to obtain this information. Figure 6.1 shows the output of this tool on our model.

The screenshot reveals that the hot spot in our model is a method called *setBoundaries*, which is called by every instance of *Gap*, several times at each iteration. Thus, tackling the parallelization of this method that represents 70.5% of the global execution

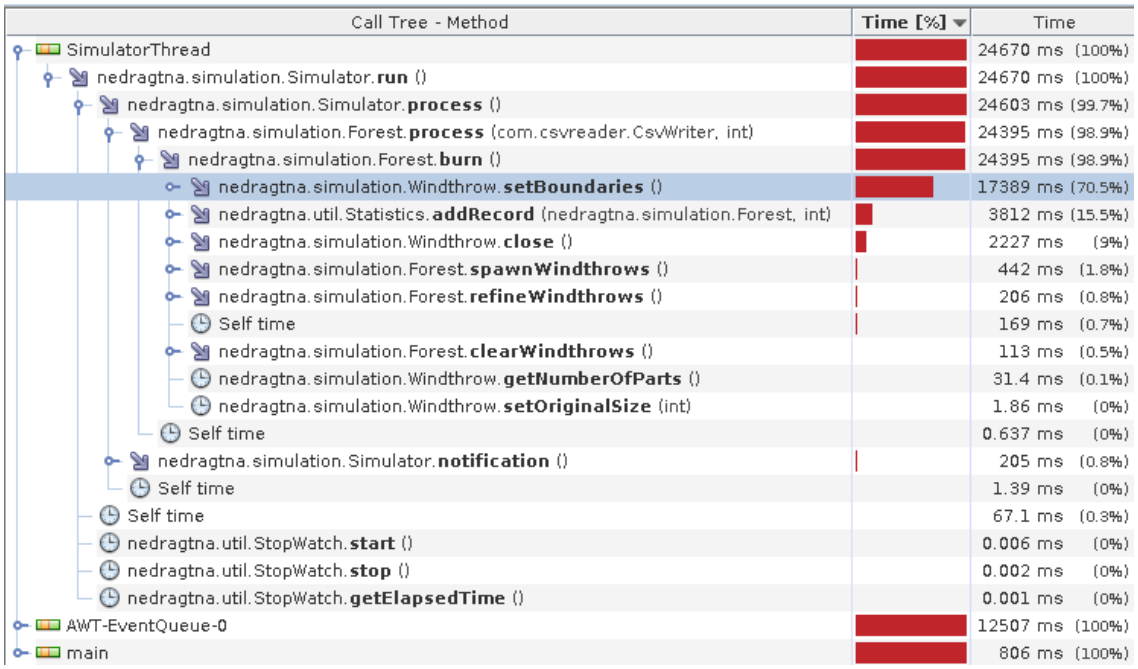


Figure 6.1: Output of the Netbeans Profiler after 200 iterations of the sequential Gap Model

time should speed up the whole simulation. The problem now is to find an appropriate way to parallelize this method while ensuring its output remains the same and taking advantage of the parallel architectures at our disposal.

6.3.2 Implementation

Now that we have identified both the part of the algorithm to be parallelized and the tool that will be used to do so, let us describe the new way to set the boundaries of gaps in parallel.

The sequential version of *setBoundaries* is gap-based, and is called successively on every gap of the map. Let us recall that Aparapi cannot handle types other than primitives yet. Thus, the *Gap* class or its *Forest* container cannot be directly used in the parallel version.

We have chosen to represent the whole *Forest* as a map of boolean cells. A cell that contains part of a gap bears the *true* value, whereas all the other cells are set to *false*. This boolean representation allows us to turn the original algorithm containing lots of nested branches into a boolean expression that involves no branch at all. This way, we

enable our parallel code to run faster when deployed on a GPU platform. In fact, the programming model of GPUs does not cope with heavy branching, and the resulting OpenCL code could suffer from performance issues that would not reflect the actual computing capabilities of GPUs.

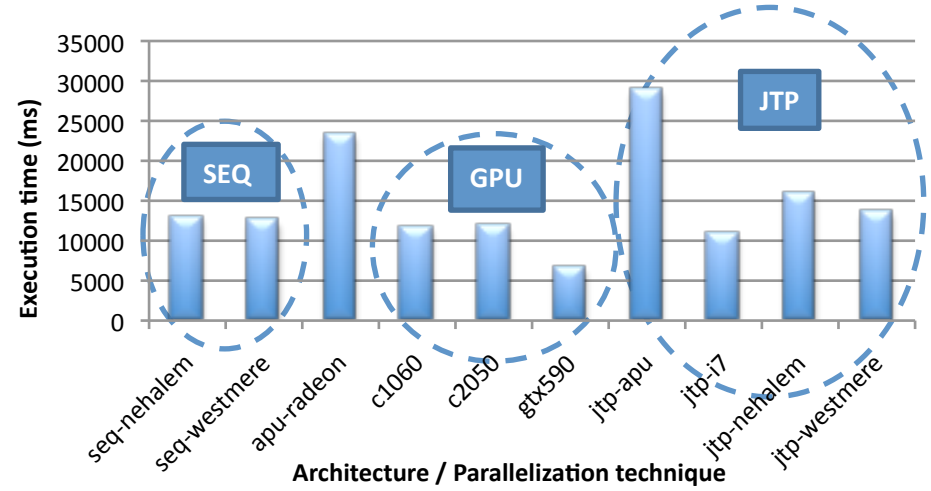
Along with this spatial parallelism, a computing element is assigned to each cell. It will translate differently following the underlying platform they run on. For instance, OpenCL-enabled GPUs will treat them as logical threads whereas they will be tasks when falling back to a Java Thread Pool (JTP) execution.

6.3.3 Results

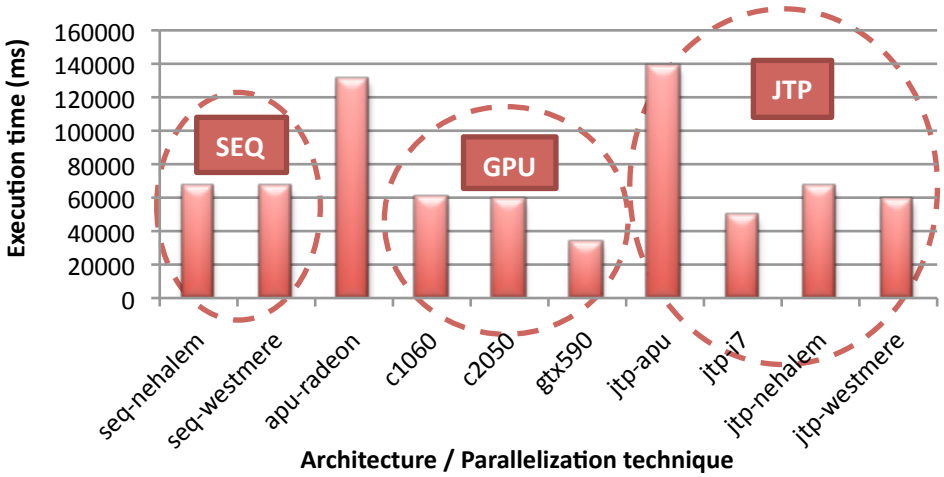
In this section, we compare the execution time of our gap model on various architectures. Actually, there are two main parameters that can lead to different performance for a given algorithm: changing the underlying platform and the size of the data to process. In our case, we compare no less than 10 different platforms, from the original sequential execution to OpenCL declinations generated automatically by Aparapi. We also consider the capability of these approaches to scale with the data by feeding the model with two different maps: a small one containing around 300,000 cells, and a large one that is about five times as big.

The first thing to notice in view of these two charts is that the size of the input data does not impact the parallelization techniques: the most efficient with a small map remains the same when dealing with a large map. It means that the automatic parallelization approach that we set up using Aparapi scales well with data.

Now regarding the platforms, three groups distinguish in both Figures 6.2(a) and 6.2(b): the sequential executions, the OpenCL-enabled GPU ones, and finally the OpenCL-disabled JTP executions. We studied sequential executions on two successive generations of Intel server processors: the recent Westmere and its predecessor, the Nehalem. The execution time on these two CPUs is roughly the same. The second group is formed by NVIDIA GPU platforms. Here again, we tested several generations: a Tesla C1060, a Tesla C2050 and a GeForce GTX 590. The two latter GPUs belong to a more recent architecture codenamed Fermi. Surprisingly, the less powerful GTX outperforms its two counterparts specifically designed to process High Performance Computing jobs. This is due to the host machine in which these devices are embedded. The Tesla GPUs are contained in machines with ECC-RAM memory, which involves a noticeable overhead when accessing data. Given that the automatically parallelized algorithm transfers the whole map at every simulation step, the two Tesla GPUs suf-



(a) Small map (300,000 cells)



(b) Large map (1,500,000 cells)

Figure 6.2: Execution time for 10 different platforms running the simulation on a different map sizes

fer from the slow memory of their host. Last but not least, the JTP runs are quite homogeneous on all the architectures but the AMD APU. This processor is a combination of two rather slow elements: the CPU part is far behind our Intel cores in terms of performance, in the same way that the embedded Radeon GPU cannot stand the comparison with NVIDIA products. Further details on these platforms are given in Section 6.4.3.

As a conclusion, the GPU parallelizations distinguish as the most efficient of our benchmark. They are consequently the ones that we will investigate further to parallelize our Gap Model. As long as all the efficient GPUs belong to the NVIDIA family

of processors, we could even consider switching to a CUDA implementation that would make the most of these hardware accelerators, and offer maximum speed-up to our simulation.

A slight limitation must be noted when looking at these results. In this work, we have run the same parallel algorithm on every hardware architecture at our disposal. However, some platforms might be more efficient with a coarser parallelization grain. JTPs for instance would have fewer tasks to schedule on their worker threads, but this is out of the scope of this study, where we are looking for the best environment to run a given parallelized algorithm.

6.3.4 Perspectives

Although the Aparapi solution is efficient, it is not very convenient to use. Indeed, it still implies some makeshift developments to be integrated in Java applications. First of all, the way it is shipped is not fully satisfying because it needs additional work from the user to be inserted in a Maven build, for example.

Then, the major issue when integrating Aparapi in a Java development chain is that it can only handle primitives Java types. In our case, we have provided methods to convert the objects running our gap model to low-level primitive arrays. If Aparapi does not provide an automatic boxed-to-primitive type converter in its upcoming versions, it could be a good opportunity to create one. Several tools could help achieve this goal, first the Java Reflection API, and especially of the *Javassist* library [Chiba, 1998] that enables defining classes at runtime. One other way lies in an intermediate representation such as Soot [Vallée-Rai et al., 2010], the intermediate representation for Java bytecode that powers Rootbeer [Pratt-Szeliga et al., 2012].

On the pure performance point of view, we have presented our work as a prototype to figure out whether parallelization was suited for our simulation. Now that preliminary results have shown satisfying enough figures, we can spend more time on optimizing the parallel code for the architecture that showed the most promising results. To do so, we have slightly modified the behaviour of Aparapi so that it systematically outputs the resulting OpenCL code. This way, we can modify this code to benefit of some hardware dedicated optimizations. While being a cross-platform tool, OpenCL allows developers to harness the specificities of underlying platforms through a mechanism called *extensions*. In our case, it would be interesting to determine whether hardware-specific optimizations can increase the execution speed of our simulation again.

By doing so, we would however disable Aparapi in our application, since it is unable to read user-written kernels. Still, other Java libraries allow developers to integrate OpenCL kernels at no cost in their applications. We particularly think of JavaCL [Chafik, 2011a], which produces Java wrapper classes at compile time to launch OpenCL kernels from a Java source code.

6.3.5 Summary

In this section, we have presented an automatic parallelization approach using OpenCL, applied to the Gap Model from [Passerat-Palmbach et al., 2012a].

In order to pair Java and OpenCL, we have chosen a free library provided by AMD, called Aparapi, which automatically transformed our sequential Java code into parallel OpenCL code. Automatic parallelization allowed us to get rid of a hot spot that used to slow down the simulation. Now that we have obtained these preliminary results, not only the simulation was sped up thanks to a partial parallelization, but we are now able to target the architecture that appeared as the most efficient, in our case: NVIDIA GPUs. Further development will consequently focus on leveraging this particular platform, from the parallel OpenCL code that we extracted thanks to our contribution to the Aparapi library. This first result enforces the approach to build parallel prototypes to obtain information, and in the case of Aparapi, a first parallel skeleton, prior to any time-consuming development.

In our quick survey about APIs able to generate OpenCL code from a high level abstraction layer, the second solution that appeared lied in a third-party library called ScalaCL. When designing a prototype, the goal is to obtain information that will reduce the next development stage. In this way, a concise and expressive language such as Scala enables us to write the source code of a parallel prototype quickly. As long as Scala runs on top of the JVM, it is perfectly interoperable with any other Java development. In the next section, we will study new automatic parallelization approaches of the Gap Model, as well as two other classical stochastic models, using Scala tools.

6.4 Prototyping Parallel Simulations Using Scala

Although the Java Virtual Machine (JVM) tends to become more and more efficient, the Java language itself evolves slowly and several new languages running on top of

the JVM appeared during the last few years, including Clojure, Groovy and Scala. These languages reduce code bloating and offer solutions to better handle concurrency and parallelism. Among JVM based languages, we have chosen to focus on Scala in this study because of its hybrid aspect: mixing both functional and object oriented paradigms. By doing so, Scala allows object-oriented developers to gently migrate to powerful functional constructs. Meanwhile, functional programming copes very well with parallelism due to its immutability principles. Having immutable objects and collections highly simplifies parallel source code since no synchronization mechanism is required when accessing the original data. As many other functional languages such as Haskell, F# or Erlang, Scala has leveraged these aspects to provide transparent parallelization facilities. In the standard library, these parallelization facilities use system threads and are limited to CPU parallelism. In the same time, the ScalaCL library generates OpenCL code at compile-time and produces OpenCL-enabled binaries from pure Scala code. Consequently, it is easier to generate automatically parallelized functional programming constructs.

In this section, we study the ability of the Scala programming language to fulfil the need for automatically built parallel prototypes. We compare the features of two frameworks: Scala Parallel Collections and ScalaCL. Both of them provide facilities to set up a data-parallelism approach on Scala collections. The capabilities of the two frameworks are benchmarked with three simulation models as well as a large set of parallel architectures.

6.4.1 Automatic Parallelization using Scala

6.4.1.1 Why is Scala a good candidate for parallelization of simulations?

First of all, let us make a brief recall on what is Scala. Scala is a programming language mixing two paradigms: object oriented and functional programming. Its main feature is that it runs on top of the Java Virtual Machine (JVM). In our case, it means that Scala developments can interoperate like clockwork with Java, thus allowing the wide range of simulations developed in Java to integrate Scala code without being modified.

The second asset of Scala is its compiler. In fact, Scalac (for Scala Compiler) offers the possibility to enhance its behavior through plugins. This mechanism offers great opportunities to generate code and ScalaCL, introduced in Section 6.2.2.1, is just a concrete example of what can be achieved when extending the Scala compiler.

Finally, Scala presents a collection framework that intrinsically facilitates paral-

lelization. As a matter of fact, Scala default collections are immutable: every time a function is applied to an immutable collection, this one remains unchanged and the result is a modified copy returned by the function. On the other hand, mutable collections are also available when explicitly summoned, albeit the Scala specification does not ensure any thread-safe access on these collections. Such an approach appears to be very interesting and efficient, when trying to parallelize an application, since no lock mechanisms are involved. Thus, concurrent accesses to the elements of the collection are not a problem as long as they are read-only accesses that do not introduce any overhead. It is a classical functional programming pattern to issue a new collection as the result of applying a function to an initial immutable collection. Although this process might appear costly, works have been done to optimize the way it is handled, and efficient implementations do not copy the entire immutable collection [Okasaki, 1999].

6.4.1.2 Scala Parallel Collections

Scala 2.9 release introduced a new set of Parallel collections mirroring the classical ones. They have been described in [Prokopec et al., 2011]. These parallel collections offer the same methods as their sequential equivalents, but the method execution will be automatically parallelized by a framework implementing a divide and conquer algorithm.

The point is they integrate seamlessly in already existing source codes because the parallel operations bear the same names as their sequential variants. As the parallel operations are implemented in separate classes they can be invoked if their data are in a parallel collection class. This is made possible thanks to a `par` method that returns a parallel equivalent of the sequential implementation of the collection, still pointing to the same data. Any subsequent operation invoked by an instance of the collection will benefit of a parallel execution without any other add from the client. Instead of applying the selected operation to each member of the collection sequentially, it is applied on each element in parallel.

Scala Parallel Collections rely on the Fork/Join framework proposed by Doug Lea [Lea, 2000]. This framework was released with the latest Java 7 SDK. It is based upon the divide and conquer paradigm. Fork/Join introduces the notion of tasks assigned to worker threads waiting in a sleeping state in a Thread Pool.Fork/Join is implemented using work stealing. This adaptive scheduling technique offers efficient load balancing features to the Java framework, provided that work is split into tasks of a small enough

granularity. Tasks are assigned to the queues of the workers, but when a worker is idle, it can steal tasks from the queue of another worker, which helps achieve the whole computation faster. Scala Parallel Collections implement an exponential task splitting technique detailed in [Cong et al., 2008] to determine the ideal task granularity.

The approach proposed by Scala Parallel Collections is, in the end, very close to what the mechanism of ScalaCL, as shown in Figure 6.3.

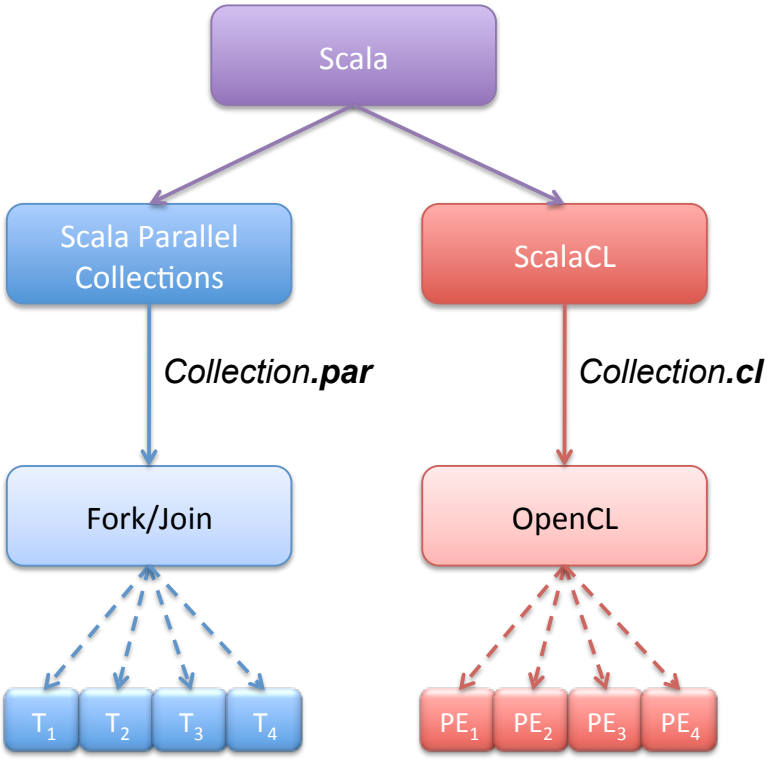


Figure 6.3: Schema showing the similarities between the two approaches: Scala Parallel Collections spread the workload among CPU threads (T), while ScalaCL spreads it among OpenCL Processing Elements (PEs)

6.4.2 Case study: three different simulation models

In this section, we will show how parallel Scala implementations of three different simulation models are close to the sequential Scala implementation. We compare sequential Scala code with its parallel declinations, and put the light on the automatic aspect of the two studied approaches. The source codes resulting from each approach remain very close to the genuine. Our benchmark consists in running several iterations of

the automatically parallelized models, and compare them with handcrafted parallel implementations.

The three models used in the benchmark were carefully chosen so that they remain simple to describe, while being representative of several main modelling methods. They are: the Ising Model [Ising, 1925], the forest Gap Model described in Appendix A [Passerat-Palmbach et al., 2012a] and Schelling's segregation model [Schelling, 1971]. They will be described more thoroughly along the next lines. Three categories are considered to classify the models:

- Is the model stepwise or agent-based?
- Does the model outputs depend on stochasticity?
- Is the model performing more computations or data accesses?

Let us sum the characteristics of our three models according to the three aforementioned categories in Table 6.2.

	Stepwise/Agent-based	Stochastic	Computational/Data
Ising	Stepwise	Yes	Computational
Gap Model	Stepwise	No	Data
Schelling	Agent-based	Yes	Data

Table 6.2: Summary of the studied models' characteristics

6.4.2.1 The case of Discrete Event Simulations (DES)

Discrete-event based models do not cope well with data-parallelism approaches. In fact, they are more likely to fit a task parallel framework. For instance, Odersky et al. introduce a discrete-event model of a circuit in [Odersky et al., 2008]. The purpose of this model is to design and implement a simulator for digital circuits. Such a model is the perfect example of the difficulty to take advantage of parallel collections when the problem is not suited. It implies communications to broadcast the events. Furthermore, events lead the order in which tasks are executed, whereas data-parallel techniques rely on independent computations. Task parallelism approaches are not considered in this work, but Scala also provides ways to easily parallelize such problems through the Scala Actors framework [Haller and Odersky, 2009]. The interested reader can find further details on the task-parallel implementation of the previously mentioned digital circuit model using Actors in [Odersky et al., 2008].

6.4.2.2 Models description

The first model to be used in the benchmark is the Ising model: a mathematical model representing ferromagnetism in statistical physics [Ising, 1925]. Ising models deal with a lattice of points, each point holding a spin value, which is a quantum property of physical particles. At each step of the stochastic simulation process and for every spin of the lattice, the algorithm determines whether it has to be flipped or not. The decision to flip or not the spin of a point in the lattice is taken in view of two criteria. The first is the value of the spins belonging to the Von Neumann neighbourhood of the current point, and the second is a random process called the Metropolis criterion.

Ising models have been studied in many dimensions, but for the purpose of this study, we consider a 2D toric lattice, solved using the Metropolis algorithm [Metropolis et al., 1953].

The second model on which we apply automatic parallelization techniques is the Forest Gap Model already foreseen in Section 6.3. Here, we focus on the bottleneck of the model: a method called several times at each simulation step that represents about 70% of the whole execution time. Although the whole model is stochastic, the part we consider in this work is purely computational.

Finally, we will implement the model of segregation of Schelling [Schelling, 1971]: a specialized individual based model that simulates the dynamics of two populations of distinct colors. Each individual wants to live in an area where at least a certain ratio of individuals of the same color as his own are living. Individuals that are unhappy with their neighbourhood move at random to another part of the map. This model converges toward a space segregated by colors with huge clusters of individual of the same color.

At the heart of the segregation model is all the decisions taken by the individuals to move from their place to another. As long as this is the most computation intensive part of the segregation model, we will concentrate our parallelization efforts on this part of the algorithm. Furthermore, it presents a naturally parallel aspect since individuals decide whether they feel the need to move independently from each other.

6.4.2.3 Scala implementations

In this section we will focus on the implementation details of the Ising model. The other implementations would not bring more precisions concerning the use of the two

Scala parallelization frameworks studied in this work.

Our Scala implementation of the Ising model represents the spin lattice by an *IndexedSeq*. *IndexedSeq* is a trait, i.e. an enhanced interface in the sense of Java that also allows methods definition; it abstracts all sorts of indexed sequences. Indexed sequences are collections that have a defined order of elements and provide efficient random access to these elements. It bears operations on this collection such as applying a given function to every element of the collection (`map`) or applying a binary operator on a collection, going through elements from left or right (`foldLeft`, `foldRight`). These operations are sufficient to express most of the algorithms. Moreover, they can be combined since they build and return a new collection containing the new values of the elements.

For instance, computing the magnetization of the whole lattice consists in applying a `foldLeft` on the *IndexSeq* to sum the values corresponding to the spin of the elements. Indeed, our lattice is a set of tuples contained in the *IndexedSeq*. Each tuple stores its coordinates in the 2D-lattice in order to easily build its Von Neumann neighbourhood, and a boolean indicating the spin of the element (*false* is a negative spin, whereas *true* stands for a positive spin).

To harness parallel architectures, we need to slightly rewrite this algorithm. The initial version consists in trying to flip the spin of a single point of the lattice at each step. This action is considered as a transformation between two configurations of the lattice. However, each point can be treated in parallel, provided that its neighbours are not updated at the same time. Therefore, the points cannot be chosen at random anymore, this would necessarily lead to a biased configuration A way to avoid this problem is to separate the lattice in two halves that will be processed sequentially. This technique is commonly referred to as the "checkerboard algorithm" [Preis et al., 2009]. In fact, the Von Neumann neighbourhood of a point located on what will be considered a white square, will only be formed by points located on black squares, and vice versa. The whole lattice is then processed in two times to obtain a result equivalent to the sequential process. The process is summed up in Figure 6.4.

The implementation lies in applying an operation to update each spin through two successive `map` invocations on the two-halves of the lattice. Not only this approach is crystal clear for the reader, but also it is quite easy to parallelize. Indeed, `map` can directly interact with the two Scala automatic parallelization frameworks presented earlier: Scala Parallel Collections and ScalaCL. Let us describe how the parallelization APIs integrate smoothly in already written Scala code with a concrete example.

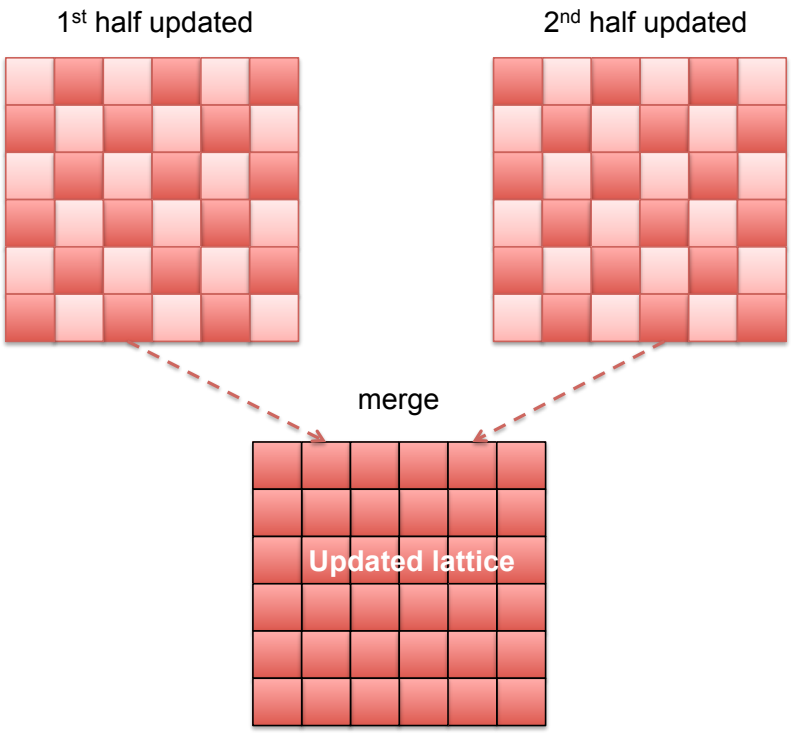


Figure 6.4: Lattice updated in two times following a checkerboard approach

Listing 6.7 is a snippet of our Ising model implementation in charge of updating the whole lattice in a sequential fashion.

```
1  def processLattice(_lattice: Lattice)(implicit rng: Random) =
2
3    new Lattice {
4      val size = _lattice.size
5      val lattice =
6        IndexedSeq.concat (
7          _lattice.filter{case((x, y), _) => isEven(x, y)}.map(spin =>
               processSpin(_lattice, spin)),
8          _lattice.filter{case((x, y), _) => isOdd(x, y)}.map(spin =>
               processSpin(_lattice, spin))
9        )
10   }
```

Listing 6.7: Sequential version of method processLattice from class IsingModel

Scala enables us to write both concise and expressive code, while keeping the most important parts of the algorithm exposed. Here, the two calls to `map` aiming at updating

each half of the lattice can be noticed. They will process in turn all the elements within the subset they have received in input. This snippet suggests an obvious parallelization of this process through the invocation of the `par` method. This call automatically provides the parallel equivalent of the collection, where all the elements will be treated in parallel by the mapped closure that processes the energy of the spins. The resulting code differs only by the extra call to the `par` method prior to the `map` action, so as Listing 6.8 shows.

```
1  def processLattice(_lattice: Lattice)(implicit rng: Random) =
2
3    new Lattice {
4      val size = _lattice.size
5      val lattice =
6        IndexedSeq.concat (
7          _lattice.filter{case((x, y), _) => isEven(x, y)}.par.map(spin =>
               processSpin(_lattice, spin)),
8          _lattice.filter{case((x, y), _) => isOdd(x, y)}.par.map(spin =>
               processSpin(_lattice, spin))
9        )
10   }
```

Listing 6.8: Parallel version of method processLattice from class IsingModel, using Scala Parallel Collections

An equivalent parallelization using ScalaCL is obtained only by replacing the `par` method by the `cl` one from the ScalaCL framework, thus enabling the code to run on any OpenCL-enabled platform.

Both Listings 6.7 and 6.8 mention an implicit parameter labelled `rng` of type `Random`. As its name suggests, it is obviously the instance of PRNG that is to be used by the model implementation. Anywhere a reference to an object `rng` is made, Scala will assume that this parameter is intended, hence it bears an `implicit` qualifier. Beyond the Scala machinery that hides behind this statement, an interesting problem is raised: how will automatic parallelization process handle the pseudorandom stream distribution across the Processing Elements (PEs)? Actually, any instance of a PRNG matching the interface of `java.util.Random` would fit the basic needs: *i.e.* picking up pseudorandom numbers. However, a class with the ability to deal with pseudorandom stream distribution is required in that case. TaskLocalRandom, depicted in Chapter 4.4, owns pseudorandom stream distribution features. At the time of writing, it cannot be used in this context yet. Some Scala code has to be adapted for Scala to take into

account the `RandomSafeRunnable` class associated to TaskLocalRandom. This work is in progress but not available yet due to a lack of time.

As automatic parallelization applies on determined parts of the initial code, the percentage of the sequential execution time affected by the parallel declinations can be computed through a profiler. Thanks to this tool, we were able to determine the theoretical benefits of attempting to parallelize part of a given model. These figures are presented in Table 6.3, along with the intrinsic parameters at the heart of each model implementation. For more details about the purpose of these parameters, the interested reader can refer to the respective papers introducing each model thoroughly.

Model	**Part of the Sequential Execution Time Subject to Parallelization**	**Intrinsic Parameters**
Gap Model [Passerat-Palmbach et al., 2012a]	70%	Map of 584x492 cells
Ising [Ising, 1925]	38%	Map of 2048x2048 cells; Threshold = 0.5
Schelling [Schelling, 1971]	50%	Map of 500x500 cells; Part of free cells at initialization = 2%; Equally-sized B/W population; Neighbourhood = 2 cells

Table 6.3: Characteristics of the three studied models

6.4.3 Results

In this section, we will study how the different implementations of the models behaved when run on a set of various architectures. All the platforms that were used in the benchmark are listed hereafter:

- 2-CPU 8-core Intel Xeon E5630 (Westmere architecture) running at 2.53GHz with ECC-memory enabled

- 8-CPU 8-core Intel Xeon E7-8837 running at 2.67GHz
- 4-core 2-thread Intel i7-2600K running at 3.40GHz
- NVIDIA GeForce GTX 580 (Fermi architecture)
- NVIDIA Tesla C2075 (Fermi architecture)
- NVIDIA Tesla K20c (Kepler architecture)

We compare a sequential Scala implementation executed on a single core of the Intel Xeon E5630 with its Scala Parallel Collections and ScalaCL declinations executed on all the platforms. Each model had to process the same input configurations for this benchmark. Measures resulting from these runs are displayed in Table 6.4.

Model	**Sequential (Xeon E5630)**	**ScalaPC (i7)**	**ScalaPC (Xeon E7-8837)**	**ScalaPC (Xeon E5630)**
Gap Model	150.04	75.67	128	109.42
Ising	45.35	11.63	33	26.83
Schelling	2961.52	525.63	344	412.86

Table 6.4: Execution times in seconds of the three models on several parallel platforms (ScalaPC stands for Scala Parallel Collections)

Unfortunately, at the time of writing ScalaCL is not able to translate an application as complex as Schelling's segregation model to OpenCL yet. In the meantime, while ScalaCL managed to produce OpenCL versions of the Gap Model and the Ising Model, these two declinations were killed by the system before being able to complete their execution.

As a consequence, we are not able to provide any result about a ScalaCL implementation of these models. This leads to the first result of this work: at the time of writing, ScalaCL cannot be used to build parallel prototypes of simulations. It is just a proof of concept that Scala closures can be transformed to OpenCL code, but it is not reliable enough to achieve such transformations on real simulation codes yet.

On the other hand, the Scala Parallel Collection API succeeded to build a parallel declination of the three benchmarked models. These declinations have successfully run on the CPU architectures retained in the benchmark, and have shown a potential performance gain when parallelizing the 3 studied models.

Let us validate this trend by now comparing handcrafted implementations on CPU and GPU. For the sake of brevity, we will focus on the Forest Gap Model, which

implementation using Scala Parallel Collections runs twice as fast as the sequential implementation on the Intel i7, according to Table 6.4. The CPU declination uses a Java Thread Pool, whereas the GPU is leveraged using OpenCL. Figure 6.5 shows the speed-ups obtained with the different parallelizations of the Forest Gap Model.

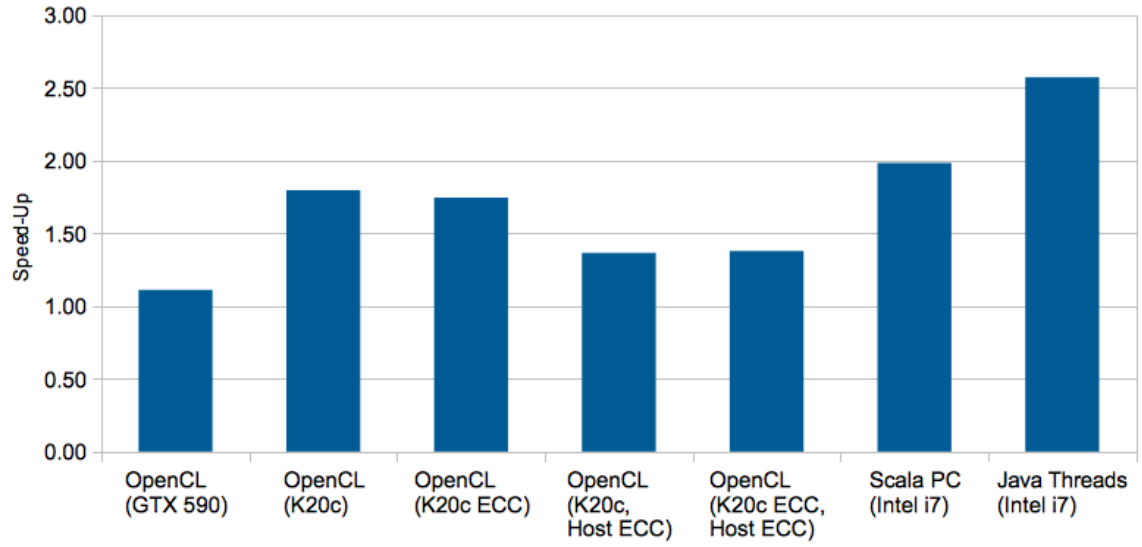


Figure 6.5: Speed-up obtained for the Gap Model depending on the underlying technique of parallelization

Results from Figure 6.5 show that an handcrafted parallelization on CPU follows the trend of the parallel prototype in terms of performance gain. In view of the lower speed-up obtained on GPU using an OpenCL implementation, it is likely that an automatically built GPU-enabled prototype would have displayed worse performance than its CPU counterpart.

Not only this last result shows that the automatic prototypes approach gives significant results, when using Scala Parallel Collections, but this also validates the relevancy of the whole approach when considering the characteristics of the involved simulation model. In Table 6.2, the Forest Gap Model had been stated as a model performing more data accesses than computations. Thus, it is logical than a parallelization attempt on GPU suffers from the latency related to memory accesses on this kind of architecture. Indeed GPUs can only be fully exploited by applications with a high computational workload. Here, the OpenCL declination performs slightly faster than the sequential one (1.11X), but it is the perfect case of the choice of an unsuited architecture that leads to disappointing performance, as exposed in introduction to this Chapter.

Automatically built parallel prototypes quickly put the light on this weakness, and avoid wasting time to develop a GPU implementation that would be less efficient than

the parallel CPU version.

In the same way, we can see from Table 6.4 that the number of available cores is not the only thing to consider when selecting a target platform. Depending on the characteristics of the model, which are summed up in Table 6.2, parallel prototypes behave differently when faced to manycore architectures they cannot fully exploit due to frequent memory accesses. The perfect example of this trend is the results obtained when running the models on the Xeon E5630, an ECC-enabled machine. As fast as it can be, this CPU-based host is significantly slowed down by its ECC memory. Indeed, ECC is known to impact execution times of about 20%.

6.4.4 Summary

In this section we have benchmarked two Scala proposals aiming at automatically parallelizing applications and their ability to help simulation practitioners to figure out the best target platform on which to parallelize their simulation. The two frameworks have been detailed and compared: Scala parallel collections that automatically creates tasks, and execute them in parallel thanks to a pool of worker threads; ScalaCL, part of the *nativelibs4java* project, which intends to produce OpenCL code from Scala closures, and to run it on the most efficient device available, be it GPU or multicore CPU.

Our study has stated that ScalaCL was still in its infancy and could only translate a limited set of Scala constructs to OpenCL at the time of writing. Although it is not able to produce a parallel prototype for a simulation yet, it deserves to be regarded as a future great language extension if it manages to improve its reliability. For its part, Scala Parallel Collections is a really satisfying framework that mixes ease of use and efficiency.

6.5 Conclusion

Regarding the experience with the Gap Model, the automatic parallelization approach has shown to be satisfying enough to design a parallel prototype of our simulation. As we have seen in the results section, automatic parallelization does not allow to leverage the most of parallel architectures, but it gives precious hints about the behaviour of an application in parallel. In this way, OpenCL is a great tool combined to automatic parallelization because it allows developers to test various kinds of architectures without changing their code at all.

Considering data-parallel simulation models at a larger scope, this work has shown how simulation practitioners can easily determine the best parallel platform on which to concentrate their development efforts. Especially, this study puts forward the inefficiency of some architectures, in our case GPUs, when faced with problems they were not designed to process initially. In the literature, some architectures have often been favoured because of their cutting-edge characteristics. However, they might not be the best solution to retain, while other better suited solutions might be underrated [Lee et al., 2010]. Thus, being able to quickly benchmark several architectures without further code rewriting is a great decision support tool. Such an approach is very important when speed-up matters to make sure that the underlying architecture is the most suited to fasten the problem.

The Scala community owns two interesting tools with Scala Parallel Collections and ScalaCL to automatically build parallel prototypes. Although ScalaCL has displayed its limits when faced with real-life applications, it is a promising effort that is worth considering for future benchmarks. The whole Java ecosystem is full of other tools such as Aparapi or JavaCL, which have been exploited in this study and have proved their efficiency to build prototypes.

Parallel prototypes are obviously useful to decide which parallel platform to target. Still, the underlying standard chosen to generate code for various platforms needs to be carefully chosen. In our case, OpenCL is widespread among parallel platforms, and allows a large choice of target architectures. Still, the relevance of the results it produces are extremely dependent on the quality of the implementation of the OpenCL driver. For example, a new parallel platform such as the Intel Xeon Phi supports OpenCL code, but the performance of OpenCL on this hardware accelerator are so bad that it is not relevant to take them into consideration when choosing the best platform on which to parallelize an application. In the future, we will consequently consider other high-level parallel abstractions such as OpenMP, which recent roadmap plans to add support for GPUs, thus extending the scope of OpenMP directives to most of the available parallel architectures.

Moreover, we have seen that automatic parallelization using Scala also raised new challenges regarding pseudorandom stream distribution. It backs our development from Chapter 4.4 and gives us new prospects to improve TaskLocalRandom.

This last chapter about automatic parallelization in the context of simulation closes the studies of this PhD thesis. We will now sum up our inputs and expose the future of this work in the last chapter.

Conclusion

> “ *It's a triumph. What thoroughness! What realism! Knew when to stop, too – didn't cut the pages. But what do you want? What do you expect?*
>
> — Francis Scott Fitzgerald, *The Great Gatsby*

This work relies on the assertion that parallel stochastic simulations imply a great care in the way pseudorandom numbers are provided to each parallel Computing Element [De Matteis and Pagnutti, 1990a; Pawlikowski and Yau, 1992; Hellekalek, 1998b; Pawlikowski and McNickle, 2001]. We have studied software engineering tools to better handle the distribution of pseudorandom streams in parallel, and especially on GPU architectures. All along the 6 chapters of this document, our theoretical proposals have been in turn presented, implemented in software libraries and concretely used in stochastic simulations.

Summary

First we have surveyed the use of random number generation for HPC and presented the main partitioning methods for stochastic parallel simulations [Hill et al., 2013]. A focus has been made on the difficulty to obtain good quality pseudorandom sequences on GPU [Passerat-Palmbach et al., 2012b], since it implies taking into consideration two different domains: GPU programming and PRNG parallelization techniques. We suggest that libraries handling the distribution of pseudorandom streams for the user represent a good software engineering practice. Libraries noticed in the state of the art, such as *cuRand* and *Thrust* for the sole GPU world, represent interesting proposal in this way, but they do not combine good software practice and sound theoretical basis.

These lacks led us to propose a set of criteria for PRNGs dedicated to GPUs. We defined a set of requirements that should be met by any PRNG implementing a distribution technique on GPU. Another set of guidelines completes it by addressing the specificity of the architecture of GPUs, the way they schedule threads and their memory hierarchy. In the same time, we investigated whether well-known pseudorandom streams distribution techniques matched GPUs. This is summarized in a taxonomy of

the distribution techniques, as well as a table assessing the suitability of these techniques in various concrete use cases and parallel environments.

We have implemented our guidelines in two tools targeting NVIDIA GPUs and manycore CPUs. ShoveRand [Passerat-Palmbach et al., 2011a], the CUDA-enabled development, is now fully functioning and offers features equivalent to what can be found in its counterparts: *cuRand* and *Thrust*, added to a better handling of pseudorandom streams distribution across CUDA threads. Having presented ShoveRand at several conferences, we noticed that it interested the attendees, but still lacked the visibility to be used by a wider community. The presentation of ShoveRand at NVIDIA's GPU Technology Conference (GTC 2013) allowed us to get in touch with the developers of Thrust. In order to spread the good practices introduced in ShoveRand, we consider proposing parts of it to be merged into Thrust in the near future.

TaskLocalRandom [Passerat-Palmbach et al., 2013c] is, to our knowledge the only library that takes Java tasks into consideration when it comes to safely distribute pseudorandom numbers. It was designed to act as a substitute to *ThreadLocalRandom*, the original class from JDK 7. Although bugs preventing a correct distribution of the pseudorandom streams have been solved in recent versions of *ThreadLocalRandom*, the LCG at the heart of *ThreadLocalRandom* remains a weakness for any scientific application that would make use of this class. This is why, we believe our developments bring an input for such applications.

The Java community seems to be concerned by this problematic, and even plans to integrate a similar development in the next release: JDK 8. This effort is currently named *SplittableRandom* [1]. It is stated by its authors (Guy Steele and Doug Lea) as *"A generator of uniform pseudorandom values applicable for use in (among other contexts) isolated parallel computations that may generate subtasks."*. We will also contact these authors to take part in this development, through what has been implemented in TaskLocalRandom.

The two tracks followed in Chapter 4 show the growing interest of the High Performance Computing community in the safe distribution of pseudorandom sequences among the Processing Elements. We have proposed tools for both CUDA and Java applications. These developments are echoed by programming efforts from each community, that back our research works. In order to share our experience with these communities, we expect to integrate parts of our implementations in their standard and widely spread libraries.

[1] `http://gee.cs.oswego.edu/dl/jsr166/dist/docs/java/util/SplittableRandom.html`, *last access 7/29/13*

ShoveRand is in use in stochastic simulations such as the Polymer Folding Model that we described in this thesis. This model exploits our Possible Futures Algorithm (PFA) [Passerat-Palmbach et al., 2013a], which relies on the parallel computation of possible evolutions of the same state, to increase the probability to obtain a valid state at each step. Compared to the initial sequential model the acceptance rate of new states significantly increased without impacting the execution time. The PFA declination becomes more and more efficient with the rise of the confinement chromosome.

A Forest Gap Model [Passerat-Palmbach et al., 2012a], along with Ising's Model [Ising, 1925] and the Schelling's Segregation Model [Schelling, 1971] constitute the set of data-parallel models used to benchmark automatic parallelization techniques in the last part of this thesis. We lay the foundations of an approach consisting in evaluating the wider possible set of parallel platforms before actually parallelizing an application [Passerat-Palmbach et al., 2013d]. Automatic parallelization tools enable us to quickly build parallel prototypes that will run on each and every parallel platform available. While this approach cannot be expected to completely parallelize a whole application for free, it gives precious hints on the behaviour of the said application on several parallel architectures. In doing so, we plan to be able to choose the best accelerator instead of investing time on a development conducting to disappointing results in terms of speed-up.

Research Prospects

Towards a New Design of Pseudorandom Streams Distribution

Most of the time, we are testing pseudorandom sequences to validate them, before they are considered safe to be used in stochastic simulations. Another practical perspective could be to formalize an assessment process on the basis of the following tuple: the nature of the application, the random number generator selected, the partitioning method and the results of the current test methods, when available. A test method can be viewed as a rough model for a particular set of applications. Systematic testing of parallel sequences can be achieved once and for all for a particular generator, thanks to the current level of computing power available. As said previously, Reuillon has used the EGI Grid to achieve the test of 65,536 parallel random streams for Mersenne Twister 19937 [Reuillon, 2008b]. The initial statuses have been generated with the DC proposed by Matsumoto and Nishimura. In the context of GPUs, we have also tested the dynamic creation of Mersenne Twister generators tuned for GPU architectures:

Mersenne Twister for Graphics Processors (MTGP) and proposed a thorough benchmark of this PRNG in Section 3.2. This method can be applied to any new PRNG will potentially invest the parallel stochastic simulation community.

The next step towards a new way to consider pseudorandom streams in applications is to fully integrate them into the modelling stage. For instance, object-oriented models could describe objects aware of their pseudorandom behaviour. This way, pseudorandom sequences could be assigned to instances, which fixes the order in which numbers are picked up in the sequence. On the one hand, pseudorandom sequences assigned to Processing Elements (PEs) are shared by all the objects running in a given PE. On the other hand, assigning sequences directly to objects adds another element to our previous theoretical tuple that ensures the sequence will be used in the way it was intended to. In conclusion, we hope to see the emergence of random-aware agents in stochastic simulations.

Can Object-Oriented Modelling Really Fit the Memory Hierarchy of GPUs?

Complex simulations present several constraints. It is much more convenient to design them using an Object-Oriented approach to structure their data structures. Object-oriented design implies to store the state of objects. This state follows the objects through all its lifetime, and contains data related to the particular instance of the class this object represents.

In physical memory, this data storage scheme translates in a series of bytes containing the data within the object. The latter bytes are stored contiguously, so that two objects will not overlap in memory. Depending on the memory allocation scheme and the time at which several objects are created, the internal data structures will be more or less scattered in physical memory.

On the other hand, the parallel architecture of GPUs is bound to a hierarchy of memory, legacy from their original graphics computation purpose. As long as several threads will perform the same operation at the same time (SIMT: Single Instruction, Multiple Threads), they will also need to reach the global memory of the GPU at the same time.

Since the first attempts of parallelizing general-purpose algorithms on GPU, the main bottleneck has always been memory accesses. Actually, the main memory of GPUs is implemented using DRAM. This kind of memory is intrinsically slower than

the computation cores of GPUs. Unfortunately, the more the performance of the cores are improved, the more the gap between their frequency and the memory bandwidth widens. According to [Hwu2012, NVIDIA2012], the rate is such that cores are about 8 times as fast as memory, at the time of writing. To overcome its slowness, DRAM implements a mechanism called burst mode, that allows several bytes from consecutive areas to be read in the same burst section at the same time. This way, a costly memory access can deliver a large amount of data at once.

Although this kind of memory access seems particularly suited to SIMD-like architectures, such as GPUs, developers must perform these accesses correctly to leverage the maximum bandwidth from the memory. When SIMD-units access data stored in non-consecutive locations in memory, we talk of uncoalesced loads. The latter disable the efficiency of the burst mode, since the requested data are part of different burst sections, read through several burst accesses.

In the case of GPUs, data are stored in row-major mode in memory. It means that two-dimensional arrays will be stored one row after another in memory. Thus, GPU threads accessing neighbouring elements of a column would perform uncoalesced loads. Actually, two kinds of uncoalesced loads can be distinguished: those tied to non-consecutive addresses and those that spread across several burst sections. The two of them will result in a significant loss of performance, although nowadays GPUs make use of cache mechanism to attenuate the effect of uncoalesced loads.

The architecture deals with this matter by organizing the global memory of GPUs in several banks. For instance, the 3GB of RAM memory available on an NVIDIA C2050 are split in 16 banks. This leads to a particular fashion to fetch data from memory, where threads requesting different data will actually gain access to different banks of the memory. Full-parallelism for memory accesses is obtained when threads running on different Streaming Multiprocessors (SM hereafter, the GPU equivalent of CPU cores), target different banks. When this is not the case, we say that there is a bank conflict and the memory access is not coalescent.

Typically, data are scattered through memory banks in a way that threads bearing consecutive identifiers should interact with different memory banks, in order to retrieve or store their data. Now, back to our original Object-Oriented considerations, it means that threads of consecutive identifiers will create instances of the same class at the same time. Internal members contained in these objects will be stored consecutively in memory, but as whole object. The point is developers scarcely manipulate a whole object, instead, they tend to reach individual members, be they of a primitive data type, or an object themselves. Thus, threads of consecutive identifiers accessing to the

same member will result in non-coalesced memory accesses, since the latter members are not stored consecutively in global memory, but separated by the other members within each object.

The situation is highly paradoxical since a feature that has been added to GPU programming to ease development can in the same time lead to dramatic performance loss! In the meantime, developers are used to Object-Oriented design, and if thinking parallel algorithms is a first hindrance, designing the associated parallel structure might again slow the development process. Problems related to data structures not suited for the memory model of the underlying platform have been studied for data structures linked to linear algebra algorithms [Kirschenmann et al., 2009; Kirschenmann, 2012]. These works propose parallel skeleton to generate several implementations of the data structures, each tuned for a specific architecture. For Java applications, Rootbeer [Pratt-Szeliga et al., 2012] can transform almost any Java construct into CUDA code. In the particular case of Object-Oriented data structures used on GPU, we suggest a software engineering tool that would enable developers to write their data structure following the Object-Oriented paradigm, while a third-party software tool would be in charge to reorganize the data structure before the compilation stage. Such an approach could benefit from Aspect-Oriented Programming, as structures to be "flattened" could be identified by a specific annotation., without forcing developers to rewrite their source code.

Evaluate the Benefits of Automatic Parallelization

In Chapter 6, we studied an approach involving automatically built prototypes to support the decision of the parallel platform to target for a development. This first attempt has proved efficient but it could be interesting to produce data characterizing its actual value in a software project.

We plan to propose metrics to set out the benefits of automatically built prototypes when considering the human resources involved and the development time. In doing so, the potential benefits in terms of speed-up obtained from a parallel platform would be tempered by other aspects such as the difficulty to develop on this platform, or the programming languages available. Our idea is that speed-up should not be the ultimate goal of parallel developments, since some applications can easily settle for a lower speed-up obtained in a shorter development time.

Bibliography

Acevedo, M. F., Urban, D. L., and Ablan, M. (1995). Transition and gap models of forest dynamics. *Ecological Applications*, 5(4):1040–1055. (Cited on page 205.)

Ackermann, J., Tangen, U., Bodekker, B., Breyer, J., Stoll, E., and McCaskill, J. (2001). Parallel random number generator for inexpensive configurable hardware cells. *Computer physics communications*, 140(3):293–302. (Cited on page 35.)

Alexandrescu, A. (2001). *Modern C++ Design*. Addison-Wesley. ISBN: 0-201-70431-5. (Cited on pages 86 and 218.)

Aluru, S. (1997). Lagged fibonacci random number generators for distributed memory parallel computers. *Journal of Parallel and Distributed Computing*, 45(1):1–12. (Cited on page 32.)

Amblard, F., Hill, D. R. C., Bernard, S., and Truffot, J. andDeffuant, G. (2003). MDA compliant design of SimExplorer a software tool to handle simulation experimental frameworks. In *Summer Computer Simulation Conference*, page 279–284. (Cited on page 212.)

AMD (2011). *Aparapi*. (Cited on page 153.)

Amdahl, G. M. (1967). Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference*, page 483–485. (Cited on page 122.)

Amini, M., Coelho, F., Irigoin, F., and Keryell, R. (2011). Static compilation analysis for host-accelerator communication optimization. In *The 24th International Workshop on Languages and Compilers for Parallel Computing, Fort Collins, Colorado*. (Cited on page 148.)

Andres, E. (1994). Discrete circles, rings and spheres. *Computers & Graphics*, 18(5):695–706. (Cited on page 208.)

Apache Software Foundation (2002). *Apache Maven Project*. (Cited on page 156.)

Basili, V. R., Briand, L. C., and Melo, W. L. (1996). A validation of object-oriented design metrics as quality indicators. *Software Engineering, IEEE Transactions on*, 22(10):751–761. (Cited on pages 22 and 23.)

Bauke, H. and Mertens, S. (2007). Random numbers for large-scale distributed monte carlo simulations. *Physical Review E*, 75(6):701–714. (Cited on pages 29 and 52.)

Bradley, T., Toit, J. d., Tong, R., Giles, M., and Woodhams, P. (2011). Parallelization techniques for random numbers generators. In Hwu, W.-m. W., editor, *GPU Computing Gems Emerald Edition*, pages 231–246. Elsevier. (Cited on pages 2, 40, 47 and 65.)

Brown, R. G., Eddelbuettel, D., and Bauer, D. (2009). DieHarder: a random number test suite. (Cited on page 28.)

Bugmann, H. (2001). A review of forest gap models. *Climatic Change*, 51:259–305. (Cited on page 205.)

Caux, J. (2012). *Parallélisation et optimisation d'un simulateur de morphogénèse d'organes Application aux éléments du rein.* PhD thesis, Université Blaise Pascal - École Doctorale Sciences pour l'Ingénieur. (Cited on page 24.)

Caux, J., Hill, D. R. C., Siregar, P., et al. (2011). Accelerating 3D cellular automata computation with GP-GPU in the context of integrative biology. *Cellular Automata-Innovative Modelling for Science and Engineering*, page 411–426. (Cited on page 146.)

Chafik, O. (2011a). *JavaCL.* (Cited on page 165.)

Chafik, O. (2011b). *ScalaCL.* (Cited on page 152.)

Chave, J. (1999). Study of structural, successional and spatial patterns in tropical rain forests using TROLL, a spatially explicit forest model. *Ecological Modelling*, 124:233–254. (Cited on page 205.)

Chiba, S. (1998). Javassist—a reflection-based programming wizard for java. In *Proceedings of OOPSLA'98 Workshop on Reflective Programming in C++ and Java*, page 174. (Cited on page 164.)

Coddington, P. (1996). Tests of random number generators using ising model simulations. Technical Report 34, Northeast Parallel Architecture Center. (Cited on pages 26, 52, 61, 62 and 63.)

Coddington, P. and Newell, A. (2004). JAPARA-A java parallel random number generator library for high-performance computing. In *Proceedings of the 18th International Parallel and Distributed Processing Symposium (IPDPS'04)-Workshop*, volume 5, page 156–166. (Cited on pages 47 and 101.)

Coddington, P. D. and Ko, S.-H. (1998). Techniques for empirical testing of parallel random number generators. In *Proceedings of the 12th international conference on Supercomputing*, page 282–288, Melbourne, Australia. ACM Press. (Cited on pages 10 and 28.)

Cong, G., Kodali, S., Krishnamoorthy, S., Lea, D., Saraswat, V., and Wen, T. (2008). Solving large, irregular graph problems using adaptive work-stealing. In *Parallel Processing, 2008. ICPP'08. 37th International Conference on*, page 536–545. (Cited on page 168.)

Céréghino, R., Leroy, C., Dejean, A., and Corbara, B. (2010). Ants mediate the structure of phytotelm communities in an ant-garden bromeliad. *Ecology*, 91(5):1549–1556. (Cited on page 206.)

Dagum, L. and Menon, R. (1998). OpenMP: an industry standard API for shared-memory programming. *Computational Science & Engineering, IEEE*, 5(1):46–55. (Cited on page 124.)

de Gennes, P.-G. (1988). *Scaling Concept in Polymer Physics*. IthacaNY: Cornell University Press, third edition. (Cited on pages 125 and 127.)

De Matteis, A., Eichenauer-Herrmann, J., and Grothe, H. (1992). Computation of critical distances within multiplicative congruential pseudorandom number sequences. *Journal of Computational and Applied Mathematics*, 39(1):49–55. (Cited on page 29.)

De Matteis, A. and Pagnutti, S. (1988). Parallelization of random number generators and long-range correlations. *Numerische Mathematik*, 53(5):595–608. (Cited on pages 11, 28, 29, 30, 67, 98 and 103.)

De Matteis, A. and Pagnutti, S. (1990a). A class of parallel random number generators. *Parallel Computing*, 13(2):193–198. (Cited on pages 6, 29 and 179.)

De Matteis, A. and Pagnutti, S. (1990b). Long-range correlations in linear and non-linear random number generators. *Parallel Computing*, 14(2):207–210. (Cited on pages 28, 30 and 33.)

De Matteis, A. and Pagnutti, S. (1995). Controlling correlations in parallel monte carlo. *Parallel Computing*, pages 73–84. (Cited on pages 11, 28 and 42.)

Dean, J. and Ghemawat, S. (2008). MapReduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113. (Cited on page 130.)

Dolbeau, R., Bihan, S., and Bodin, F. (2007). HMPP: a hybrid multi-core parallel programming environment. In *Workshop on General Purpose Processing on Graphics Processing Units (GPGPU 2007)*. (Cited on page 148.)

Durst, M. J. (1989). Using linear congruential generators for parallel random number generation. In *Proceedings of the 21st Winter Simulation Conference*, page 462–466, New York. ACM. (Cited on page 11.)

Eddy, W. F. (1990). Random number generators for parallel processors. *Journal of Computational and Applied Mathematics*, 31(1):63–71. (Cited on page 11.)

Eichenauer-Herrmann, J. and Grothe, H. (1989). A remark on long-range correlations in multiplicative congruential pseudo random number generators. *Numerische Mathematik*, 56(6):609–611. (Cited on page 29.)

El Bitar, Z., Lazaro, D., Coello, C., Breton, V., Hill, D. R. C., and Buvat, I. (2006). Fully 3D monte carlo image reconstruction in SPECT using functional regions. *Nuclear Instruments and Methods in Physics Research Section A: Accelerators,Spectrometers, Detectors and Associated Equipment*, 569(2):399–403. (Cited on pages 7, 31 and 74.)

Entacher, K. and Hechenleitner, B. (2003). Pitfalls when using parallel streams in OMNET++ simulations. In *Interdomain Performance and Simualtion (IPS) Workshop*, Salzburg, Austria. (Cited on pages 11 and 37.)

Entacher, K., Uhl, A., and Wegenkittl, S. (1998). Linear congruential generators for parallel monte carlo: the leap-frog case. *Monte Carlo Methods and Applications*, 4:1–16. (Cited on page 29.)

Entacher, K., Uhl, A., and Wegenkittl, S. (1999). Parallel random number generation: long-range correlations among multiple processors. In *Parallel Computation*, page 107–116. Springer. (Cited on page 11.)

Ewing, G., Pawlikowski, K., and McNickle, D. (1999). Akaroa-2: Exploiting network computing by distributing stochastic simulation. In *Proceedings of the 13th European Simulation Multiconference*, pages 175–181, Warsaw, Poland. SCSC. (Cited on page 46.)

Feng, W.-c. and Xiao, S. (2010). To GPU synchronize or not GPU synchronize? In *Circuits and Systems (ISCAS), Proceedings of 2010 IEEE International Symposium on*, page 3801–3804. (Cited on page 134.)

Ferrenberg, A., Landau, D., and Wong, Y. (1992). Monte carlo simulations: Hidden errors from "good" random number generators. *Physical Review Letters*, 69(23):3382–3384. (Cited on pages 100 and 103.)

Flynn, M. (1966). Very high-speed computing systems. *Proceedings of the IEEE*, 54(12):1901–1909. (Cited on page 16.)

Frederickson, P., Hiromoto, R., Jordan, T., Smith, B., and Warnock, T. (1984). Pseudo-random trees in monte carlo. *Parallel Computing*, 1(2):175–180. (Cited on page 34.)

Fudenberg, G. and Mirny, L. A. (2012). Higher-order chromatin structure: bridging physics and biology. *Current Opinion in Genetics & Development*, 22(2):115–124. (Cited on page 125.)

Fujimoto, R. M. (2000). *Parallel and Distributed Simulation Systems*. Wiley-Interscience, New York. (Cited on page 6.)

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1995). *Design patterns: elements of reusable object-oriented software*. Addison-wesley Reading, MA. (Cited on pages 86, 150 and 209.)

Gentle, J. E. (2003). *Random Number Generation and Monte Carlo Methods (Statistics and Computing)*. Springer, 2 edition. ISBN-13: 978-0387001784. (Cited on page 6.)

Gosling, J., Joy, B., Steele, G., and Bracha, G. (2005). *The Java language specification*. Prentice Hall, third edition edition. (Cited on page 98.)

Gourlet-Fleury, S., Cornu, G., Dessard, H., Picard, N., and Sist, P. (2004). Modelling forest dynamics for practical management purposes. *Bois et Forêts des Tropiques*, 280(2):41–52. (Cited on page 205.)

Gustafson, J. L. (1988). Reevaluating amdahl's law. *Communications of the ACM*, 31(5):532–533. (Cited on page 122.)

Haller, P. and Odersky, M. (2009). Scala actors: Unifying thread-based and event-based programming. *Theoretical Computer Science*, 410(2):202–220. (Cited on page 169.)

Haramoto, H., Matsumoto, M., Nishimura, T., Panneton, F., and LÕEcuyer, P. (2008). Efficient jump ahead for f2-linear random number generators. *INFORMS Journal on Computing*, 20(3):385–390. (Cited on pages 35, 47 and 65.)

He, B., Fang, W., Luo, Q., Govindaraju, N. K., and Wang, T. (2008). Mars: a MapReduce framework on graphics processors. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, page 260–269. (Cited on page 130.)

Hechenleitner, B. (2004). *Defects in Random Number Routines of Well-Known Network Simulators and Appropriate Improvements*. PhD thesis. (Cited on page 11.)

Hechenleitner, B. and Entacher, K. (2002). On shortcomings of the NS-2 random number generator. In Znati, T. and McDonald, B., editors, *Proceedings of the Communication Networks and Distributed Systems Modeling and Simulation*, pages 71–77. SCS. (Cited on page 30.)

Hellekalek, P. (1998a). Don't trust parallel monte carlo! In *Proceedings of Parallel and Distributed Simulation PADS98*, pages 82–89, Alberta, Canada. (Cited on pages 10, 26 and 100.)

Hellekalek, P. (1998b). Good random number generators are (not so) easy to find. *Mathematics and Computers in Simulation*, 46(5-6):485–505. (Cited on pages 6, 26, 32, 61, 100 and 179.)

Hill, D. R., Mazel, C., Passerat-Palmbach, J., and Traoré, M. K. (2013). Distribution of random streams for simulation practitioners. *Concurrency and Computation: Practice and Experience*, 25:1427–1442. doi:/10.1002/cpe.2942. (Cited on pages 26, 29, 42, 44, 52, 89, 101 and 179.)

Hill, D. R. C. (1996). *Object-oriented analysis and simulation*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA. (Cited on page 212.)

Hill, D. R. C. (1997). Object-oriented pattern for distributed simulation of large scale ecosystems. In *SCS Summer Computer Simulation Conference*, pages 945–950. (Cited on page 212.)

Hill, D. R. C. (2003). URNG: a portable optimization technique for software applications requiring pseudo-random numbers. *Simulation Modelling Practice and Theory*, 11(7-8):643–654. (Cited on pages 8 and 12.)

Hill, D. R. C. (2010). Practical distribution of random streams for stochastic high performance computing. In *IEEE International Conference on High Performance Computing & Simulation (HPCS 2010)*, pages 1–8. invited paper. (Cited on pages 83 and 220.)

Hill, M. D. and Marty, M. R. (2008). Amdahl's law in the multicore era. *Computer*, 41(7):33–38. (Cited on page 122.)

Hoberock, J. and Bell, N. (2010). *Thrust: A Parallel Template Library*. Version 1.3.0. (Cited on pages 48, 91 and 220.)

Hong, S., Kim, S. K., Oguntebi, T., and Olukotun, K. (2011). Accelerating CUDA graph algorithms at maximum warp. In *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming*, pages 267–276. ISBN: 978-1-4503-0119-0. (Cited on page 215.)

Howes, L. and Thomas, D. (2007). Efficient random number generation and application using CUDA. In *GPU Gems 3*. Addison-Wesley Professional. (Cited on page 38.)

Ising, E. (1925). Beitrag zur theorie des ferromagnetismus. *Zeitschrift für Physik A Hadrons and Nuclei*, 31(1):253–258. 10.1007/BF02980577. (Cited on pages 169, 170, 174 and 181.)

James, F. (1990). A review of pseudorandom number generators. *Computer Physics Communications*, 60(3):329–344. (Cited on page 7.)

Janowczyk, A., Chandran, S., and Aluru, S. (2008). Fast, processor-cardinality agnostic PRNG with a tracking application. In *Sixth Indian Conference on Computer Vision, Graphics & Image Processing*, page 171–178. (Cited on pages 33 and 74.)

Joshi, S. (2012). Leveraging aparapi to help improve financial java application performance. Technical report, AMD. (Cited on page 160.)

Junier, I., Dale, R. K., Hou, C., Képès, F., and Dean, A. (2012). CTCF-mediated transcriptional regulation through cell type-specific chromosome organization in the β-globin locus. *Nucleic Acids Research*. (Cited on page 125.)

Junier, I., Martin, O., and Képès, F. (2010). Spatial and topological organization of DNA chains induced by gene co-localization. *PLoS computational biology*, 6(2):e1000678. (Cited on pages 124, 126 and 134.)

Karimi, K., Dickson, N. G., and Hamze, F. (2010). *A Performance Comparison of CUDA and OpenCL*. submitted. (Cited on pages 22, 27 and 148.)

Khronos (2010). The OpenCL specification 1.1. Specification 1.1, Khronos OpenCL Working Group. (Cited on page 27.)

Khronos (2011). The OpenCL specification 1.2. Specification 1.2, Khronos OpenCL Working Group. (Cited on pages 20, 147 and 155.)

Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M., and Irwin, J. (1997). *Aspect-oriented programming*. Springer. (Cited on page 217.)

Kirk, D. B. and Hwu, W.-m. W. (2010). *Programming Massively Parallel Processors*. Morgan Kaufmann. ISBN: 978-0123814722. (Cited on pages 2, 39, 71 and 215.)

Kirschenmann, W. (2012). *Vers des noyaux de calcul intensif pérennes*. PhD thesis. (Cited on page 184.)

Kirschenmann, W., Plagne, L., and Vialle, S. (2009). Multi-target c++ implementation of parallel skeletons. In *Proceedings of the 8th workshop on Parallel/High-Performance Object-Oriented Scientific Computing*, page 7. (Cited on page 184.)

Kitching, R. (2000). *Food webs and container habitats: the natural history and ecology of phytotelmata*. Cambridge University Press. (Cited on page 206.)

Kleijnen, J. (1986). *Statistical tools for simulation practitioners*. Marcel Dekker, Inc. (Cited on page 6.)

Klöckner, A., Pinto, N., Lee, Y., Catanzaro, B., Ivanov, P., and Fasih, A. (2012). PyCUDA and PyOpenCL: a scripting-based approach to GPU run-time code generation. *Parallel Computing*, 38(3):157–174. (Cited on pages 20 and 151.)

Knuth, D. (1969). *The Art of Computer Programming*, volume 2: Seminumerical Algorithms. Addison-Wesley. (Cited on pages 7, 27, 57 and 100.)

Lagarias, J. C. (1993). Pseudorandom numbers. *Statistical Science*, 8:31–39. (Cited on page 10.)

Langdon, W. B. (2008). A fast high quality pseudo random number generator for graphics processing units. In *IEEE CEC 2008, Hong Kong*, pages 459–465. IEEE. (Cited on page 37.)

Langdon, W. B. (2009). A fast high quality pseudo random number generator for nVidia CUDA. In *GECCO'09*, volume 10, pages 2511–2514. ACM. (Cited on pages 37, 38 and 71.)

Langowski, J. and Heermann, D. (2007). Computational modeling of the chromatin fiber. *Seminars in Cell & Developmental Biology*, 18(5):659–667. (Cited on pages 124 and 125.)

Lazaro, D., El Bitar, Z., Breton, V., Hill, D. R. C., and Buvat, I. (2005). Fully 3D monte carlo reconstruction in SPECT: a feasibility study. *Physics in Medicine and Biology*, 50:3739. (Cited on page 7.)

Lea, D. (2000). A java fork/join framework. In *Proceedings of the ACM 2000 conference on Java Grande*, page 36–43. (Cited on pages 81 and 167.)

Leach, P. J., Mealling, M., and Salz, R. (2005). A universally unique identifier (UUID) URN namespace - RFC 4122. Technical report, Internet Engineering Task Force (IETF). (Cited on page 110.)

L'Ecuyer, P. (1988). Efficient and portable combined random number generators. *Communications of the ACM*, 31(6):742–751. (Cited on page 115.)

L'Ecuyer, P. (1990). Random numbers for simulation. *Communications of the ACM*, 33:85–98. (Cited on page 52.)

L'Ecuyer, P. (1996). Maximally equidistributed combined tausworthe generators. *Mathematics of computation*, 65(213):203–213. (Cited on pages 47 and 48.)

L'Ecuyer, P. (1999). Good parameters and implementations for combined multiple recursive generators. *Operations Research*, 47(1):159–164. (Cited on pages 35, 44, 80, 92, 97, 102, 104, 108 and 114.)

L'Ecuyer, P. (2001). Software for uniform random number generation: Distinguishing the good and the bad. In *Proceedings of the Winter Simulation Conference 2001*, volume 1, page 95–105. (Cited on pages 47 and 74.)

L'Ecuyer, P. (2010). Pseudorandom number generators. In *Encyclopedia of Quantitative Finance*, volume Simulation Methods in Financial Engineering, pages 1431–1437. John Wiley & Sons, Ltd, Chichester, UK, e. platen and p. jaeckel edition. (Cited on pages 6, 26, 35, 54, 83 and 100.)

L'Ecuyer, P. and Buist, E. (2005). Simulation in java with SSJ. In *Proceedings of the 37th conference on Winter simulation*, page 611–620. (Cited on pages 10, 47 and 74.)

L'Ecuyer, P. and Côté, S. (1991). Implementing a random number package with splitting facilities. *ACM Transactions on Mathematical Software (TOMS)*, 17(1):98–111. (Cited on page 101.)

L'Ecuyer, P. and Leydold, J. (2005). rstream: Streams of random numbers for stochastic simulation. Technical report, Department of Statistics and Mathematics, Abt.

f. Angewandte Statistik u. Datenverarbeitung, WU Vienna University of Economics and Business. (Cited on pages 7, 31 and 101.)

L'Ecuyer, P., Meliani, L., and Vaucher, J. (2002a). SSJ: a framework for stochastic simulation in java. In *Proceedings of the 34th Winter Simulation Conference: exploring new frontiers*, page 234–242. (Cited on pages xv, 7, 32, 35, 44, 45, 52, 96, 101 and 112.)

L'Ecuyer, P. and Simard, R. (1999). Beware of linear congruential generators with multipliers of the form a = ±2q±2r. *ACM Transactions on Mathematical Software*, 25(3):367–374. (Cited on page 58.)

L'Ecuyer, P. and Simard, R. (2007). TestU01: a c library for empirical testing of random number generators. *ACM Transactions on Mathematical Software*, 33(4):22:1–40. (Cited on pages 26, 28, 31, 32, 41, 55, 57, 58, 89, 100 and 103.)

L'Ecuyer, P. and Simard, R. (2009). *TestU01:A Software Library in ANSI C for Empirical Testing of Random Number Generators - User's guide, detailed version.* (Cited on pages 54, 55, 56 and 58.)

L'Ecuyer, P., Simard, R., Chen, E. J., and Kelton, W. D. (2002b). An object-oriented random-number package with many long streams and substreams. *Operations Research*, 50(6):1073–1075. (Cited on page 101.)

Lee, J. K. and Smith, A. J. (1984). Branch prediction strategies and branch target buffer design. *Computer*, 17(1):6–22. (Cited on page 124.)

Lee, V. W., Kim, C., Chhugani, J., Deisher, M., Kim, D., Nguyen, A. D., Satish, N., Smelyanskiy, M., Chennupaty, S., Hammarlund, P., et al. (2010). Debunking the 100X GPU vs. CPU myth: an evaluation of throughput computing on CPU and GPU. In *ACM SIGARCH Computer Architecture News*, volume 38, page 451–460. (Cited on pages 146 and 178.)

Li, Y. and Mascagni, M. (2003). Improving performance via computational replication on a large-scale computational grid. In *Proceedings of the 3st International Symposium on Cluster Computing and the Grid (CCGrid 2003)*, pages 442–448. (Cited on pages 7, 31, 46 and 74.)

Lieberman-Aiden, E., van Berkum, N. L., Williams, L., Imakaev, M., Ragoczy, T., Telling, A., Amit, I., Lajoie, B. R., Sabo, P. J., Dorschner, M. O., et al. (2009). Comprehensive mapping of long-range interactions reveals folding principles of the human genome. *science*, 326(5950):289–293. (Cited on page 123.)

Maigne, L., Hill, D. R. C., Calvat, P., Breton, V., Reuillon, R., Legre, Y., and Donnarieix, D. (2004). Parallelization of monte carlo simulations and submission to a grid environment. *Parallel processing letters*, 14(2):177–196. (Cited on pages 1, 7, 12, 31, 74 and 103.)

Marsaglia, G. (1985). A current view of random number generators. In *Computer Science and Statistics, Sixteenth Symposium on the Interface*, pages 3–10. Elsevier. (Cited on page 56.)

Marsaglia, G. (2003). Xorshift RNGs. *Journal of Statistical Software*, 8(14):1–6. (Cited on page 47.)

Marsaglia, G., Narasimhan, B., and Zaman, A. (1990). A random number generator for PC's. *Computer Physics Communications*, 60(3):345–349. (Cited on pages 27 and 48.)

Marsaglia, G. and Zaman, A. (1991). A new class of random number generators. *Annals of Applied Probability*, 3(3):462–480. (Cited on page 48.)

Mascagni, M. (1998). High-performance monte carlo tools. *IEEE Computational Science & Engineering*, 5(2):97–98. (Cited on pages 29, 33 and 46.)

Mascagni, M. (1999). Some methods of parallel pseudorandom number generation. *IMA Volumes in Mathematics and Its Applications*, 105:277–288. (Cited on page 52.)

Mascagni, M. and Chi, H. (2004). Parallel linear congruential generators with sophie-germain moduli. *Parallel Computing*, 30(11):1217–1231. (Cited on pages 7 and 33.)

Mascagni, M. and Srinivasan, A. (2000). Algorithm 806: SPRNG: a scalable library for pseudorandom number generation. *ACM Transactions on Mathematical Software (TOMS)*, 26(3):436–461. (Cited on page 46.)

Mascagni, M. and Srinivasan, A. (2004). Parameterizing parallel multiplicative lagged-fibonacci generators. *Parallel Computing*, 30(7):899–916. (Cited on pages 46 and 74.)

Matsumoto, M. and Nishimura, T. (1998). Mersenne twister: A 623-dimensionnally equidistributed uniform pseudorandom number generator. *ACM Transactions on Modeling and Computer Simulations: Special Issue on Uniform Random Number Generation*, 8(1):3–30. (Cited on pages 10, 12, 41, 52, 59 and 101.)

Matsumoto, M. and Nishimura, T. (2000). Dynamic creation of pseudorandom number generators. In Niederreiter, H. and Spanier, J., editors, *Monte Carlo and Quasi-Monte Carlo Methods 1998*, pages 56–69. Springer. (Cited on pages 28, 33, 46, 54 and 59.)

McConnell, S. (2004). *Code Complete: A Practical Handbook of Software Construction*. Microsoft Press. (Cited on pages 22 and 23.)

Metropolis, N., Rosenbluth, A., Rosenbluth, M., Teller, A., Teller, E., et al. (1953). Equation of state calculations by fast computing machines. *The journal of chemical physics*, 21(6):1087. (Cited on page 170.)

Meyer, B. (1988). *Object-Oriented Software Construction*. Prentice hall New York, 1st edition. (Cited on page 23.)

Meyers, S. (2005). *Effective C++: 55 Specific Ways to Improve Your Programs and Designs*. Addison-Wesley, 3rd edition. (Cited on page 23.)

Miller, G. A. (1956). The magical number seven, plus or minus two: Some limits on our capacity for processing information. *Psychological review*, 63(2):81. (Cited on page 23.)

MPI Forum (1993). Document for a standard message-passing interface. Technical report, University of Tennessee. (Cited on page 124.)

Nickolls, J. and Dally, W. J. (2010). The GPU computing era. *IEEE Micro*, pages 56–69. (Cited on page 13.)

Niederreiter, H. (1992). *Quasi-Monte Carlo Methods*. Wiley Online Library. (Cited on page 7.)

Nokia (2010). *QtOpenCL*. (Cited on page 150.)

NVIDIA (2010a). *NVIDIA CUDA Best Practices Guide Version 3.0*. (Cited on page 47.)

NVIDIA (2010b). *NVIDIA CUDA Programming Guide Version 3.2*. (Cited on pages 37, 39 and 71.)

NVIDIA (2011). *NVIDIA CUDA Programming Guide Version 4.0*. (Cited on pages xv, 12, 16, 18 and 216.)

NVIDIA (2012). NVIDIA's next generation CUDA compute architecture: Kepler GK110. Technical report. (Cited on pages 15, 91 and 139.)

Odersky, M., Spoon, L., and Venners, B. (2008). *Programming in Scala*. Artima Incorporated. (Cited on page 169.)

Okasaki, C. (1999). *Purely functional data structures*. Cambridge Univ Pr. (Cited on page 167.)

Orivel, J. and Leroy, C. (2011). The diversity and ecology of ant gardens (hymenoptera: Formicidae; spermatophyta: Angiospermae). *Myrmecological News*, 14:73–85. (Cited on page 205.)

Panneton, F. (2004). *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*. PhD thesis, Département d'informatique et de recherche opérationnelle - Faculté des arts et des sciences - Université de Montréal. (Cited on pages 10 and 58.)

Panneton, F., L'Ecuyer, P., and Matsumoto, M. (2006). Improved long-period generators based on linear reccurences modulo 2. *ACM Transactions on Mathematical Software*, 32(1):1–16. (Cited on pages 47 and 52.)

Papakonstantinou, A., Gururaj, K., Stratton, J., Chen, D., Cong, J., and Hwu, W. (2009). FCUDA: enabling efficient compilation of CUDA kernels onto FPGAs. In *IEEE 7th Symposium on Application Specific Processors, 2009. SASP'09.*, page 35–42. (Cited on page 148.)

Park, S. K. and Miller, K. W. (1988). Random number generators: Good ones are hard to find. *Communications of the ACM*, 31(10):1192–1201. (Cited on pages 37 and 52.)

Passerat-Palmbach, J., Caux, J., Pennec, Y. L., Reuillon, R., Junier, I., Kepes, F., and Hill, D. R. C. (2013a). Parallel stepwise stochastic simulation: Harnessing GPUs to explore possible futures states of a chromosome folding model thanks to the possible futures algorithm (PFA). In *ACM SIGSIM Conference on Principles of Advanced Discrete Simulation (PADS)*, pages 169–177, Montreal, Canada. (Cited on page 181.)

Passerat-Palmbach, J., Caux, J., Schweitzer, P., Siregar, P., Mazel, C., and Hill, D. R. C. (2013b). Harnessing aspect oriented programming on GPU: application to warp-level parallelism (WLP). *The International Journal of Computer Aided Engineering and Technology*, page submitted. under review. (Cited on page 211.)

Passerat-Palmbach, J., Caux, J., Siregar, P., and Hill, D. R. C. (2011a). Warp-level parallelism: Enabling multiple replications in parallel on GPU. In *Proceedings of*

the European Simulation and Modeling Conference 2011, pages 76–83. ISBN: 978-90-77381-66-3 *(best paper award)*. (Cited on pages 72, 80, 92, 180 and 211.)

Passerat-Palmbach, J., Forest, A., Pal, J., Corbara, B., and Hill, D. R. C. (2012a). Automatic parallelization of a gap model using java and OpenCL. In *Proceedings of the European SImulation and Modeling Conference (ESM)*, pages 24–31. (Cited on pages 89, 160, 165, 169, 174, 181 and 205.)

Passerat-Palmbach, J., Mazel, C., Bachelet, B., and Hill, D. R. C. (2011b). ShoveRand: a model-driven framework to easily generate random numbers on GP-GPU. In *IEEE International Conference on High Performance Computing & Simulation*, pages 41–48. IEEE. (Cited on page 48.)

Passerat-Palmbach, J., Mazel, C., and Hill, D. R. C. (2012b). Pseudo-random streams for distributed and parallel stochastic simulations on GP-GPU. *Journal of Simulation*, 6(3):141–151. doi:10.1057/jos.2012.8. (Cited on page 179.)

Passerat-Palmbach, J., Mazel, C., and Hill, D. R. C. (2012c). ThreadLocalMRG32k3a: a statistically sound substitute to pseudorandom number generation in parallel java applications. In *Proceedings of the IEEE High Performance Computing and Simulation conference*, pages 543–550. *(nominated for the outstanding paper award)*. (Cited on pages 81 and 100.)

Passerat-Palmbach, J., Mazel, C., and Hill, D. R. C. (2013c). TaskLocalRandom: a statistically sound substitute to pseudorandom number generation in parallel java tasks frameworks. *Concurrency and Computation: Practice and Experience*, page submitted. under review. (Cited on pages 81 and 180.)

Passerat-Palmbach, J., Mazel, C., Mahul, A., and Hill, D. R. C. (2010). Reliable initialization of GPU-enabled parallel stochastic simulations using mersenne twister for graphics processors. In *Europeans Simulation and Modeling Conference 2010*, pages 187–195. ISBN: 978-90-77381-57-1. (Cited on pages 47, 53, 65, 69 and 98.)

Passerat-Palmbach, J., Mazel, C., Reuillon, R., and Hill, D. R. C. (2013d). Automatic parallelization of simulations on manycore architectures using scala: A case study. to be published. (Cited on page 181.)

Pawlikowski, K. (2003a). Do not trust all simulation studies of telecommunication networks. In *Information Networking*, page 899–908. (Cited on page 46.)

Pawlikowski, K. (2003b). Towards credible and fast quantitative stochastic simulation. In *Proceedings of International SCS Conference on Design, Analysis and Simulation of Distributed Systems, DASD*, volume 3. (Cited on pages 11 and 212.)

Pawlikowski, K., Jeong, H., and Lee, J. (2002). On credibility of simulation studies of telecommunication networks. *IEEE Communications Magazine*, 40(1):132–139. (Cited on page 46.)

Pawlikowski, K. and McNickle, D. (2001). Speeding up stochastic discrete-event simulation. In *Proc. European Simulation Symposium, ESS'01*, volume 1, page 132–138. (Cited on pages 6 and 179.)

Pawlikowski, K. and Yau, V. (1992). On automatic partitioning, run-time control and output analysis methodology for massively parallel simulations. In *Proceedings of the European Simulation Symposium, ESS'92*, pages 135–139. (Cited on pages 6, 11, 46 and 179.)

Pawlikowski, K., Yau, V., and McNickle, D. (1994). Distributed stochastic discrete-event simulation in parallel time streams. In *Proceedings of the 26th conference on Winter simulation*, page 723–730. (Cited on page 212.)

Percus, O. and Kalos, M. (1989). Random number generators for MIMD parallel processors. *Journal of Parallel and Distributed Computing*, 6(3):477–497. (Cited on pages 11 and 34.)

Podlozhnyuk, V. (2007). Parallel mersenne twister. Technical report, NVIDIA. (Cited on page 41.)

Pratt-Szeliga, P. C., Fawcett, J. W., and Welch, R. D. (2012). Rootbeer: Seamlessly using GPUs from java. In *2012 IEEE 14th International Conference on High Performance Computing and Communication & 2012 IEEE 9th International Conference on Embedded Software and Systems (HPCC-ICESS)*, page 375–380. (Cited on pages 148, 164 and 184.)

Preis, T., Virnau, P., Paul, W., and Schneider, J. (2009). GPU accelerated monte carlo simulation of the 2D and 3D ising model. *Journal of Computational Physics*, 228(12):4468–4477. (Cited on page 171.)

Prokopec, A., Bagwell, P., Rompf, T., and Odersky, M. (2011). A generic parallel collection framework. In *Euro-Par 2011 Parallel Processing*, page 136–147. Springer. (Cited on page 167.)

Reith, D., Mirny, L., and Virnau, P. (2011). GPU based molecular dynamics simulations of polymer rings in concentrated solution: Structure and scaling. *Progress of Theoretical Physics Supplement*, 191:135–145. (Cited on page 124.)

Reuillon, R. (2008a). *Simulations stochastiques en environnements distribués - Application aux grilles de calcul*. PhD thesis, Université Blaise Pascal - École Doctorale Sciences pour l'Ingénieur. (Cited on pages 7, 30 and 73.)

Reuillon, R. (2008b). Testing 65536 parallel pseudo-random number streams. In *EGEE Grid User Forum 2008*. Poster. (Cited on pages 41, 101, 103 and 181.)

Reuillon, R., Chuffart, F., Leclaire, M., Faure, T., Dumoulin, N., and Hill, D. R. C. (2010). Declarative task delegation in OpenMOLE. In Smari, W. W., editor, *Proceedings of the 2010 International Conference on High Performance Computing and Simulation*, pages 55–62. (Cited on page 74.)

Reuillon, R., Hill, D. R. C., El Bitar, Z., and Breton, V. (2008). Rigorous distribution of stochastic simulations using the DistMeToolkit. *IEEE Transactions on Nuclear Science*, 55(1):595–603. (Cited on page 77.)

Reuillon, R., Leclaire, M., and Rey-Coyrehourcq, S. (2013). OpenMOLE, a workflow engine specifically tailored for the distributed exploration of simulation models. *Future Generation Computer Systems*, 29(8):1981–1990. (Cited on page 74.)

Reuillon, R., Traore, M. K., Passerat-Palmbach, J., and Hill, D. R. (2011). Parallel stochastic simulations with rigorous distribution of pseudo-random numbers with DistMe: application to life science simulations. *Concurrency and Computation: Practice and Experience*, 24(7):723–738. (Cited on pages 26, 28, 74 and 101.)

Rukhin, A., Soto, J., Nechvatal, J., Smid, M., Barker, E., Leigh, S., Levenson, M., Vangel, M., Banks, D., Heckert, A., Dray, J., and Vo, S. (2001). A statistical test suite for random and pseudorandom number generators for cryptographic applications. Technical report, NIST. (Cited on page 26.)

Ryoo, S., Rodrigues, C. I., Baghsorkhi, S. S., Stone, S. S., Kirk, D. B., and Hwu, W.-m. W. (2008). Optimization principles and application performance evaluation of a multithreaded GPU using CUDA. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, page 73–82. (Cited on page 133.)

Rütti, M. (2004). *A random number generator test suite for the C++ standard*. PhD thesis, Diploma Thesis. (Cited on page 28.)

Rütti, M., Troyer, M., and Petersen, W. (2004). A generic random number generator test suite. *Arxiv preprint math/0410385*. (Cited on page 28.)

Saito, M. (2011). Tiny mersenne twister (TinyMT): a small-sized variant of mersenne twister. (Cited on pages 44 and 65.)

Saito, M. and Matsumoto, M. (2008). SIMD-oriented fast mersenne twister: a 128-bit pseudorandom number generator. In Keller, A., Heinrich, S., and Niederreiter, H., editors, *Monte Carlo and Quasi-Monte Carlo Methods 2006*, volume 2, pages 607–622. Springer Berlin Heidelberg. (Cited on pages 10, 12, 35, 38 and 52.)

Saito, M. and Matsumoto, M. (2012). A deviation of CURAND: standard pseudorandom number generator in CUDA for GPGPU. presentation. (Cited on page 92.)

Saito, M. and Matsumoto, M. (2013). A variant of mersenne twister suitable for graphics processors. *ACM Transactions on Mathematical Software*, 39(2). (Cited on pages 12, 39, 40, 41, 52, 83 and 96.)

Salmon, J. K., Moraes, M. A., Dror, R. O., and Shaw, D. E. (2011). Parallel random numbers: as easy as 1, 2, 3. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, page 16:1–16:12, Seattle, Washington. ACM. (Cited on pages 36, 46, 89, 100, 103 and 118.)

Scarpino, M. (2011). *OpenCL in Action*. Manning Publications, Shelter Island, NY. (Cited on pages xxi and 150.)

Schelling, T. C. (1971). Dynamic models of segregation. *The Journal of Mathematical Sociology*, 1(2):143–186. (Cited on pages 169, 170, 174 and 181.)

Schweitzer, P., Mazel, C., Fehr, F., Carloganu, C., and Hill, D. R. C. (2013). Proper parallel monte carlo for computed tomography of volcanoes. In *Proceedings of the 2013 International Conference on High Performance Computing & Simulation*, pages 519–526, Helsinki, Finland. IEEE. (Cited on page 31.)

Siek, J. and Lumsdaine, A. (2000). Concept checking: Binding parametric polymorphism in c++. In *First Workshop on C++ Template Programming, Erfurt, Germany*. (Cited on page 87.)

Sipper, M. and Tomassini, M. (1996). Generating parallel random number generators by cellular programming. *International Journal of Modern Physics C*, 7(2):181–190. (Cited on page 34.)

Smith, J. E. (1998). A study of branch prediction strategies. In *25 years of the international symposia on Computer architecture (selected papers)*, page 202–215. (Cited on page 124.)

Sommerville, I. (2010). *Software Engineering*. Addison-Wesley, 9th edition. (Cited on page 22.)

Spinczyk, O., Gal, A., and Schröder-Preikschat, W. (2002). AspectC++: an aspect-oriented extension to the c++ programming language. In *Proceedings of the Fortieth International Conference on Tools Pacific: Objects for internet, mobile and embedded applications*, page 53–60. (Cited on page 217.)

Srinivasan, A. (1998). Introduction to parallel RNGs. (Cited on page 30.)

Srinivasan, A., Mascagni, M., and Ceperley, D. (2003). Testing parallel random number generators. *Parallel Computing*, 29(1):69–94. (Cited on page 11.)

Stevanovic, R., Topic, G., Skala, K., Stipcevic, M., and Rogina, B. M. (2008). Quantum random bit generator service for monte carlo and other stochastic simulations. *Lecture Notes in Computer Science*, 4818:508–515. (Cited on page 8.)

Strick, T. R., Dessinges, M.-N., Charvin, G., Dekker, N. H., Allemand, J.-F., Bensimon, D., and Croquette, V. (2003). Stretching of macromolecules and proteins. *Reports on Progress in Physics*, 66:1. (Cited on page 125.)

Sussman, M., Crutchfield, W., and Papakipos, M. (2006). Pseudorandom number generation on the GPU. In *Proceedings of the 21st ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware*, page 87–94. (Cited on pages 37 and 38.)

Tausworthe, R. (1965). Random numbers generated by linear recurrence modulo two. *Mathematics of Computation*, 19(90):201–209. (Cited on page 48.)

Tomassini, M. (1999). Parallel and distributed evolutionary algorithms: A review. (Cited on page 34.)

Tomassini, M., Sipper, M., Zolla, M., and Perrenoud, M. (1999). Generating high-quality random numbers in parallel by cellular automata. *Future Generation Computer Systems*, 16(2-3):291–305. (Cited on page 34.)

Topa, P. and Młocek, P. (2012). GPGPU implementation of cellular automata model of water flow. *Parallel Processing and Applied Mathematics*, page 630–639. (Cited on page 146.)

Touraille, L., Traoré, M. K., Hill, D. R. C., et al. (2010). Enhancing DEVS simulation through template metaprogramming. In *2010 Summer Simulation Multiconference*, pages 394–402. (Cited on page 24.)

Traoré, M. K. and Hill, D. R. C. (2001). The use of random number generation for stochastic distributed simulation: application to ecological modeling. In *Proceedings of the 13th European Simulation Symposium*, page 555–559, Marseille. (Cited on pages 26, 29 and 52.)

Vallée-Rai, R., Co, P., Gagnon, E., Hendren, L., Lam, P., and Sundaresan, V. (2010). Soot: A java bytecode optimization framework. In *CASCON First Decade High Impact Papers*, page 214–224. (Cited on page 164.)

Van Deursen, A., Klint, P., and Visser, J. (2000). Domain-specific languages: An annotated bibliography. *Sigplan Notices*, 35(6):26–36. (Cited on page 3.)

Vattulainen, I. and Ala-Nissila, T. (1995). Mission impossible: Find a random pseudo-random number generator. *Computers in Physics*, 9(5):500–510. (Cited on page 221.)

Wittenbrink, C., Kilgariff, E., and Prabhu, A. (2011). Fermi GF100 GPU architecture. *IEEE Micro*, 31(2):50–59. (Cited on page 14.)

Wu, P. and Huang, K. (2006). Parallel use of multiplicative congruential random number generators. *Computer Physics Communications*, 175(1):25–29. (Cited on pages 31, 32 and 33.)

Xiao, S. and Feng, W.-c. (2010). Inter-block GPU communication via fast barrier synchronization. In *Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, page 1–12. (Cited on page 134.)

Zhmurov, A., Rybnikov, K., Kholodov, Y., and Barsegov, V. (2010). Efficient pseudo-random number generators for biomolecular simulations on graphics processors. Technical report, CERN. (Cited on pages 38 and 71.)

Appendix A

A Simple Guiana Rainforest Gap Model

This model was developed in collaboration with Dr. Bruno Corbara and the help of two graduate students: Arthur Forest and Julien Pal. It has been published at the European Simulation and Modelling (ESM) conference in 2012 [Passerat-Palmbach et al., 2012a].

A.1 Introduction

Gap models study the dynamics of forests, and particularly the trees that fall, and progressively regrow over the years. Considering a precise square area of the forest, we will look at the gap dynamics (their appearance and progressive disappearance). Gaps are the "empty" areas formed by fallen trees and the neighbours they have carried away while collapsing. Gap models are among the most widely used in forest modelling. While studies tackling forest dynamics are legion [Acevedo et al., 1995; Chave, 1999; Bugmann, 2001; Gourlet-Fleury et al., 2004], for this simulation we have chosen to stick with a simplified gap model, with emphasis on the resulting light distribution. This model is intended to become the base of a more ambitious multi-scale multi-agent simulation model. This is why we interested in its parallelization in Chapter 6.3, so that it does not slow down the future simulation.

The simulation model that we aim at implementing represents the long term dynamics of Ant Gardens (AGs) and of some of their inhabitants in a tropical rainforest. An AG is a complex arboreal suspended structure including an ant nest and symbiotic plants growing on it. In French Guiana, AGs are initiated by different species of ants that incorporate the plant seeds in the humus-rich carton of their nest [Orivel and Leroy, 2011]. AGs are installed on a more or less sun-exposed site depending on the light preferences of ant species. Among the plants growing on AGs is a water-holding one (a tank-plant from the bromeliad family) that harbours various aquatic organisms

from microorganisms to vertebrates (batrachian tadpoles) among which many insect larvae. The tanks of these bromeliads harbour different communities depending on the ant species inhabiting their resident AG, partly due to canopy openness and resulting incident light [Céréghino et al., 2010].

The whole AG model will help us study the consequences of human activities on the forest. Indeed, when Man builds a road, the latter creates a huge edge to a degree comparable to an artificial linear gap. The edge effect that results is very important. Man encourages the development of species accustomed to this type of environment. The anthropic action may favour particular ants and therefore particular aquatic insect larvae. Many studies (see the survey in [Kitching, 2000]) have shown that ecosystems that develop in bromeliads are home to many mosquito larvae. In French Guiana, some species of mosquitoes are vectors of "dengue" (due to arboviruses; some as dengue haemorrhagic fever are deadly if left untreated). The potential danger of such uncontrolled proliferation is evident, which explains the value of studying such a configuration by simulation.

A.2 Model Description

Now, let us consider the gap model at the heart of this study. According to data provided by domain specialists, we know that in some studied area in French Guiana, 33 gaps per year appear over a $300m^2$ area, on average. This size can range from $20m^2$ to $20000m^2$, and it takes 20 to 25 years for a gap to structurally close completely, and this according to an exponential decay law. Indeed, in the early years, many young seedlings are taking advantage of the sunlight reaching the ground, and gradually as the trees grow, they close the gap surface, and reduce the amount of light reaching the ground. Finally, according to measurements made on two sites, 1.1% of forest area falls every year on the first site and 1.3% on the second. This means that statistically, half the forest area is affected by gaps in 69 years, and nearly all its surface (99%) is in 400 years.

In addition to the gaps dynamics, in our model we had to consider the intensity of the incident light over time, as it is directly related to the area exposed by gaps. The light model is recomputed annually, as it evolves with the gaps. To represent the illumination at a point, we must take into account both the direct exposure when it belongs to a gap, but also the gradient of light scattered from different points of the simulated area.

A UML class diagram of our model is presented in Figure A.1.

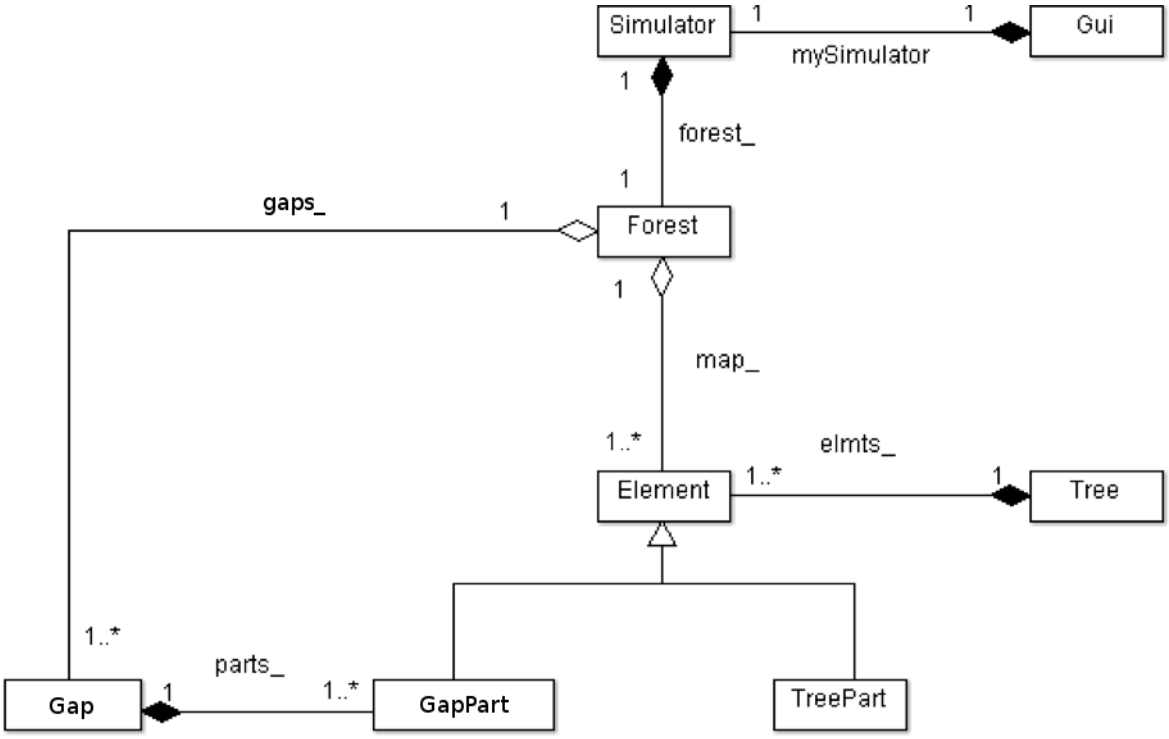


Figure A.1: UML class diagram of the gap model

A.2.1 Making trees fall

A fall starts with one tree that can tumble down in eight different directions drawn randomly. These eight possibilities can be represented by three different fall shapes as shown in Figure A.2. From these shapes, another random draw occurs to decide in which way the tree falls.

The coloured cells represent the space occupied by the parts of the tree on the ground. When trees fall, these parts are considered as parts of a windthrow.

Now, when a tree falls it affects its neighbours that can be themselves dragged down in the fall. Falling neighbours are governed by several rules: the size of the windthrow generated by the neighbours must be lower than the size determined for the original windthrow. Moreover, the neighbour tree which falls can not fall in any direction, but coherently to the way its parent is falling. This process can be repeated again and again in order to make neighbours of neighbours falls, provided the initial size picked up for the windthrow has not been reached.

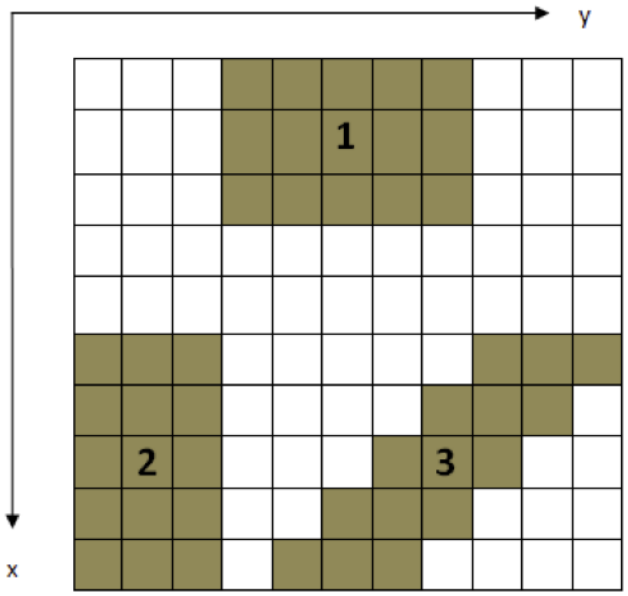


Figure A.2: Three different fall shapes for trees

A.2.2 Light-based tree regrowth

The sunlight actually attaining the ground impacts the way trees regrow in windthrows. In order to represent the path of the sun in our model, we observed the amount of light reaching the ground during a day. What appears is that the central area of a windthrow is constantly lit, as no tree can shade the ground. Then, the further a spot on the ground is from the centre of the windthrow, the fewer sunlight it gets during a day. We consequently decided to represent the light on the ground of each windthrow as a target. This target is divided into several concentric circles of different colours. Each circle represents a different level of lighting, decreasing from the centre circle to the circle on the edge.

Circles are drawn using the Andres's algorithm [Andres, 1994], allowing to draw several consecutive circles without leaving holes in the resulting full circle they compose. We chose the colour red for the most lit areas and the yellow colour for areas least affected by light. Figure A.3 is a screenshot of the simulation GUI where the light circles appear over windthrows.

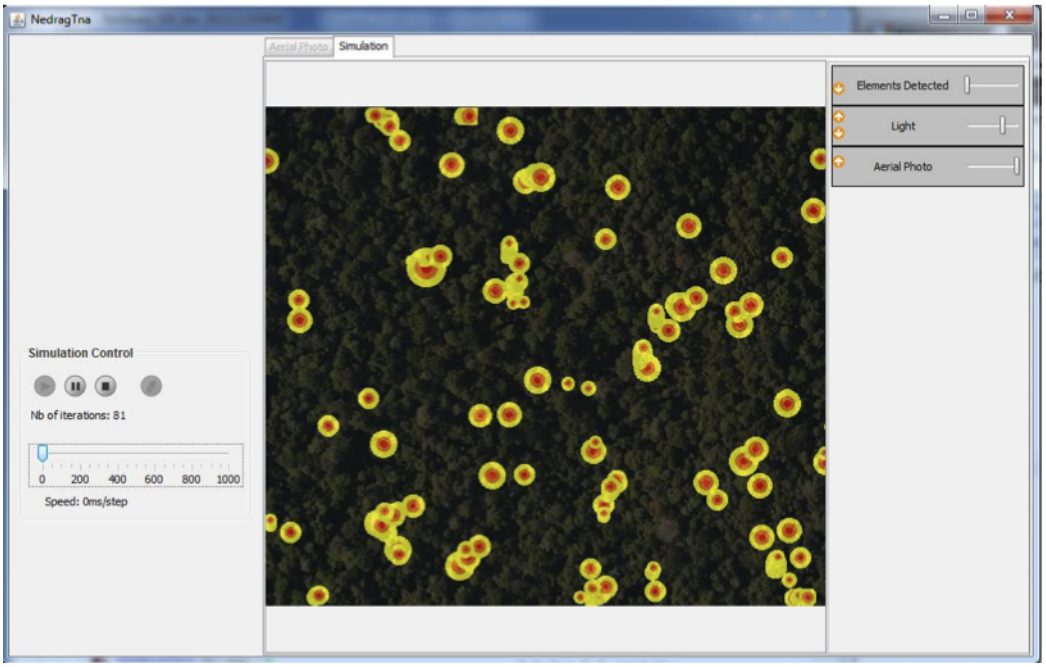


Figure A.3: Concentric light circles determine the amount of sunlight reaching the ground in windthrows

A.2.3 Closing windthrows thanks to the sunlight model

The concentric circles representing light describe three levels of lighting. These different levels have now to be taken into account in the regrowth of trees. As the level of sunlight reaching the ground is the highest at the centre of the windthrow, trees spawning in this area will grow faster than the others. Thus, windthrows refill from their center to their edges. As the years go by, all the different areas of the windthrow get refilled until total closure of the gap in the forest.

More precisely, we have defined a repopulation rate picked up at random between 5 and 10 years. Each year, the model determines whether current age of the windthrow corresponds to a year of repopulation. In such a case, the windthrow is repopulated accordingly to the rate previously determined.

This process based upon lighting rules the regrowth of windthrow younger than 20 years old. Older windthrows use a different repopulation scheme, from the edges to the centre.

The problem of different repopulation methods fits perfectly the strategy design pattern [Gamma et al., 1995]. In this model, windthrows take 100 years to fully

repopulate and thus close completely. We have created a different strategy per age group, and windthrows delegate their ageing to the method offered by the strategy algorithm they are currently associated with. In doing so, we can apply a specific ageing and repopulation process to windthrows according to their age.

A.3 Conclusion

The purpose of this Gap Model is to serve as a basis for a future model studying the long term dynamics of Ant Gardens. Our model is based upon the appearance of gaps in the forest due to tree falls. Even falling trees are randomly picked up, the total area of fallen trees follows the tendency of data provided by specialists. Gap areas are repopulated with new trees, growing at different speeds depending on their location in the gap, also called windthrow. Actually, this location impacts the amount of light received per day, and thus, the rate at which trees grow. The combination of trees falling and repopulating gaps forms the dynamics of the forest area that are studied by this model.

Appendix B

Harnessing Aspect Oriented Programming on GPU: Application to Warp-Level Parallelism (WLP)

This appendix is an extension of the paper [Passerat-Palmbach et al., 2011a], which was awarded as the best paper of the European Simulation and Modelling (ESM) conference in 2011. The present version is currently under review to be published in the International Journal of Computer Aided Engineering and Technology (IJCAET) [Passerat-Palmbach et al., 2013b].

B.1 Introduction

Replications are a widespread method to obtain confidence intervals for stochastic simulation results. It consists in running the same stochastic simulation with different random sources and averaging the results. According to the Central Limit Theorem, the average result is approximated in an accurate enough way by a Gaussian Law, for a number of replications greater than 30. Thus, for a number of replications greater than 30, we can obtain a confidence interval with a satisfactory precision. This average result depends on the stochastic variability of the application. Some applications can settle for fewer replications, but are consequently less keen to take advantage of Multiple Replication in Parallel (MRIP).

There are many cases where a single simulation can last for a while, so 30 of them run sequentially may represent a very long computation time. Because of this overhead, 30 replications are hardly run in most simulations. Instead, a good practice is often to run 3 replications when debugging, and 10 replications are commonly used to compute a confidence interval. To maintain an acceptable computation time while running 30 or more replications, many scientists proposed to run in parallel these independent simulations. This approach has been named Multiple Replication in Parallel

in the nineties [Pawlikowski et al., 1994]. As its name suggests, its main idea is to run each replication in parallell [Hill, 1997; Pawlikowski, 2003b]. In addition, when we explore an experimental plan we have to run different sets of replications, with different factor levels according to the experimental framework [Hill, 1996; Amblard et al., 2003]. In this study, we will not consider any constraints that need to be satisfied when implementing MRIP. One of the main barriers that often prevents simulationists to achieve a decent amount of replications is the lack of knowledge in the parallelization techniques. Another common hindrance is the amount of parallel computing facilities available. Our work tackles this problem by introducing a way to harness the computational power of GPUs (Graphics Processing Units) to process MRIP or DOEs (Design of Experiments) faster than on a scalar CPU (Central Processing Unit).

GPUs deliver such an overwhelming power at a low cost that they now play an important role in the High Performance Computing world. However, this kind of devices display major constraints, tied to its intrinsic architecture. Basically, GPUs have been designed to deal with computation intensive applications such as image processing. One of their well-known limits are the slow memory accesses. Indeed, since GPUs are designed to be efficient at computation, they badly cope with applications frequently accessing memory. Except by choosing the right applications, the only thing we can do to overcome this drawback is to wait for the hardware to evolve in such a way. Since the NVIDIA generation codenamed Fermi, GPUs have shown a move in this way by improving cache memories available on the GPU. This leads to better performances for most applications at no development cost, only by replacing the old hardware by the state-of-the-art one (Kepler at the time of writing).

Now, what we can actually think about is the way we program GPUs. Whatever the programming language or architecture chosen to develop an application with, CUDA (Compute Unified Device Architecture) or OpenCL, the underlying paradigm is the same: SIMT (Single Instruction, Multiple Threads). Thus, applications are tuned to exploit the hardware configuration, which is a particular kind of SIMD architecture (Single Instruction, Multiple Data). To obtain speed-ups, parallel applications must be implemented in an SIMD-compliant way. This point reduces the scope of GPU-enabled applications.

The SIMT paradigm automatically groups into 32-wide bundles called warps. Warps are the base unit used to schedule both computation on Arithmetic and Logic Units (ALUs) and memory accesses. Threads within the same warp follow the SIMD pattern, i.e. they are supposed to execute the same operation at a given clock cycle. If they do not, a different execution branch is created and executed sequentially every time

a thread needs to compute differently from its neighbours. The latter phenomenon is called branch divergence, and leads to significant performance drops. However, threads contained in different warps do not suffer the same constraint. They are executed independently, since they belong to different warps.

This appendix describes Warp-Level Parallelism (WLP), a paradigm to evaluate the approach of using GPUs to compute MRIP, using an independent warp for each replication. In order to make it easier to use, we introduce an Aspect Oriented Programming (AOP) declination of WLP using an handcrafted preprocessor. As a matter of fact, this appendix also details the different solutions to couple AOP with CUDA. It shows what AOP can bring to CUDA-enabled applications, in terms of software engineering and extensibility. In the further sections, we will:

- Describe a mechanism to run MRIP on GPU;
- Propose an implementation of our approach: WLP;
- Introduce the different approaches to implement AOP and those compatible with CUDA;
- Detail the AOP version of WLP;
- Benchmark WLP with three different simulation models.

B.2 A Warp Mechanism to Speed Up Replications

Two problems arise when trying to port replications to GPU threads, considering a replication per thread. First, we generally compute few replications, whereas we have seen that GPUs needed to achieve large amounts of computations to hide their memory latency. Second, replications of stochastic simulations are not renowned for their SIMD-friendly behaviour. Usually, replications fed with different random sources will draw different random numbers at the same point of the execution. If a condition result is based on this draw, divergent execution paths are likely to appear, forcing threads within a same warp to be executed sequentially because of the intrinsic properties of the device.

The idea that we propose in this work is to take advantage of the previously introduced warp mechanism to enable fast replications of a simulation. Instead of having to deal with Thread-Level Parallelism (TLP) and its constraints mentioned above, we place ourselves at a slightly higher scope to manipulate warps only. Let this paradigm be called Warp-Level Parallelism (WLP), as opposed to TLP. Now running only one

replication per warp, it is possible to have each replication to execute different instructions without being faced to the branch divergence problem.

But to successfully enable easy development of simulation replications on GPU using one thread per warp, two mechanisms are needed.

First, it is necessary to restrict each warp to use only one valid thread. By doing so, we ensure not to have divergent paths within a warp. Moreover, we artificially increase the device's occupancy, and consequently, we take advantage of the quick context switching between warps to hide slow memory accesses. Theoretically, we should use the lowest block size maximizing occupancy. For instance, a C2050 board owns 14 Streaming Multiprocessors (SMs), and can schedule at most 8 blocks per SM. In this case, the optimal block size when running 50 replications would be 32 threads per block. This situation is represented in Figure B.1, where we can see two warps running their respective first threads only. The 31 remaining threads are disabled, and will stall until the end of the kernel. Unfortunately, the GigaThread scheduler, introduced in the previous section, does not always enable a kernel to run on every available SM. In addition, memory constraints applying to SMs might compromise this ideal case by reducing the number of available blocks per SM.

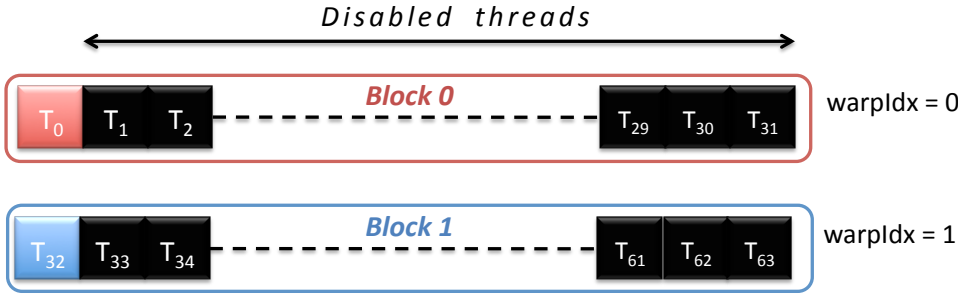


Figure B.1: Representation of thread disabling to place the application at a warp-level

Second, there has to be an easy solution to get a unique index for each warp. TLP relies essentially on threads identifiers to retrieve or write data back. Thus, WLP needs to propose an equivalent mechanism so that warps can be distinguished to access and compute their own data.

Thanks to the two tools introduced in this section, it is possible to create a kernel where only one thread per warp will be valid, and where it will be easy to make each valid thread compute different instructions, or work on different data depending on the new index.

Although we could not figure out the real behaviour of the GigaThread Engine

dispatcher, the characteristics noticed in this part are sufficient to evaluate the performance of the dispatching policy. Furthermore, the new scheduling features introduced of NVIDIA GPUs benefit the overall performance of our warp-based approach, given that it highly relies on warp scheduling and block dispatching.

B.3 Implementation

Now that we have defined our solution, we will propose an implementation in this section. To do so, we need to focus on two major constraints: first, we should keep a syntax close to C++ and CUDA, so that users are not confused when they use our approach. Second, we need to propose compile-time mechanisms as much as possible. Indeed, since WLP only exploits a restricted amount of the device's processing units, we have tried to avoid any overhead implied by our paradigm.

This study intends to prove that our approach is up and running. Thus, this section will only introduce a restricted number of keywords used by WLP. As we have seen previously, we first have to be able to identify the different warps, in the same way SIMT does with threads. One way to obtain the warp identifier is to compute it at runtime. Indeed, we know that warps are formed by 32 threads in current architectures [NVIDIA2011a]. Thus, knowing the running kernel configuration thanks to CUDA defined data-structures, the warp identifier can be determined using simple operations only, similarly to what [Hong et al., 2011] have done. The definition of a `warpIdx` variable containing the warp's identifier can be written as in Listing B.1:

```
1 const unsigned int warpIdx = (
2   threadIdx.x + blockDim.x * (
3       threadIdx.y + blockDim.y * (
4               threadIdx.z + blockDim.z * (
5                       blockIdx.x + gridDim.x * blockIdx.y
6   ) ) ) ) / warpSize;
```

Listing B.1: Const-definition of warpIdx

Conceptually, this definition is ideal because `warpIdx` is declared as a 'constant variable', and the warp identifier does not change during a kernel execution. This formula fits with the CUDA way to number threads, which first considers threads' x indices, then y and finally z, within a block. The same organization is applied to blocks numbering [Kirk and Hwu, 2010]. Please note that the `warpSize` variable is provided

by CUDA. This makes our implementation lasting since warp sizes may evolve in future CUDA architectures.

Although this method introduces superfluous computations to figure out the kernel's configuration, we find it easier to understand for developers. Another way to compute the warp's identifier would have been to write CUDA PTX (Parallel Thread Execution) assembly[NVIDIA, 2011]. The latter is the Instruction Set Architecture (ISA) currently used by CUDA-enabled GPUs. CUDA enables developers to insert inlined PTX assembly into CUDA high-level code, as explained in [NVIDIA2011b]. However, this method is far less readable than ours, and would not be more efficient since we only compute `warpIdx` once: at initialization.

This warp identifier will serve as a base in WLP. When classical CUDA parallelism makes a heavy use of the runtime-computed global thread identifier, WLP proposes `warpIdx` as an equivalent.

Now that we are able to figure out threads' parent warp, let us restrain the execution of the kernel to a warp scope. Given that we need to determine whether or not the current thread is the first within its belonging warp, we will be faced to problems similar to those encountered when trying to determine the warp identifier. In fact, a straightforward solution relying on our knowledge of the architecture quickly appears. It consists in determining the global thread identifier within the block to ensure it is a multiple of the current warp size. Once again, the kernel configuration is issued by CUDA intrinsic data structures, but we still need a reliable way to get the warp size to take into account any potential evolution. Luckily, we can figure out this size at runtime thanks to the aforementioned `warpSize` variable. Consequently, here is how we begin a warp-scope kernel in WLP:

```
1 if ( ( threadIdx.x + blockDim.x *
2     ( threadIdx.y + blockDim.y * threadIdx.z ) )
```

Listing B.2: Directive enabling warp-scope execution

We now own the bricks to perform WLP, but still lack a user-friendly API. Indeed, it would not be adapted to ask our users to directly use complex formulas without having wrapped them up before in higher-level calls. A first attempt to do so is implemented though macros. As compile-time mechanisms, macros do not cause any runtime overhead. They are also perfectly handled by *nvcc*, the CUDA compiler. Our previous investigations result in two distinct macros: `WARP_BEGIN` and `WARP_INIT`, which respectively mark the beginning of the warp-scope code portion, and correctly fill the warp identifier variable. When `WARP_INIT` presents no particularities, except the

requirement to be called before any operations bringing into play `warpIdx`, `WARP_BEGIN` voluntary forgets the block-starting brace following the *if* statement. By doing so, we expect users to place both opening and closing braces of their WLP code if needed, just as they would do with any other block-initiating keyword.

To sum up, please note once again that this implementation mainly targets to validate our approach. Still, it lays the foundation of a more complete API dedicated to WLP. Unfortunately, macros are not convenient to use. They do not provide control check until compilation. Macros are also quite hard to debug, because they are inlined in the code. Thus, they are not suited to write production codes. Macros were useful in our case to validate the concept though.

B.4 AOP Declination of WLP

In this section, we study the possibility to take advantage of the inputs brought by Aspect Oriented Programming [Kiczales et al., 1997] in GPU programming, and especially when using CUDA.

AOP consists in defining entry points where code is inserted in order to modify the program's behaviour. These entry points are called pointcuts in AOP. A traditional example of aspects usage is the insertion of a pointcut to wrap a function call. For instance, the function call can be wrapped around the verification of a pointer, or an exception catching block. The point is this change has a small impact on the code thanks to the way aspects are handled. Indeed, the specific operations are externalized, and usually inserted at compile time. This way, operations can be done without being inserted directly in the code.

AOP generally involves a third-party software that will preprocess the source code in order to actually add the equivalent parts matching the aspect directives. For C++, a preprocessor was released in 2002 and is called AspectC++ [Spinczyk et al., 2002]. AspectC++ comes with lots of features but is still experimental, and thus does not fully support the C++ syntax and standard constructs. The aspect has to be defined in an "aspect header" (*.ah* extension) that will be used by the AspectC++ preprocessor (*ac++*) to weave the C++ code matching the pointcuts.

Our first attempt has been to try to implement WLP through pointcut matching any function called `wlp_*()`. In our case, we expected to obtain a similar behaviour to what plain macros delivered. The conditional statement, previously achieved by the

WARP_BEGIN macro, would have been added with the aspect specific code. The *warpIdx* variable, previously declared and initialized by *WARP_INIT*, would have been defined through an inlined function. Unfortunately, such an approach could not be completed for several reasons. First WLP would have been available at a kernel scope only, whereas the original implementation allows to turn each programming block into a WLP-run operation. Second, AspectC++ displays troubles parsing CUDA code and the Thrust library, making it unavailable within a CUDA project. This led us to abandon AspectC++ and to head towards another aspect implementation.

A classic way to implement handcrafted aspects in C++ is by using templates. This design is close to policy classes [Alexandrescu, 2001] where each new behaviour that is intended to be inserted is implemented at the heart of a new template class. This class is seen as a component, which can be plugged into the base class or into other components. In order to support this feature, each involved class must accept a template parameter. Moreover, when using C++03, the template function's prototype has to match the wrapped function's to enhance it. This forces the user to write its own aspect any time he wants to use WLP. This is why we abandoned this method. A new feature from C++11 called variadic templates would enable us to provide aspect facilities through templates without the need for the user to write its own template aspect class. However, *nvcc*, the CUDA compiler, does not support C++11 at the time of writing. In the same way, a third approach could have been considered by harnessing variadic macros. Yet, the latter are not standard in C++03 either.

Other languages, such as Java for instance, implement aspects through annotations inserted in the code. The latter are understood by a preprocessor that makes them part of the language keywords. As long as the C-style fashion to deal with aspects has shown unavailable or awkward in a CUDA-enabled project, we turned to annotations. This implied to write our own preprocessor, which is designed to adopt the same behaviour as Java annotations. The preprocessor itself is a simple Perl script that will rapidly parse the code to find the annotations. Its behaviour is simple and focuses on detecting the annotations in the original code, and neither evaluates the whole code, nor builds a full syntax tree. This which prevents issues with unsupported C++/CUDA features that are encountered by more complete pieces of software like AspectC++.

Macros make the code harder to debug because they are inlined. An aspect preprocessor can workaround this issue by taking advantage of the `#line` pragma. This pragma can be used to point to the compiler what will be the next line number to consider, no matter the previous line number. By doing so, when our preprocessor weaves the code, it also inserts the said pragma. The aspect is then totally transparent

for the user in case of a build error or when debugging. He will be warned of problems in his code at the exact line they were before any aspect code was inserted.

Concretely, in order to define a WLP kernel, the "// @warp" annotation just needs to be added before the kernel implementation. Two others annotations have been defined to enable WLP on part of the code only. The code parts that need WLP are delimited by "// @warp: begin" and "// @warp: end" at, respectively, the beginning and end of the block. Equivalent keywords from the macro WLP implementation and the AOP version are summed up in Table B.1. Obviously, this preprocessing phase occurs prior to the classical building stages from CUDA.

Aspect code	**Macro equivalence**

// @warp	WARP_BEGIN {
	WARP_INIT
	// actual code
	}

// @warp: begin	WARP_BEGIN { WARP_INIT
// @warp: end	}

Table B.1: Equivalence between original WLP through macros and aspect implementation

In a more concrete way, Listings B.3 and B.4 expose dummy code snippets using aspects to enable WLP. Listing B.3 shows an example of code for a whole WLP kernel, while Listing B.4 presents a code where WLP is bounded to part of the code only.

```
1 // @warp
2 __global__ void TestKernel(float * deviceArray) {
3   deviceArray[0] = 0.;
4   deviceArray[warpIdx] = 1.;
5 }
```

Listing B.3: A whole WLP kernel

```
1 __global__ void TestKernel2(float * deviceArray) {
2   deviceArray[0] = 0.;
3   // @warp: begin
4   deviceArray[warpIdx] = 1.;
5   // @warp: end
6 }
```

Listing B.4: WLP on part of the code only

B.5 Results

In this part, we introduce three well-known stochastic simulation models in order to benchmark our solution. We have compared WLP's performances on a Tesla C2050 board to those of a state-of-the-art scalar CPU: an Intel Westmere running at 2.527 GHz. For all of the three following models, each replication runs in a different warp when considering the GPU, whereas the CPU runs the replications sequentially. The following implementations use L'Ecuyer's Tausworthe three-component PRNG, which is available on both CPU and GPU respectively through Boost.Random and Thrust.Random [Hoberock and Bell, 2010] libraries. Random streams issued from this PRNG are then split into several sub-sequences according to the Random Spacing distribution technique [Hill, 2010].

B.5.1 Description of the models

First, we have a classical Monte Carlo simulation used to approximate the value of Pi. The application draws a succession of random points' coordinates. The number of random points present in the quarter of a unit circle are counted and stored. At the end of the simulation, the Pi approximation corresponds to the ratio of points in a quarter of the unit circle to the total number of drawn points. The output of the simulation is therefore an approximate Pi value. This model takes two input parameters: the number of random points to draw and the number of replications to compute.

The second simulation is a M/M/1 queue. For each client, the time duration before its arrival and the service time is randomly drawn. All other statistics are computed from these values. The program outputs are the average idle time, the average time in queue of the clients and the average time spent by the clients in the system. Because it

did not impact the performances, the parameters of the random distribution are static in our implementation. Only the number of clients in the system and the number of replications, which modify the execution time, can be specified when running the application.

The last simulation is an adaptation of the random walk tests for PNRGs exposed in [Vattulainen and Ala-Nissila, 1995]. The idea is to simulate a walker moving randomly on a chessboard-like map. The original application tests the independence of multiple flows of the same PRNG. To achieve this, multiple random walkers are run with different initializations of a generator on identically configured maps. Basically, each walker computes a replication. In the end, we count the number of walkers in every area of the map. Depending on the PRNG quality, we should find an equivalent number of walkers in each area. When the original version splits the map in four quarters, our implementation uses 30 chunks to put the light on the opportunity of our approach when there are many divergent branches in an application.

B.5.2 Comparison CPU versus GPU warp

As we can see in Figure B.2, the CPU computation time of the Monte Carlo application approximating the value of Pi grows linearly with the number of replications. The GPU computation time increases only by steps. This behaviour is due to the huge parallel capability of the device. Until the GPU is fully loaded, adding another replication does not impact the computation time, because they are all done in parallel. So, when the device is full, any new iteration will increase the computation time. This only happens on the 65^{th} replication because the GPU saved some resources in case a new kernel would have to be computed simultaneously. The same mechanism explains that after this first overhead, a new threshold appears and so on.

Due to this behaviour, GPUs are less efficient than CPUs when the board is nearly empty. When less than 30 replications are used, more than two-thirds of the board computational power is idle. Because sequential computation on CPU is widely faster than sequential computation on GPU, if only a little of the parallel capability of the device is used, the GPU runs slower. But when the application uses more of the device parallel computation power, the GPU becomes more efficient than the CPU.

The pattern is very similar for the second model: the M/M/1 queue (see Figure B.3). When the board does not run enough warps in parallel, the CPU computation is faster than the GPU one. But with this model, the number of replications needed for the GPU approach to outperform the CPU is smaller than what we obtained with the

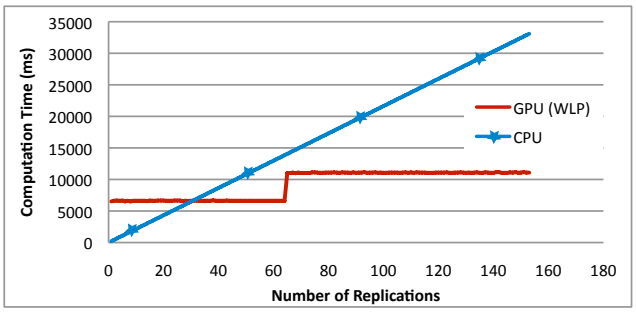


Figure B.2: Computation time versus number of replications for the Monte Carlo Pi approximation with 10,000,000 draws

previous simple model. The GPU computation is here faster as soon as 20 replications are performed, when it required 30 replications to show its efficiency with the first model. This can be explained by GPUs' architecture, where memory accesses are far more costly than floating point operations in terms of processing time. If the application has a better computational operations per memory accesses ratio, it will run more efficiently on GPU. Thus, the GPU approach will catch with the CPU one faster.

This point is very important because it means that depending on the application characteristics, it can be adequate to use this approach from a certain number of replications, or not. A solution is to consider the warp approach only when the number of replications is big enough to guaranty that most of the applications will run faster.

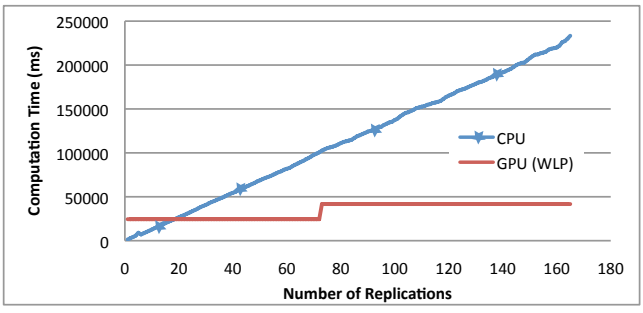


Figure B.3: Computation time versus number of replications for a M/M/1 queue model with 10,000 clients

B.5.3 Comparison GPU warp versus GPU thread

If the advantages of WLP-enabled replications compared to CPU ones in terms of computation time have been demonstrated with the previous examples, it is necessary to determine if WLP outperforms the classic TLP.

This case study has been achieved using the last model introduced: our adaptation of the random walk. Figure B.4 shows the computation time noticed for each approach: CPU, GPU with WLP and GPU with TLP (named *thread* in the caption). Obviously, CPU and WLP results confirm the previous pattern: the CPU computation time increases linearly when the WLP one increases by steps. TLP follows logically the same evolution shape as WLP. Although it is impossible to see it here because the number of replications is too small, it also evolves step by step, similarly to the warp approach. WLP consumes a whole warp for each replication. In the same time, TLP activates 32 threads per warp. Thus, the latter's steps will be 32 times as long as WLP's. Having said that, we easily conclude that the first step in TLP will occur after the 2048^{th} replication.

As we can see in Figure B.4, the computation time needed by the thread approach is significantly more important than the computation time of the warp approach (about 6 times bigger for the first 64 replications). But WLP catches up with TLP when the number of replications increases. When more than 700 replications are performed, the benefit of using the warp approach is greatly reduced. The best use of the warp approach for this model is obtained when running between 20 and 700 replications. Please note that this perfectly matches our replications amount requirement. It even allows the user to run another set of replications according to an experimental plan, or to run another set of replications with a different high quality PRNG. The latter practice is a good way to ensure that the input pseudo-random streams do not bias the results.

These results are backed up by the output of the NVIDIA Compute profiler for CUDA applications. The latter tool allows developers to visualize many data about their applications. In our case, we have studied the ratio between the time spent accessing global memory versus computing data. Such figures are displayed in Figure B.5 for both TLP and WLP versions of the random walk simulation. Our approach obviously outperforms TLP, given that the ratio of overall Global Memory access time versus computation time is about 2.5 times as big for TLP.

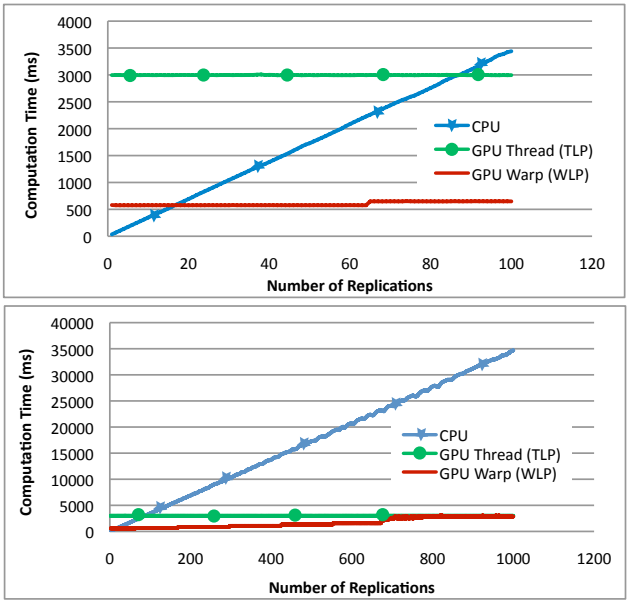


Figure B.4: Computation time versus number of replications for a random walk model with 1,000 steps *(above: 100 replications, below: 1,000 replications)*

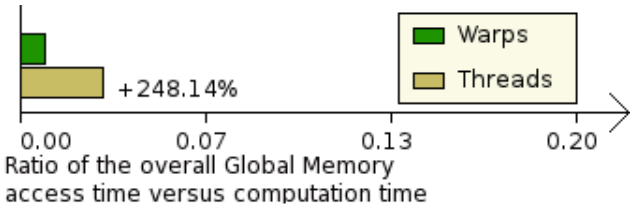


Figure B.5: Comparison of TLP and WLP ratio of the overall Global Memory access time versus computation time

To explain this ratio, let us recall that computation time was lower for WLP. Since the same algorithm is computed by the two different approaches, we should have noticed the same amount of Global Memory accesses in the two cases. In the same way, the profiler indicates significant differences between Global Memory reads and writes for TLP and WLP. These figures are summed up in Table B.2:

	TLP	***WLP***
Reads	225	18
Writes	302	104

Table B.2: Number of read and write accesses to Global Memory for TLP and WLP versions of the Random Walk

B.6 Conclusion

This work has shown that using GPUs to compute MRIP was both possible and relevant. Having depicted nowadays GPUs' architecture, we have detailed how warp scheduling was achieved on such devices, and especially how we could take advantage of this feature to process codes with a high rate of branch divergent parts. Our approach, WLP (Warp-Level Parallelism), can also help users to easily distribute their DOE experimental plans with replications on GPU.

WLP has been implemented thanks to simple arithmetic operations. Consequently, WLP displays a minimalist impact on the overall runtime performance. In order to validate the approach, the internal mechanisms enabling WLP have first been wrapped in macros. Then, for the sake of user-friendliness, another high level version has been proposed following an aspect-oriented approach. Aspects are implemented through annotations, preprocessed by a Perl script to generate the corresponding WLP blocks. At the time of writing, our version is functional and allows users to create blocks of code that will be executed independently on the GPU. Each warp will run an independent replication of the same simulation, determined by the warp identifier figured out at runtime. By doing so, we prevent performances to drop as they would do in an SIMT environment confronted to branch-divergent execution paths. WLP also tackles the GPU underutilization problem by artificially increasing the occupancy.

To demonstrate our approach performances, we have compared the execution times of a sequence of independent replications for three different stochastic simulations. Results show that WLP is at least twice as fast as a thread running on a cutting-edge CPU when asked to compute a reasonable amount of replications, that is to say more than 30 replications. This will always be the case when a stochastic simulation is studied with a design of experiments, where for each combination of deterministic factors, at least 30 replications shall be run, according to the previously mentioned Central Limit Theorem. WLP also overcomes the traditional CUDA SIMT performances by up to 6 to compute the same set of replications. Here, SIMT suffers of an underutilized GPU, whereas WLP takes advantage of the fast scheduling of warps.

Insofar performances of WLP increase with the Fermi architecture compared to Tesla, but it is difficult to forecast the same behaviour on new architectures without adapting the approach. The lack of information regarding the hardware and especially the schedulers ruling threads execution on a CUDA GPU forces us to perform new experiments and possibly adapt WLP.

Still the aspect-oriented declination of WLP opens new perspective in terms of software engineering for GPU devices. We have shown that AOP could be harnessed for such devices, provided an handcrafted preprocessor is available. Future releases of the CUDA toolkit should even withdraw this constraint by allowing established solutions such as AspectC++ and C++11 to fully match CUDA's specificities. Aspect could then deliver their full potential, and in our case, allow us to implement an OpenCL declination of WLP without changing the way it is used in client source code.

This thesis was written while listening to the following tunes...