Université de Montréal

Logiciel de génération de nombres aléatoires dans OpenCL

par
Nabil KEMERCHOU

Département d'informatique et de recherche opérationnelle
Faculté des arts et des sciences

Mémoire présenté à la Faculté des études supérieures
en vue de l'obtention du grade de Maître ès sciences (M.Sc.)
en informatique

Août, 2015

RÉSUMÉ

clRNG et *clProbdist* sont deux interfaces de programmation (APIs) que nous avons développées pour la génération de nombres aléatoires uniformes et non uniformes sur des dispositifs de calculs parallèles en utilisant l'environnement OpenCL. La première interface permet de créer au niveau d'un ordinateur central (*hôte*) des objets de type *stream* considérés comme des générateurs virtuels parallèles qui peuvent être utilisés aussi bien sur l'*hôte* que sur les dispositifs parallèles (unités de traitement graphique, CPU multi-noyaux, etc.) pour la génération de séquences de nombres aléatoires. La seconde interface permet aussi de générer au niveau de ces unités des variables aléatoires selon différentes lois de probabilité continues et discrètes.

Dans ce mémoire, nous allons rappeler des notions de base sur les générateurs de nombres aléatoires, décrire les systèmes hétérogènes ainsi que les techniques de génération parallèle de nombres aléatoires. Nous présenterons aussi les différents modèles composant l'architecture de l'environnement OpenCL et détaillerons les structures des APIs développées. Nous distinguons pour clRNG les fonctions qui permettent la création des streams, les fonctions qui génèrent les variables aléatoires uniformes ainsi que celles qui manipulent les états des streams. clProbDist contient les fonctions de génération de variables aléatoires non uniformes selon la technique d'inversion ainsi que les fonctions qui permettent de retourner différentes statistiques des lois de distribution implémentées. Nous évaluerons ces interfaces de programmation avec deux simulations qui implémentent un exemple simplifié d'un modèle d'inventaire et un exemple d'une option financière. Enfin, nous fournirons les résultats d'expérimentation sur les performances des générateurs implémentés.

Mots clés: RNG, parallélisme, OpenCL, GPU.

ABSTRACT

clRNG and *clProbdist* are two application programming interfaces (APIs) that we have developed respectively for the generation of uniform and non-uniform random numbers on parallel computing devices in the OpenCL environment. The first interface is used to create at a central computer level (*host*) objects of type *stream* considered as parallel virtual generators that can be used both on the host and on parallel devices (graphics processing units, multi-core CPU, etc.) for generating sequences of random numbers. The second interface can be used also on the host or devices to generate random variables according to different continuous and discrete probability distributions.

In this thesis, we will recall the basic concepts of random numbers generators, describe the heterogeneous systems and the generation techniques of parallel random number, then present the different models composing the OpenCL environment. We will detail the structures of the developed APIs, distinguish in clRNG the functions that allow creating streams from the functions that generate uniform random variables and the functions that manipulate the states of the streams. We will describe also clProbDist that allow the generation of non-uniform random variables based on the inversion technique as well as returning different statistical values related to the distributions implemented. We will evaluate these APIs with two simulations, the first one implements a simplified example of inventory model and the second one estimate the value of an Asian call option. Finally, we will provide results of experimentations on the performance of the implemented generators.

Keywords : RNG, Parallel Computing, OpenCL, GPU.

TABLE DES MATIÈRES

LISTE DES TABLEAUX

LISTE DES FIGURES

LISTE DES CODES SOURCE

LISTE DES ALGORITHMES

LISTE DES SIGLES

ALU	Arithmetic Logic Unit
API	Application programming Interface
CDF	Cumulative Distribution Function
CPU	Central Processing Unit
CRN	Commun Random Number
FPGA	field-programmable gate arrays
GPU	Graphical Processing Unit
GPGPU	General Purpose Graphical Processing Unit
GMCH	Graphics and Memory Controller Hub
ICH	I/O Controller Hub
IRN	Independent Random Number
JIT	Just In Time
LFSR	Linear Feedback Shift Register
MPPA	Massively parallel processor array
PC	Personnel Computer
PDF	Probability Density Function
RNG	Random Number Generator
SIMD	Single instruction multiple data
SSJ	Stochastic Simulation in Java
V.A.	Variable(s) Aléatoire(s)
WI	Work item

À mes parents, mon épouse et mon fils.

REMERCIEMENTS

J'aimerais remercier tous les enseignants dont j'ai eu l'honneur d'être l'étudiant, particulièrement le Professeur Pierre L'Ecuyer pour m'avoir beaucoup appris durant ses cours et m'avoir dirigé et soutenu tout au long de ce travail.

Je remercie aussi le Conseil de Recherche en Sciences Naturelles et en Génie du Canada (CRSNG) pour avoir soutenu financièrement ce projet.

Je tiens à souligner le support précieux apporté par David Munger dans ce projet au niveau de la conception et des tests ; merci pour cette collaboration efficace.

Mes remerciements vont finalement à mes parents pour avoir fait de moi ce que je suis, à toute ma famille pour l'encouragement et le soutien tant appréciés.

CHAPITRE 1

INTRODUCTION

Les générateurs de nombres pseudo-aléatoires (PRNG) sont au cœur de toute simulation stochastique. Nous utilisons généralement des algorithmes déterministes pour produire des séquences périodiques de nombres réels qui se comportent comme s'ils étaient générés à partir d'une vraie source de nombres aléatoires indépendants et uniformément distribués sur l'intervalle (0,1) (ou i.i.d $\mathscr{U}(0,1)$). Ces nombres peuvent ensuite être transformés pour générer des variables aléatoires selon d'autres lois de probabilité.

Depuis quelques années, nous observons que la fréquence d'horloge des processeurs ne change plus aussi vite que durant les décennies précédentes. Cela est attribué notamment au phénomène de dissipation thermique [35]. Les fabricants ont recours à l'augmentation du nombre de microprocesseurs intégrés sur une même puce pour fournir une architecture multi-noyaux. Cela a un impact important sur la façon de développer les logiciels qui doivent être écrits en *multithreads* pour tirer avantage de la puissance de calcul et du parallélisme offerts.

Dans ce mémoire, nous nous intéressons à la conception et l'implémentation d'interfaces de programmations (APIs) pour des générateurs de nombres aléatoires dans un contexte parallèle [23]. Les dispositifs de calcul tels que les unités de traitement graphique (GPU) ou les unités de traitement accéléré (APU) offrent la possibilité d'exécuter un programme de façon parallèle sur des centaines, voire des milliers d'*éléments de traitement* (PE) qui peuvent être considérés comme des unités arithmétiques et logiques (ALU) ayant chacune une mémoire privée. Les APIs à concevoir utiliseront ce modèle d'exécution pour fournir un gain considérable en performance par rapport à une exécution en série sur un CPU.

La première interface *clRNG* permettra de créer au niveau d'un processeur central

appelé *hôte* des objets de type stream qui seront utilisés pour la génération de nombres aléatoires indépendants. Ces streams doivent être copiés sur les dispositifs de calcul (*device*) pour qu'ils puissent être exploités par des processus parallèles appelés *work items*. Les streams peuvent être subdivisés en plusieurs sous-séquences (sous-streams) de tailles égales, chacun pouvant être considéré comme un RNG virtuel. La manipulation de ces séquences comme le saut en avant vers d'autres sous-séquences ou le retour vers l'état initial peut se faire au niveau de l'hôte comme au niveau du dispositif. La deuxième interface *clProbdist* permettra la génération de nombres aléatoires non uniformes avec le même modèle d'exécution parallèle.

Plusieurs générateurs seront implémentés au niveau des deux APIs. La première interface sera évaluée à travers la simulation Monte-Carlo d'un exemple simplifié du modèle d'inventaire. Pour la deuxième interface, nous utiliserons la simulation d'une option financière asiatique pour illustrer l'estimation d'une espérance mathématique (la valeur équitable de l'option, dans ce cas-ci) par Monte Carlo sur des processeurs parallèles. Cela permettra aussi de montrer comment utiliser les APIs de façon efficace.

OpenCL est un environnement de développement qui définit, en plus d'un langage de programmation, un cadre d'exécution parallèle permettant de faire abstraction des dissimilarités entre les plates-formes hétérogènes. Cela permet ainsi d'assurer que le code des RNG devienne portable sur différent dispositifs parallèles.

Dans ce qui suit, nous allons rappeler des notions de base sur les générateurs de nombres aléatoires uniformes et introduire la norme OpenCL. Le plan du mémoire et la contribution apportée seront décrits par la suite.

1.1 Les générateurs pseudo-aléatoires uniformes

Un PRNG correspond à un algorithme déterministe qui a pour but de produire une séquence de nombres qui semble la plus aléatoire possible. Le générateur est défini [20]

formellement comme une structure $(S, \mathscr{U}, \mu, f, g)$ où :

— S : un ensemble fini d'états

— $\mathscr{U}$: un ensemble fini de symboles de sortie

— μ : une loi de probabilité sur l'espace d'états

— $f : S \to S$: fonction de transition déterministe d'un état à l'autre

— $g : S \to \mathscr{U}$: une fonction de sortie qui fait correspondre à chaque état une valeur de sortie dans l'intervalle continu (0,1).

Initialement, le générateur est à l'état s_0 (appelé *germe*). L'application répétée de la fonction f sur l'état initial génère une séquence d'états $s_0, s_1, s_2, s_3..$ qui seront transformés par la fonction g en des sorties pseudo-aléatoires correspondantes $x_0, x_1, x_2, x_3...$ telles que :

$$s_n = f(s_{n-1}) \qquad \forall n \geq 1$$
$$x_n = g(s_n)$$

Puisque S est un ensemble fini et f est une fonction déterministe, alors la séquence de sortie va obligatoirement se répéter. Nous appelons la période ρ du générateur le plus court *cycle de la séquence*, qui est défini comme suit :

$$\rho = \min\{\nu > 0 : s_{n+\nu} = s_n \quad \text{pour un} \quad n \geq 0\}$$

avec $1 \leq \rho \leq |S|$, il se peut que l'égalité ne soit vraie que pour n à partir d'un certain $n_0 > 0$. Dans le cas où l'état du générateur est représenté sur b bits de mémoire, alors on aura [20] une borne supérieure $\rho \leq 2^b$. En général, la période peut dépendre de l'état initial s_0. Pour illustrer le concept, prenons l'exemple d'un générateur simple appelé *générateur à congruence linéaire* (LCG) :

$$x_n = f(x_{n-1}) = ax_{n-1} \mod m$$
$$u_n = x_n/m$$

où mod est l'opérateur modulo, m est un entier positif et $0 < a < m$. La suite des états est obtenue en appliquant la fonction f successivement et à chaque fois en divisant par m pour obtenir une valeur de sortie qui est strictement entre 0 et 1 à condition qu'on n'ait jamais $x_n = 0$. Puisque l'état 0 est absorbant (on ne quitte jamais cet état une fois atteint) il faut enlever cet état de l'ensemble des états possibles, soit $S = \{1\ ,\ \ldots\ ,\ (m-1)\}$. Cela nous donne les états de sortie suivantes $U = \{1/m\ ,\ \ldots\ ,\ (m-1)/m\}$. Ainsi la période de ce générateur est $p \leq m$. La période est égale à $m-1$ si et seulement si m est un nombre premier et a est un élément primitif modulo m [12, 13].

1.2 La norme OpenCL

OpenCL est une norme qui définit un langage de programmation *OpenCL C* (basé sur C99) qui permet d'écrire du code portable, que l'on peut faire exécuter sur différents types de matériel. Ce code contient des noyaux (kernels) dont plusieurs copies peuvent s'exécuter simultanément sur plusieurs *unités de traitement* (PE) qui composent le dispositif de calcul parallèle (CPU multicoeurs, GPU, APU, etc). Cela définit aussi une interface de programmation (API) qui fournit un ensemble de commandes permettant par exemple le lancement des kernels. La synchronisation entre les work items ou le transfert d'espaces mémoires (buffers) sur le dispositif se fait à partir de l'ordinateur *hôte* faisant office de chef d'orchestre.

Chaque PE possède une unité ALU et une mémoire privée avec accès rapide et peut aussi communiquer avec d'autres PE via une mémoire partagée ou globale, mais de façon plus lente. Les work items (processus parallèles) sont organisés en blocs appelés *work groups* qui s'exécutent en mode SIMD (single instruction multiple data). Ils exécutent

la même instruction mais diffèrent par les données que chacun est en train de traiter. Le standard OpenCL est maintenu par le consortium Khronos.

1.3 Plan du mémoire

Le chapitre suivant présente plus en détail les générateurs de nombres aléatoires avec leurs différentes catégories et leurs critères de qualité ainsi que les différentes techniques de génération de nombres aléatoires non uniformes, la méthode de simulation Monte-Carlo et la technique des nombres aléatoires communs.

Le chapitre 3 contient une description des systèmes hétérogènes parallèles. Nous listerons les principales contraintes liées au développement dans ce contexte. Ensuite nous présenterons des techniques de génération parallèle de nombres aléatoires avec les critères de qualité spécifiques à ce contexte.

Le chapitre 4 détaille l'architecture de la norme OpenCL avec le modèle de la plateforme et ses différentes composantes, le modèle d'exécution décrivant comment les work items sont organisés ainsi que les étapes d'exécution d'un kernel. Nous présenterons aussi le modèle hiérarchique de la mémoire. Le chapitre se termine par une liste des bonnes pratiques de programmation permettant d'optimiser l'utilisation des ressources dans le contexte d'OpenCL.

Le chapitre 5 définit les générateurs implémentés dans clRNG et décrit en détail les fonctionnalités de cette API en distinguant les fonctions qui permettent la création des streams, les fonctions qui génèrent les nombres aléatoires uniformes et les fonctions qui manipulent l'état courant des séquences. clProbDist sera présenté par la suite. Nous définirons les lois de probabilité implémentées et nous détaillerons les structures de données et les fonctions développées.

Le chapitre 6 détaille les simulations utilisées pour illustrer et expérimenter les deux

APIs. Nous implémenterons un exemple simplifié d'un modèle d'inventaire ainsi qu'une option financière afin d'évaluer différentes fonctionnalités de clRNG et clProbdDist. Plusieurs configurations de streams et de sous-streams seront expérimentées sur différents dispositifs. Nous donnerons des statistiques et des indicateurs de performance.

Enfin, les principales conclusions seront présentées dans le dernier chapitre avec des idées d'extensions pour des travaux futurs.

1.4 Contribution

Les APIs que nous avons implémentées exploitent les algorithmes disponibles dans la librairie Java SSJ qui a été construite par une équipe de simulation de l'Université de Montréal. Ce travail a l'originalité d'avoir réadapté ces algorithmes pour créer une nouvelle conception avec des nouvelles structures de données et des nouvelles fonctions pour les RNG sur les plates-formes parallèles. La contribution principale de ce travail consiste en la conception et l'implémentation de deux interfaces de programmation pour la génération de nombres aléatoires sur des dispositifs de calcul parallèles avec des streams et sous-streams en OpenCL. Pour notre part, nous avons eu la chance de participer à la conception de ces interfaces avec Pierre L'Ecuyer et son assistant David Munger [22]. Nous avons aussi programmé et testé tous les générateurs des APIs et implémenté les exemples de simulation que nous avons expérimentés.

Nous avons implémenté quatre générateurs uniformes et évalué leurs performances sur différents dispositifs ; nous avons étendu le générateur à base de compteur Philox [9] avec les fonctionnalités de streams, sous-streams et sauts en avant, et nous avons implémenté un nouveau générateur *cLFSRMRG* (combinant les sorties du générateur LFSR113 avec MRG31K3p), ce dernier passe toutes les batteries de tests du module *TestU01*. Nous avons aussi implémenté cinq générateurs de nombres aléatoires non uniformes suivant des lois de probabilités continues et discrètes. Beaucoup de contraintes d'implémentation ont été résolues de façon élégante comme celle du transfert des struc-

tures de données de la distribution de Poisson de l'ordinateur hôte vers le dispositif de calcul parallèle (tel un GPU, par exemple). Ces structures sont des tableaux précalculés permettant d'améliorer la performance de calcul. Cependant, leurs tailles changent suivant les paramètres de la loi de probabilité (voir chapitre 5).

Nous avons développé des exemples concrets de simulations pour évaluer nos APIs. Tous nos logiciels et exemples sont disponibles aux utilisateurs au niveau de l'Université de Montréal [19] et au niveau de la compagnie AMD [30].

CHAPITRE 2

GÉNÉRATEURS DE NOMBRES ALÉATOIRES

Les nombres aléatoires peuvent être générés par des appareils qui exploitent des processus physiques qui produisent des signaux statistiquement aléatoires (comme les effets photoélectriques ou le bruit thermique généré par les semi-conducteurs) d'où nous pouvons extraire des bits qui pourront être utilisés dans la construction de nombres aléatoires. Cette façon a l'avantage de générer des valeurs reconnues comme étant des vrais nombres aléatoires. Cependant, le matériel nécessaire n'est pas facile à mettre en place et la séquence aléatoire produite ne peut pas être générée exactement pareille une deuxième fois [23]. Un autre type de générateur appelé *pseudo-aléatoire* (PRNG) est basé sur des algorithmes déterministes qui permettent de générer des séquences de nombres qui *semblent* être aléatoires et qui peuvent être totalement reproduites, ce qui les rend adaptées aux méthodes de simulation par ordinateur.

Dans ce qui suit, je vais définir les catégories des générateurs pseudo-aléatoires uniformes, leurs critères de qualité et les tests statistiques utilisés pour vérifier l'uniformité des nombres produits. Ensuite, je présenterai les générateurs de nombres aléatoires non uniformes avec quelques techniques permettant de transformer une variable aléatoire uniforme en une variable qui suit une loi de probabilité non uniforme. Enfin, la méthode de simulation Monte-Carlo sera définie avec la technique des nombres aléatoires communs.

2.1 Catégories des générateurs Uniformes

Les RNGs peuvent être classés selon la nature mathématique de leur fonction de transition qui est soit linéaire, non linéaire ou combinée. Nous distinguons aussi les générateurs à base de compteur ayant une fonction de transition qui fait simplement incrémenter un index mais contient au niveau de la fonction de sortie toute la complexité

pour générer les variables aléatoires.

2.1.1 Générateurs basés sur une récurrence linéaire

Les générateurs à récurrence multiple (MRG) :
Un générateur de type MRG (Multiple Recursive Generator) a la fonction de transition linéaire suivante [14] :

$$x_n = (a_1 x_{n-1} + a_2 x_{n-2} + ... + a_k x_{n-k}) \mod m$$
$$u_n = x_n / m$$

où : m et k sont des entiers positifs, les a_i et b appartiennent à $\mathbb{Z}_m = \{0, 1, ..., m-1\}$, l'état du générateur à l'étape n correspond au vecteur $s_n = (x_n, ..., x_{n-k+2}, x_{n-k+1})$ et la valeur de x_n est comprise entre 0 et $m-1$. La période maximale est $\rho = m^k - 1$ [28].

Les générateurs à rétroaction linéaire (LFSR) :
Contrairement au générateur MRG qui produit des nombres qui sont des multiples de $1/m$, le générateur LFSR se distingue par une discrétisation plus fine des nombres aléatoires. Il possède la fonction de transition et la fonction de sortie suivante :

$$x_n = (a_1 x_{n-1} + a_2 x_{n-2} + ... + a_k x_{n-k}) \mod 2$$
$$u_n = \sum_{j=1}^{L} x_{n\nu+j-1} 2^{-j}.$$

La valeur de L équivaut à la taille d'un mot machine, soit $L = 32$ ou $L = 64$ et ν est un entier positif correspondant à la taille du pas [28], ce qui fait que les valeurs générées sont des multiples de $1/2^L$. La fonction de transition est un cas particulier de celle décrite pour les MRG ; donc, nous avons la période maximale $\rho = 2^k - 1$, il faut choisir l'ordre de la récurrence assez grand pour obtenir une longue période. LFSR est un cas très particulier de la famille des générateurs linéaires sur $\mathbb{F}_2$, qui contient beaucoup d'autres générateurs populaires tels que Mersenne twister, WELL, etc.

2.1.2 Générateurs avec récurrence non-linéaire

La formule linéaire de la récurrence des RNG présentés dans la section précédente se traduit par une structure trop régulière des points construits dans l'hypercube a s dimensions à partir des s sorties consécutives, ce qui rend ces générateurs non adéquats pour l'utilisation dans certains domaines critiques comme la cryptographie [28]. Pour remédier à cela, de la non-linéarité est introduite dans les fonctions de transition et de sorties sous forme de formules quadratiques, cubiques ou en utilisant des composants inversés. Pour plus de détails, voir [13, 23] et les références qui s'y trouvent.

2.1.3 Générateurs combinés

Chacune des deux catégories présentées précédemment possède ses avantages et ses inconvénients. Pour les générateurs non linéaires, les points ont moins de structure et l'uniformité est beaucoup plus difficile à mesurer mathématiquement (on doit se limiter à des tests empiriques). Les générateurs linéaires quant à eux offrent une meilleure performance. D'autres générateurs sont apparus qui combinent les sorties de plusieurs générateurs de mêmes familles ou appartenant à des familles différentes, cela pour concilier la performance avec la qualité de la structure des points générés.

Plusieurs approches ont été proposées [14] pour combiner J générateurs de même famille. Par exemple, pour $j = 1, ..., J$, on note m_j, k_j, $x_{j,n}$, $u_{j,n}$ et ρ_j les valeurs de m, k, x_n, u_n et la période qui correspondent à la composante j, nous pouvons construire un RNG qui a une fonction de transition qui correspond à la combinaison linéaire des $x_{j,n}$ états de sorties produits à partir de J générateurs de type MRG ; le résultat est ramené entre 0 et $m_1 - 1$ pour obtenir les u_n. La période dans ce cas est égale au plus petit commun multiple des périodes de ses composants $\rho = PPCM(\rho_1, ..., \rho_j)$.

Nous trouvons dans [10] des exemples de générateurs ayant des composantes de familles différentes combinant un MRG ou un LFSR avec un autre générateur non linéaire

par l'addition des sorties $u_{1,n}$ et $u_{2,n}$ des deux RNGs comme suit $((u_{1,n}+u_{2,n}) \mod 1)$ ou par l'application du ou-exclusif bit à bit entre les sorties $(u_{1,n} \oplus u_{2,n})$. Cela permet d'améliorer la structure trop régulière des générateurs linéaires tout en développant des algorithmes plus performants que les générateurs non linéaires. Dans tous les cas, nous notons que nous pouvons construire un générateur combiné de grande période même si les J composantes ont relativement des petites périodes.

Exemple Mrg32k3a : Ce générateur est constitué de deux composants d'ordre 3 de type MRG. Son état contient deux vecteurs $s_{1,n} = (x_{1,n}, x_{1,n+1}, x_{1,n+2})$ et $s_{2,n} = (x_{2,n}, x_{2,n+1}, x_{2,n+2})$ qui suivent la récurrence linéaire suivante [25] :

$$x_{1,n} = (1403580\, x_{1,n-2} - 810728\, x_{1,n-3}) \mod m_1$$
$$x_{2,n} = (527612\, x_{2,n-1} - 1370589\, x_{2,n-3}) \mod m_2$$

avec : $m_1 = 2^{32} - 209 = 4294967087$ et $m_2 = 2^{32} - 22853 = 4294944443$.
La sortie u_n de ce générateur est définie par la formule suivante :

$$z_n = (x_{1,n} - x_{2,n}) \mod m_1$$
$$u_n = \begin{cases} z_n/4294967088 & \text{pour } z_n > 0 \\ m_1/4294967088 & \text{pour } z_n = 0. \end{cases}$$

La période est : $\rho = (m_1^3 - 1)(m_2^3 - 1)/2 \approx 2^{191}$.

2.1.4 Générateurs à base de compteur

Les générateurs présentés jusqu'ici ont été conçus pour générer des variables aléatoires en utilisant une fonction de sortie triviale et avec une fonction de transition qui contient toute la complexité. Les générateurs à base de compteur utilisent une approche inverse ; la fonction de transition se résume en l'incrémentation d'un compteur tandis

que la fonction de sortie utilise une fonction bijective complexe pour générer les variables i.i.d $\mathscr{U}(0,1)$.

Un RNG à base de compteur est défini formellement [9] par un espace d'état interne d'entiers $\mathscr{S}$, un espace de sortie $\mathscr{U}$, une espace de clés $\mathscr{K}$, une multiplicité de sortie entière J et respectivement par la fonction de transition et la fonction de sortie suivante :

$$f : \mathscr{S} \to \mathscr{S}$$
$$g : \mathscr{K} \times \mathbb{Z}_J \times \mathscr{S} \to \mathscr{U}.$$

Chaque état interne est constitué de p bits et la fonction de transition f est simplement un compteur de la forme $f(s) = (s+1) \mod 2^p$. La fonction de sortie g est l'essence du générateur et se constitue d'un sélecteur h_j composé (notons l'opérateur $\circ$) avec une fonction bijective complexe b_k ayant une clé k, ainsi pour $j = 1, ..., J$ nous aurons :

$$g_{k,j} = h_j \circ b_k.$$

Le rôle du sélecteur h_j est simplement de choisir le $j$ème bloc de r bits à partir des p bits (avec $r \leq \lfloor p/J \rfloor$). Par exemple, pour un générateur ayant un espace d'état avec $p = 128$ bits et $J = 4$ et pour une clé k donnée, la fonction bijective générera une valeur de 128 bits et le sélecteur h_j choisira comme valeur de sortie un bloc de 32 bits parmi les quatre blocs disponibles à chaque incrémentation. La fonction de transition g utilise 2^p états distincts et génère J sorties à chaque fois. Donc, la période de l'état du générateur est égale à $J2^p$.

Ce type de générateur a l'avantage de simplifier la création de streams et de sous-streams de nombres aléatoires, il suffit pour cela de diviser l'espace d'états du compteur en plusieurs segments et d'appliquer l'addition entre une valeur de pas et l'état courant pour effectuer le saut en avant. Les générateurs à base de compteur sont aussi bien adap-

tés au contexte parallèle vu que nous n'avons pas besoin de générer les états de façon successive pour produire les sorties aléatoires.

2.2 Segments et sous-segments

Partitionner un générateur - ayant une fonction de transition $f(s_i) = s_{i+1}$ et une période ρ - en segments (streams) et sous-segments (sous-streams) disjoints peut se faire en choisissant des entiers positifs v et w tels que $z = v + w$ puis en divisant la période en des séquences de longueur $Z = 2^z$ états. Chacune est subdivisée à son tour en $V = 2^v$ sous-segments de longueur $W = 2^w$. Si nous notons I_g l'état initial du segment g, alors nous aurons $I_1 = s_0$, $I_2 = f^z(s_0)$, ... $I_g = f^z(I_{g-1})$. Le premier sous-segment du segment g commence à l'état I_g, le second sous-segment commence à l'état $f^w(I_g)$ et le troisième à l'état $f^{2w}(I_g)$ et ainsi de suite.

À n'importe quel moment, l'état courant du segment g est noté C_g et le début du sous-segment qui contient cet état est noté B_g ; l'état initial du prochain sous-segment est noté N_g. La figure 2.1 montre que l'état courant du segment g est à la 4e valeur du troisième sous-segment [25].

L'organisation du générateur en plusieurs segments ainsi que la possibilité de pouvoir sauter de façon rapide en avant vers le prochain sous-segment nous permettent de considérer chaque séquence comme étant un RNG virtuel [12] .

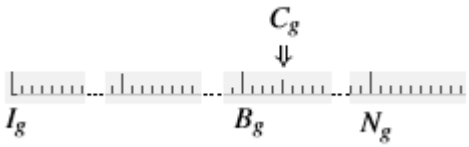

Figure 2.1 – Division de la période du générateur en segments et sous-segments

2.3 Critères de qualité des générateurs uniformes

Les générateurs de nombres aléatoires uniformes qui satisfont les caractéristiques suivantes peuvent être considérés comme étant de bonne qualité [13, 1] :

- Période : qui doit être assez longue pour que la séquence de nombres aléatoires ne puisse pas être complètement consommée par n'importe quel système.
- Reproductibilité : c'est un critère très important en simulation ; cela permet de reproduire toujours la même séquence de nombres aléatoires si nous répétons l'exécution du générateur à partir du même état initial et cela permet la vérification des résultats ou la résolution des erreurs des expérimentations.
- Efficacité : le générateur doit pouvoir fournir des millions de nombres aléatoires dans un temps qui n'impacte pas la performance du système qui l'utilise ; il doit aussi être implémenté de façon à exploiter la mémoire des dispositifs de calcul de façon efficace.
- Portabilité : l'implémentation des générateurs ne doit pas dépendre de l'architecture d'un type de machine donné.
- La possibilité de faire des sauts en avant vers les prochains segments et sous-segments sans avoir à générer tous les nombres intermédiaires.
- Bonne émulation du hasard : cela signifie que le générateur doit avoir des bonnes bases théoriques pour permettre la génération de nombres pseudo-aléatoires que l'on ne peut pas distinguer facilement d'une vraie source aléatoire.

En plus de ces critères qualitatifs, un bon générateur doit posséder d'autres caractéristiques quantitatives qui sont vérifiées par des tests théoriques et statistiques qui seront présentés dans la prochaine section.

Mesure d'uniformité et équidistribution :

Considérons l'ensemble Ψs contenant les vecteurs composés des s sorties successives $u_{n,s} = (u_n, .., u_{n+s-1})$ d'un générateur de vrais nombres aléatoires. Si ces vecteurs sont uniformément distribués dans l'hypercube unitaire de dimension s pour chaque entier positif s et $n \geq 0$ et pour tous les états initiaux s_0, alors nous dirons que les sorties

u_n sont indépendantes et identiquement distribuées $\mathscr{U}(0,1)$. Cela est une propriété qui ne s'applique qu'à des vraies variables aléatoires uniformes au sens mathématique. Pour les PRNGs, nous voulons approximer cette propriété [25]. Pour mesurer l'uniformité, nous utilisons - pour les générateurs de type MRGs - le *test spectral* en dimension s qui mesure l'uniformité de l'ensemble des s-tuples de valeurs successives produites par le générateur.

Une autre propriété importante est l'équidistribution qui implique que si l'on divise l'hypercube unitaire $[0,1)^s$ en 2^{sl} cases de même taille et si l'on place l'ensemble des points 2^k de Ψ_s dans ces cases-là, alors nous devrions trouver que chaque case a le même nombre de points qui est égale à 2^{k-sl}. Nous ne pouvons vérifier cette propriété en pratique que pour les générateurs $\mathbb{F}_2$-linéaires. Cela inclut les LFSR en particulier.

2.4 Tests statistiques

Les tests théoriques présentés dans la section précédente permettent d'analyser la structure des points produits par le générateur. Nous pouvons aussi tester statistiquement l'hypothèse H_0 selon laquelle la séquence des n sorties sont i.i.d $\mathscr{U}(0,1)$; il suffit de définir un test statistique Z qui a une distribution connue ou qui peut être approximée :

$$Z = \mathscr{S}(u_0, u_1, ..., u_n - 1).$$

Pour tester l'hypothèse, nous calculons la valeur de la statistique p appelée *valeur-p*, qui est la probabilité d'obtenir la même valeur du test (ou une valeur encore plus extrême) si H_0 était vraie. Nous dirons que le test a "échoué" si le résultat est trop petit (par exemple moins que 10^{-10}) ou trop grand (entre 1 - 10^{-10} et 1). Si la statistique mesurée n'est pas remarquable (par exemple 10^{-4} < valeur-p < (1 - 10^{-4})), alors le test est considéré comme "réussi". Les valeurs intermédiaires sont notées "douteuses" et il faut refaire le test avec d'autres échantillons u_n et d'autres tests statistiques plus puissants jusqu'à ce que le test réussisse ou échoue.

Un test qui a réussi exprime seulement le fait qu'il n'a pas été suffisant pour détecter une différence entre une vraie séquence de nombres aléatoires et la séquence en question. Cependant si l'on a beaucoup de tests qui réussissent alors cela permet d'être plus confiant dans l'affirmation que les u_n ne sont pas faciles à distinguer d'une vraie source de nombres aléatoires i.i.d $\mathscr{U}(0,1)$.

Plusieurs logiciels de tests statistiques ont été développés pour vérifier les séquences de sorties des RNG, notamment TestU01 [24] qui offre trois batteries de tests préconfigurés, à savoir "Small Crush"(avec 10 tests et 16 valeurs-p), "Crush" (96 tests et 187 valeurs-p) et "Big Crush" (106 tests et 254 valeurs-p) et ayant des temps d'exécution respectivement de quelques secondes à quelques minutes et jusqu'à plusieurs heures.

Test sériel :

Ce test n'est qu'une variante du *test d'ajustement de Pearson* (Goodness-of-fit) qui a en fait de nombreuses variantes (voir par exemple [26]). Le test sériel permet de vérifier si un ensemble d'observations suit une loi de distribution donnée [28]. Nous pouvons utiliser le Chi-deux comme statistique, mais nous pouvons aussi utiliser autre chose, comme le nombre de collisions par exemple. Il y a plusieurs variantes de ce test implantées dans TestU01. Dans le contexte des RNG, nous l'utilisons comme suit :

1. Nous générons un échantillon de r vecteurs composés de s points chacun à partir de $n = rs$ nombres aléatoires successifs, soit :
   $U_i = (u_i, u_{i+1}, ..., u_{i+s-1}) \quad i = 0..r-1$
2. Nous partitionnons les axes de l'hypercube unitaire $[0,1)^s$ en d sous-segments de longueur $1/d$ afin de former $k = d^s$ cellules.
3. Le test statistique Z est formé en comptant le nombre de points N_j qui tombent dans chaque cellule j pour $j = 1, .., k$, dans ces cas, les vecteurs $(N_1, .., N_k)$ suivent une distribution multinomiale avec des paramètres $(k, p_1, .., p_k)$, où $p_j = 1/k$
4. Sous l'hypothèse H_0 on sait que la quantité X^2 suit une distribution Chi-deux

avec $(k-1)$ degrés de liberté quand la valeur de r tend vers l'infini :

$$X^2 = \sum_{j=1}^{k} \frac{(N_j - r/k)^2}{r/k}.$$

5. Enfin, nous calculons la valeur x prise par X^2 et nous acceptons ou rejetons H_0 suivant la valeur de $p = P(X^2 < x \mid H_0)$.

2.5 Générateurs de nombres aléatoires non uniformes

Ces générateurs permettent de produire des séquences aléatoires qui suivent une loi de probabilité non uniforme (loi de Poisson, loi normale, etc.). Cela permet de développer des simulations qui se rapprochent le plus du comportement réel des modèles stochastiques analysés. Nous décrivons dans ce qui suit quelques méthodes pour générer ce type de variables.

2.5.1 Inversion

Chaque loi de probabilité est définie par une fonction de répartition (CDF) qui prend en entrée une valeur x et détermine la probabilité u de trouver une valeur plus petite. Nous pouvons simplement utiliser une variable uniforme $\mathscr{U}(0,1)$ comme la valeur d'entrée de la fonction de répartition inverse pour obtenir une variable non uniforme qui suit la loi distribution qui est définie par le CDF. À titre d'exemple, les formules suivantes montrent cette relation pour la distribution exponentielle :

$$F_{\exp}(x) = P(X \leq x) = 1 - e^{-x}$$
$$x = F_{\exp}^{-1}(x) = -\ln(1-u).$$

La technique d'inversion peut être utilisée seulement quand il existe une forme analytique (ou une bonne approximation) de l'inverse de la CDF.

2.5.2 Transformation

Cette méthode permet de générer deux variables aléatoires suivant la loi normale standard en faisant un changement de variables sur deux uniformes $\mathscr{U}(0,1)$ comme suit :

$$g_1 = \sqrt{-2\ln(1-u_2)}\sin(2\pi u_1), \qquad g_2 = \sqrt{-2\ln(1-u_2)}\cos(2\pi u_1).$$

Cette transformation de variables a l'avantage d'être simple et permet de générer plusieurs variables à la fois, mais elle est coûteuse en temps de calcul à cause des fonctions mathématiques utilisées.

2.5.3 Méthode de rejet

Cette méthode est appropriée dans le cas où on veut générer une variable aléatoire suivant une loi de probabilité ayant une fonction de densité $f(x)$ compliquée. Il suffit de trouver une distribution $g(x)$ à partir de laquelle nous pouvons facilement échantillonner (e.g. la distribution uniforme) ainsi qu'une constante c telle que $c \cdot g(x) \geq f(x)\ \forall x$. Nous pouvons accepter un échantillon X tiré à partir de $c \cdot g(x)$ comme étant tiré à partir de $f(x)$ selon la procédure suivante :

1. Soit $X \sim g(x)$
2. Soit $V \sim U(0,1)$
3. Si $V \leq \frac{f(X)}{c \cdot g(X)}$ alors accepter X, sinon rejeter et retourner à 1.

2.6 Simulations avec la méthode Monte-Carlo

La méthode de simulation Monte-Carlo utilise l'échantillonnage aléatoire pour étudier les propriétés de systèmes qui ont des composantes qui se comportent de façon stochastique [28]. Nous utilisons l'ordinateur pour générer des observations suivant une loi de probabilité qui modélise les fonctionnalités du phénomène étudié et permet ensuite de faire de l'inférence statistique.

Par exemple, dans un centre d'appels téléphoniques, supposons que W est le nombre de clients qui attendent plus de x minutes durant une journée (une variable aléatoire) et on veut estimer $E[W]$ sur une longue période. Pour cela, on simule n réalisations de la journée, on note W_i la réalisation de W pour la simulation i, et on estime $E[W]$ par la moyenne des W_i, soit $\bar{W}_n$. Ainsi, nous pouvons modéliser le nombre d'appels reçus dans un intervalle de temps (t_i, t_j) comme étant une variable aléatoire qui suit la loi de Poisson avec une moyenne λ_{ij}, et nous pouvons modéliser le temps de service de chaque appel comme étant une variable aléatoire qui suit une loi exponentielle avec un paramètre β. Nous pouvons implémenter le modèle avec un programme informatique qui simule les arrivées durant une unité de temps (une journée). Nous répétons la réalisation n fois pour obtenir les temps moyens des attentes W_i pour $0 \leq i < n$. Enfin, on obtient une estimation de l'espérance mathématique de la statistique recherchée comme suit :

$$\bar{W}_n \approx \mathbb{E}(W_n) = \frac{1}{n} \sum_{i=1}^{n} W_i.$$

D'autres statistiques peuvent être calculées, comme par exemple l'intervalle de confiance et la variance des observations, souvent soutenus par des histogrammes.

La méthode Monte-Carlo peut être utilisée dans l'estimation de l'intégrale d'une fonction f qui n'a pas de forme explicite comme dans le cas de la fonction de densité de probabilité (PDF) de la loi Normale :

$$I(f) = \int_0^c \frac{1}{\sqrt{2\pi}} e^{-x^2} dx = \Phi(c) - 1/2.$$

Pour approximer cette intégrale, il suffit de générer n variables aléatoires i.i.d sur $\mathscr{U}(0, C)$ puis d'estimer l'intégrale par $Q_n = \frac{c}{n} \sum_{i=1}^{n} f(x_i)$. On aura [28] :

$$\mathbb{E}(Q_n) = \frac{c}{n} \sum_{i=1}^{n} \mathbb{E}(f(X_i)) = c \int_0^c \frac{f(x)}{c} dx = I(f).$$

2.6.1 Nombres Aléatoires Communs

La méthode de simulation Monte-Carlo permet aussi de comparer plusieurs configurations d'un même modèle afin de savoir quels sont les meilleurs paramètres qui optimisent la statistique étudiée ou pour savoir l'effet d'une modification de paramètres sur le système. Par exemple si l'on voulait savoir l'impact du recrutement d'un nouvel agent au niveau d'un centre d'appels sur le temps d'attente des clients, nous pourrions exécuter une première fois la simulation avec le système à son état original puis simuler une deuxième fois avec l'ajout de la nouvelle ressource ; nous calculerions enfin l'espérance mathématique de la différence des temps d'attente entre les deux simulations.

Supposons que l'on veut estimer la différence entre $\mathbb{E}(X_1)$ et $\mathbb{E}(X_2)$ par la différence entre X_1 et X_2 qui sont des mesures calculées à partir de deux systèmes similaires. Nous pouvons soit utiliser exactement la même séquence de nombres aléatoires aux mêmes endroits ou utiliser des séquences totalement indépendantes. La première façon de faire est appelée technique des *nombres aléatoires communs* (CRN) et permet généralement d'introduire une covariance positive entre X_1 et X_2. La deuxième technique sera naturellement appelée technique des *nombres aléatoires indépendants* (IRN), et la variance de la différence des deux variables s'écrira de la façon suivante [18] :

$$Var[X_2 - X_1] = Var[X_2] + Var[X_1] - 2Cov[X_1, X_2].$$

Il est clair que l'utilisation des nombres en IRN permet d'annuler le terme de la covariance, ce qui donne une variance plus importante que la technique CRN dans le cas où la corrélation est positive (ce qui est vrai sous certaines conditions, voir section 4.3 de [28]).

CHAPITRE 3

LES SYSTÈMES HÉTÉROGÈNES ET LES GÉNÉRATEURS PARALLÈLES

Ce chapitre décrit les systèmes hétérogènes et les contraintes qu'ils présentent dans le développement des applications. Seront aussi décrites les différentes techniques de génération parallèle de nombres aléatoires ainsi que les critères de qualité dans ce contexte.

3.1 Les systèmes hétérogènes

De nos jours, les fabricants de matériel informatique produisent des dispositifs de calcul hétérogènes intégrant des processeurs avec des jeux d'instructions différents tout en cachant la dissimilarité aux utilisateurs. Plusieurs consortiums tels que *khronos* ou *hsafoundation* ont récemment vu le jour afin de mettre en place des spécifications et définir des standards de développement ouvert qui permettent d'exploiter tout le potentiel des matériels hétérogènes.

Nous pouvons maintenant exploiter les cartes graphiques GPGPU (General Purpose Graphical Processing Unit) pour effectuer des traitements parallèles sur de gros volumes de données, chose qui se faisait habituellement au niveau du CPU. Ce dernier continuera à s'occuper de la gestion du système d'exploitation et déléguera les traitements intensifs aux périphériques de calcul. La conception de systèmes informatiques modernes doit profiter de ces plates-formes hétérogènes afin d'offrir aux utilisateurs un niveau de performance élevé.

La figure 3.1 schématise la structure d'un PC avec plusieurs CPUs de différents types, une unité de traitement graphique (GPU) ainsi qu'un contrôleur d'entrées/sorties (ICH) et un contrôleur central graphique et mémoire (GMCH) permettant de connecter et de contrôler les dispositifs périphériques à la carte mère de l'ordinateur [32].

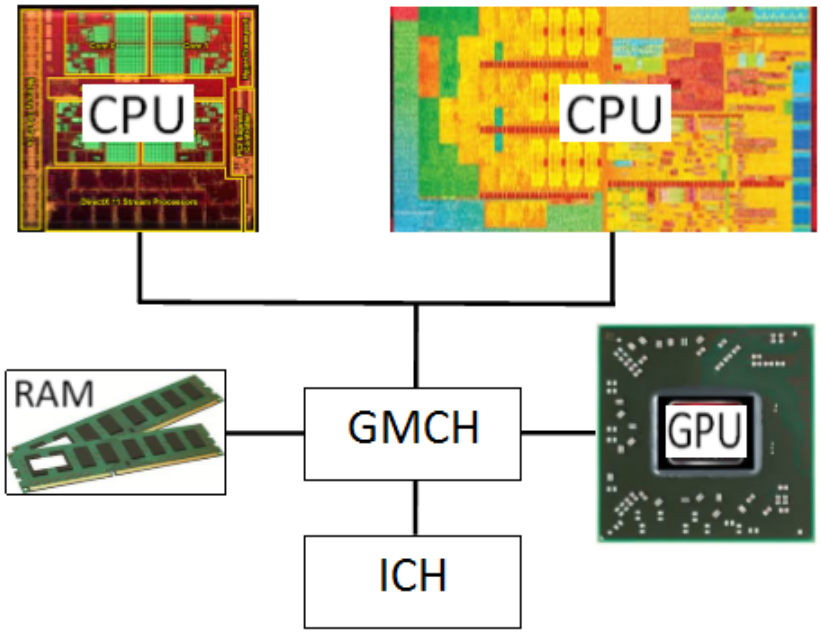

Figure 3.1 – Diagramme d'un PC avec plusieurs CPUs et un GPU

3.2 Exemples de systèmes parallèles

Il existe différents dispositifs de calcul parallèle qui peuvent être utilisés pour produire des nombres aléatoires, chacun ayant une architecture qui le rend plus adapté pour être utilisé avec certaines techniques de générations. Par exemple, la méthode de rejet est plus adaptée en CPU qu'en GPU à cause du branchement utilisé par l'algorithme qui va générer de la divergence d'exécution au niveau des processus parallèles (voir prochain chapitre). Cependant, la méthode de transformation est plus performante au niveau des FPGAs et GPUs [36]. Voici ci-dessous la description de quelques-uns de ces dispositifs :

— Unité de traitement centrale multi-noyaux (multi-core CPU) :
Pour améliorer la performance au niveau des CPUs plusieurs techniques sont utilisées comme le *parallélisme d'instructions* qui permet d'augmenter le taux avec lequel les instructions s'exécutent en même temps. Le *parallélisme de tâches* quant à lui cherche à augmenter le nombre de processus qui s'exécutent simultanément en intégrant plusieurs CPUs sur un seul circuit électronique ; chacun

exécute une tâche à la fois et possède son propre ensemble de registres internes avec une *unité arithmétique et logique* (ALU) pour effectuer les opérations. Tous les noyaux partagent une vue cohérente de la mémoire centrale du système. Cela correspond aux CPUs multi-noyaux.

— Unité de traitement graphique d'usage général (GPGPU) :

Les GPU sont de plus en plus exploités au-delà des traitements traditionnels d'infographie ; nous pouvons les utiliser aussi pour faire du calcul parallèle de façon souvent plus performante qu'avec les CPUs. La carte graphique GPGPU a une architecture qui permet d'exécuter des milliers de processus en parallèle (work items) qui sont organisés sous forme de groupes appelés *wraps* (AMD appelle ces groupes *wavefronts*) et qui exécutent tous la même instruction mais chaque work item traite une donnée différente selon le modèle *Single-Instruction Multiple Data* (SIMD).

— Unité de traitement accéléré (APU) :

C'est simplement un processeur qui intègre sur le même circuit un CPU et un GPU qui se partagent une mémoire cache. Cette combinaison permet d'améliorer la performance de calcul, en effet, le GPU nous permet de bénéficier de la puissance de calcul parallèle directement au niveau du processeur. Cependant le GPU intégré dans l'APU reste inférieur en performance à un GPU discret construit sur sa propre carte graphique.

— Circuit logique programmable (FPGA) :

C'est un circuit intégré composé de nombreux blocs logiques de mémoires (cellules) qui peuvent être reprogrammés (reconfigurés) après leur fabrication pour effectuer une fonction donnée. L'intérêt est qu'un même circuit puisse être réutilisé dans différents systèmes électroniques.

— Réseau de processeurs massivement parallèles (MPPA) :

C'est un autre type de circuit intégré contenant un grand nombre de processeurs organisés sous forme d'une grille rectangulaire et qui exécutent en parallèle des tâches selon l'architecture MIMD (*Multiple-Instruction Multiple-Data*). L'échange des données entre processeurs se fait à travers des interconnexions

configurables, ce qui permet d'effectuer en parallèle plus de traitements qu'un circuit conventionnel.

3.3 Contraintes avec le développement sur les systèmes hétérogènes

Le développement d'applications dans un environnement hétérogène soulève plusieurs défis au niveau des pratiques de programmation qui n'existent pas dans une architecture homogène [11]. Ces pratiques doivent être prises en compte dans le développement des générateurs de nombres aléatoires pour qu'ils soient portables sur différents types d'architecture, parmi ces défis on peut citer :

— Jeux d'instructions différents : les dispositifs de calculs peuvent avoir des jeux d'instructions différents, ce qui génère de l'incompatibilité lors de l'exécution.
— Différentes interprétations de la mémoire : les systèmes peuvent interpréter le positionnement des données dans la mémoire selon différentes façons ; une architecture petit-boutiste (little-endian) par exemple organise un mot mémoire avec l'octet de poids faible en premier alors qu'une architecture gros-boutiste fait l'inverse. Cela peut causer un problème de portabilité du code, car selon l'architecture, nous n'obtiendrons pas la même donnée.
— Indisponibilité des APIs systèmes : une application peut avoir besoin d'utiliser une librairie de fonctions qui est disponible dans un système mais qui peut ne pas l'être dans un autre système.
— Indisponibilité des options des langages de programmation : certaines plates-formes n'offrent pas la possibilité d'utiliser des structures de données disponibles dans les langages de programmation ; par exemple, les dispositifs de calcul compatibles avec la norme OpenCL ne supportent pas les pointeurs de fonctions.

3.4 Techniques de génération parallèle de nombres aléatoires

Les techniques discutées dans cette section servent à partager une longue séquence de nombres aléatoires entre plusieurs processeurs qui s'exécutent en parallèle.

3.4.1 Utilisation d'un serveur central

Une première façon de partager une longue séquence est d'utiliser un générateur central qui distribue à la demande des nombres aléatoires pour chaque processeur. Cette idée simple a l'inconvénient de ne pas permettre la reproductibilité des résultats des simulations vu que - d'une exécution à l'autre - les nombres distribués pourront être utilisés différemment (e.g., par différents processus). Cette technique peut aussi créer un goulot d'étranglement au niveau du serveur en cas où il existe beaucoup de processeurs.

3.4.2 Fractionnement de séquence

Cette technique consiste à partitionner la séquence principale en des blocs contigus de taille égale et qui ne se chevauchent pas, chacun ayant une période assez longue pour qu'aucune simulation ne puisse la consommer totalement. Étant donné une séquence de nombres aléatoires X_i nous aurons la sous-séquence d'ordre j et de longueur m qui a la forme suivante : $X_{k+(j-1)m}$ avec $k = 0, .., m-1$. Il faut s'assurer qu'il n'existe pas de corrélations apparentes entre les différentes sous-séquences pour ne pas avoir des résultats biaisés dans les simulations. Les générateurs doivent aussi implémenter la fonctionnalité de saut en avant pour une meilleure utilisation de cette technique [25, 31].

3.4.3 La technique de saut (Leapfrog)

Cette méthode permet la création de sous-séquences sur N processeurs de sorte que le processeur d'ordre P reçoive les nombres suivants $X_P, X_{P+N}, X_{P+2N}, etc.$ de la même manière qu'un jeu de cartes est distribué tour à tour entre des joueurs [8]. Cela nécessite la possibilité de pouvoir sauter N états en avant. Cette technique implique aussi que si le nombre des processeurs change, alors la séquence assignée à chacun sera différente. Cela pose problème si l'on veut reproduire exactement la même simulation.

3.4.4 Paramétrisation de générateurs indépendants

Contrairement aux techniques précédentes dans lesquelles une seule séquence est divisée entre plusieurs processeurs, cette technique propose d'assigner à chaque processeur

le même type de générateur mais avec un état initial (germe) différent. Ce dernier doit être choisi de façon aléatoire et indépendante (e.g. à travers un autre RNG) afin d'éviter qu'une corrélation ne se produise entre les séquences générées. Il faut aussi s'assurer que le type de générateur utilisé possède une assez longue période pour éviter le chevauchement des séquences. Cette méthode doit être implémentée de sorte qu'elle soit indépendante du nombre de processeurs pour assurer la reproductibilité des simulations.

3.5 Critères de qualité des générateurs parallèles

En plus des conditions citées dans la section 2.3, plusieurs auteurs ont défini d'autres critères nécessaires à un bon RNG dans le contexte parallèle [1, 8, 23] :

- Le générateur doit pouvoir produire des séquences de nombres aléatoires pour n'importe quel nombre de processeurs parallèles.
- Chacune des séquences parallèles doit satisfaire aux critères de qualité exigé pour un générateur utilisé en série.
- Il ne doit pas y avoir de corrélations ni dans la même séquence ni entre les séquences qui s'exécutent simultanément sur les processeurs.
- Les séquences parallèles doivent être complètement indépendantes les unes des autres, la technique de fractionnement par exemple permet d'assurer cela.
- L'implantation des RNGs doit être efficace et éviter le transfert de données entre processeurs ; autrement dit, chaque générateur doit être assigné à un seul processeur, ce dernier devant pouvoir produire sa propre séquence de façon indépendante afin d'assurer la reproductibilité de l'exécution.

Dans le cas des dispositifs de calcul de type GPU, certains auteurs [31] recommandent la conception d'algorithmes de génération de nombres aléatoires qui utilisent les nombres en virgules flottantes en simple précision, parce que le type `double` nécessite plus de temps de calcul ou n'existe pas au niveau de tous les dispositifs. Cependant cet inconvénient peut disparaître avec les futurs matériels. Une autre recommandation est d'éviter l'utilisation de la mémoire globale dans le cas d'accès répétitif aux données

afin d'optimiser le temps de calcul. L'auteur dans [29] précise que la taille de l'état du RNG devient un élément critique et doit être assez petit vu que l'espace mémoire des éléments de traitement (PE) au niveau du GPU est assez restreint comme il sera décrit dans le prochain chapitre.

CHAPITRE 4

ARCHITECTURE D'OpenCL

Ce chapitre décrit les différents concepts et éléments composant l'environnement OpenCL et détaille ses modèles sous-jacents.

4.1 Le modèle de la plate-forme

Une application développée avec OpenCL est composée généralement d'un module principal qui s'exécute sur un ordinateur central appelé *hôte* et qui est écrit en langage de haut niveau (C, C++, Java, etc.) ainsi qu'un autre module contenant un ensemble de fonctions appelées *kernels* écrites en langage OpenCL C et destinées à s'exécuter en parallèle sur les dispositifs de calcul. Ces kernels sont compilés à la volée (JIT) par le programme sur l'hôte pour générer un code binaire spécifique au dispositif cible, ce qui permet de rendre le code portable sur les systèmes hétérogènes compatibles OpenCL.

Le modèle de plate-forme de la norme permet au programme du hôte de configurer et d'utiliser plusieurs dispositifs en même temps, chacun effectuant une tâche spécifique grâce à des milliers de processus appelés **work items** qui s'exécutent simultanément dans des unités de calculs. Ce *parallélisme des tâches* est illustré dans la figure 4.1 ; notons que le *hôte* affecte les noyaux et les commandes aux dispositifs via des files d'attente (Command Queues), le tout étant géré à travers une structure de données appelé *contexte*.

L'API d'OpenCL offre aussi différents types de commandes qui permettent par exemple à l'hôte de lire ou d'écrire des blocs de données (*buffers*) dans la mémoire du dispositif ou de synchroniser l'exécution des work items entre eux. Il existe d'autres commandes qui permettent de faire le profilage de l'exécution des processus parallèles. OpenCL définit une structure de données appelée *plate-forme* qui permet de lister tous les dispositifs de calcul installés et d'identifier les attributs et les fonctionnalités qui sont supportées.

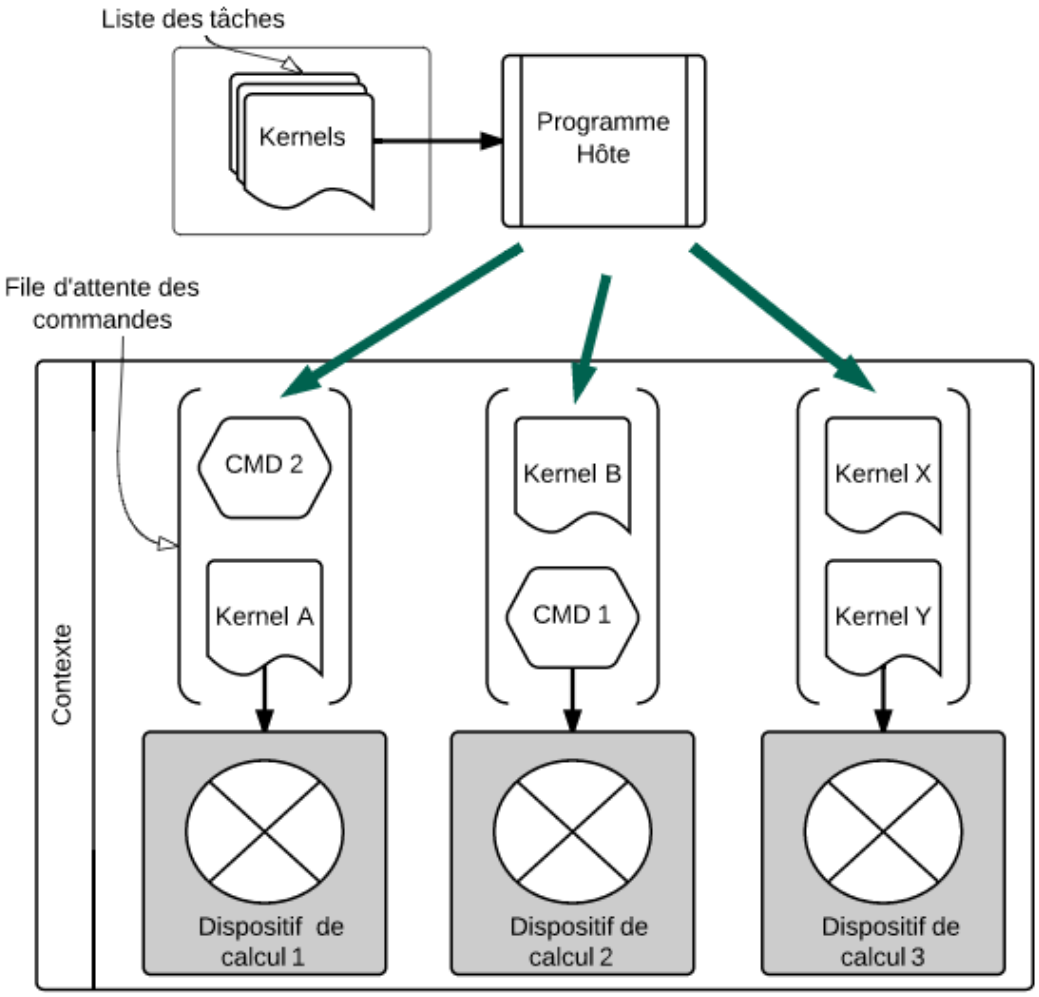

Figure 4.1 – Affectation des kernels et des commandes aux différents dispositifs

La figure 4.2 montre les principales composantes d'un dispositif compatible avec OpenCL. Notons qu'il y a quatre unités de calcul (CU) qui se partagent une mémoire globale et exécutent indépendamment les commandes venant de l'*hôte*. Chaque CU est composé d'un ensemble d'*éléments de traitement* (PE) qui se partagent une mémoire locale et exécutent tous le même code, mais les calculs effectués diffèrent par les données traitées. Ce type d'exécution on mode SIMD correspond au *parallélisme de données*.

Les étapes d'exécution d'un programme OpenCL :

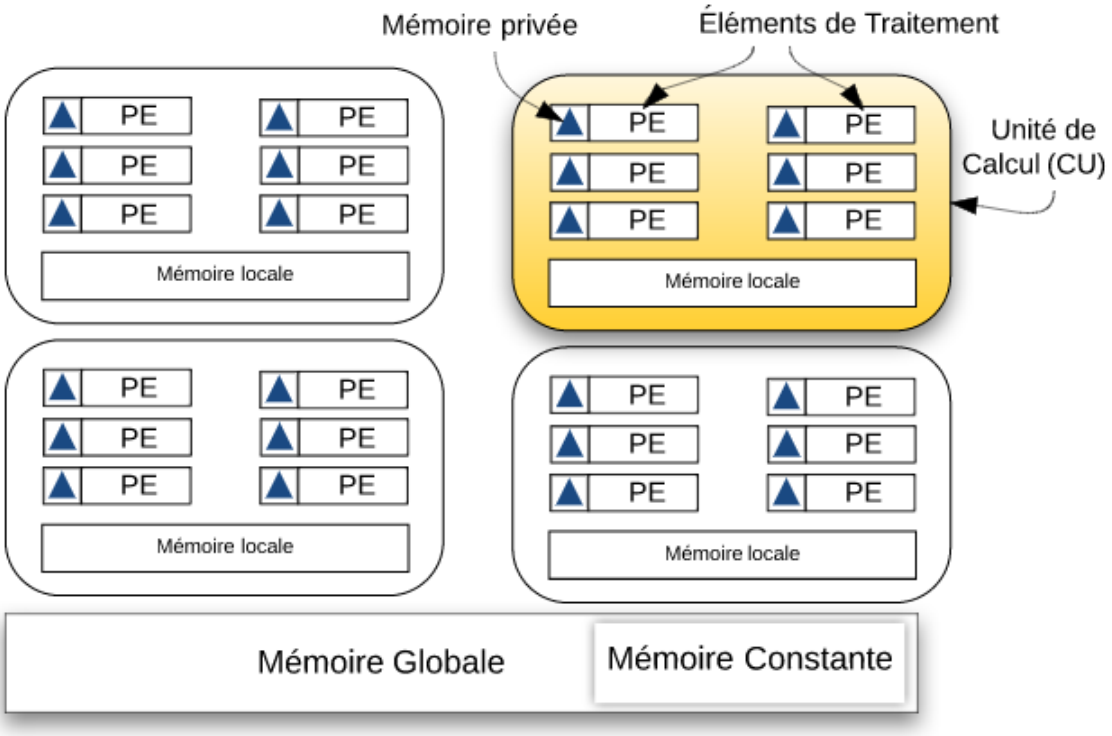

Figure 4.2 – Un dispositif de calcul avec 4 CU

Pour faire exécuter un kernel au niveau d'un dispositif, le programme hôte effectue les étapes suivantes :

1. Sélection de la plate-forme, du dispositif et du contexte.
2. Chargement du programme source du kernel en mémoire du système.
3. Compilation de ce code avec le compilateur OpenCL C.
4. Création de l'objet de type kernel.
5. Mise en place du kernel dans la file d'attente du dispositif.
6. Mise en place des arguments du kernel.
7. Lancement du kernel.
8. Récupération du résultat à la fin de l'exécution.

4.2 Le modèle d'exécution OpenCL

L'architecture présentée dans la section précédente permet aux dispositifs d'avoir une puissance de calcul souhaitée pour le traitement de grand volume de données, cela grâce à l'exécution parallèle des work items au niveau des unités de traitement. Il faut savoir que le dispositif de calcul divise les work items en plusieurs groupes de même taille appelés *work groups* ; chacun va s'exécuter sur une seule unité de calcul (CU). Cependant plusieurs CUs peuvent exécuter différents work groups en même temps.

Les work items peuvent être organisés sur plusieurs niveaux (dimensions) afin de mieux manipuler les structures de données qui peuvent, elles aussi, être sur plusieurs niveaux. Par exemple, pour traiter les données d'un simple tableau, nous pouvons lancer (sur une seule dimension) autant de work items qu'il existe d'éléments dans le tableau et chaque work item aura un seul identifiant unique (global ID) qui pourra être utilisé comme indice vers l'élément du tableau. Si, par contre, nous voulons appliquer un traitement sur chacun des points d'une image 2D ayant une taille d'un mégapixel, alors on peut configurer le dispositif pour lancer un million de work items organisés en deux dimensions (soit 1024 WI par dimension). Ainsi, chaque work item aura deux indices globaux qui lui permettront d'accéder au bon pixel de l'image. Cette structuration des work items en dimensions est appelée *NDRange* ; on utilise la fonction OpenCL `clEnqueueNDRangeKernel(...)` pour lancer le kernel en spécifiant le nombre total de work items, la taille du work group et la dimension à utiliser.

La figure 4.3 montre l'organisation de 20 work items sur deux dimensions. Chacun des 4 work groups sera affecté à un CU et chacun des work items sera exécuté sur un seul PE ; les valeurs (x, y) correspondent à des indices globaux des WI et permettent d'accéder au bon élément de la structure de données qui est manipulée [32] .

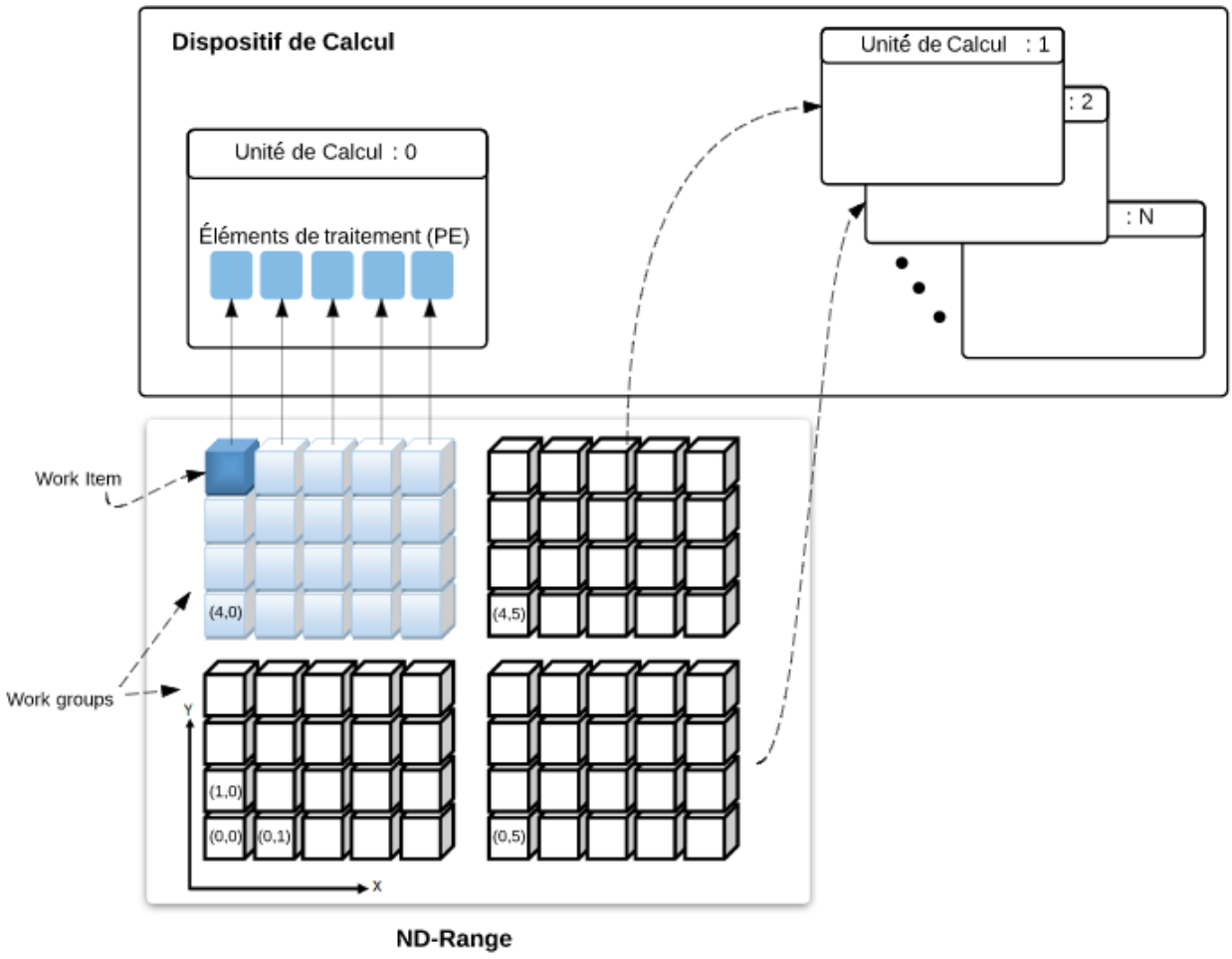

Figure 4.3 – Modèle d'exécution

4.3 Le modèle de la mémoire OpenCL

OpenCL définit quatre types de mémoires qui peuvent être utilisés par les work items et qui diffèrent par leur taille, leurs performances et leurs emplacements sur la plate-forme :

— **La mémoire globale :** chaque dispositif a une seule *mémoire globale* qui a la plus grande taille par rapport aux autres types et qui est accessible en lecture/écriture à partir de l'hôte ainsi qu'à tous les work items s'exécutant sur le dispositif.

— **La mémoire constante :** cette mémoire est similaire au type précédent ; cepen-

dant, les données qui y sont transférées sont accessibles seulement en lecture. Par contre, cette mémoire a l'avantage d'être plus rapide d'accès pour les work items que la mémoire globale.

— **La mémoire locale :** chaque unité de calcul possède une mémoire locale qui est plus petite en taille que les deux derniers types mais offre aux work items un accès ($\sim 100x$) plus rapide qu'avec la mémoire globale [33]. Les données transférées au niveau de cette mémoire peuvent être partagées par les work items d'un même work group mais cela nécessite l'utilisation de commandes de synchronisation (`barrier`) pour s'assurer de la cohérence des données[2]. Voir le listing 6.15 pour un exemple concrèt.

— **La mémoire privée :** chaque élément de traitement (PE) offre au work item qui s'exécute dessus une mémoire privée de petite taille sous forme de plusieurs registres ; cette mémoire est la plus rapide d'accès et permet d'avoir la meilleure performance.

Il faut noter que le programme de l'hôte ne peut transférer les blocs de données (buffers) qu'au niveau de la mémoire globale ou locale. La mémoire privée peut recevoir uniquement des données de type primitif. Le code du kernel doit contenir la logique de renvoi de tout résultat vers la mémoire globale vu que l'hôte ne peut récupérer les données qu'à partir de ce niveau-là. Notons aussi que plus la taille de la mémoire est grande, plus elle est placée loin de l'endroit d'exécution des work items (i.e, PE) et donc plus l'accès aux données sera coûteux en temps.

Le dispositif OpenCL comparé à un établissement scolaire :

Une analogie intéressante [33] peut se faire entre l'architecture d'un dispositif OpenCL et la structure d'un établissement scolaire dans lequel on y trouve un tableau d'affichage global (mémoire globale) accessible à tous les écoliers (work items). Ces derniers sont divisés en plusieurs groupes (work groups), chacun affecté à une classe différente (unité de calcul) ayant un tableau noir (mémoire locale) qui est accessible aux écoliers d'un même groupe et qui est utilisé pour partager des données entre les élèves de ce groupe seulement ; chaque écolier possède son propre cahier de notes (mémoire privée) qui est

utilisé pour sauvegarder ses données personnelles ; un professeur (hôte) peut donner un exercice à faire (kernel) à tous les élèves qui vont travailler simultanément et de façon indépendante à le résoudre.

4.4 Stratégies d'optimisation :

Dans ce qui suit, nous donnons quelques bonnes pratiques qui permettent d'améliorer la performance de l'exécution dans un dispositif OpenCL :

— **Accès séquentiel à la mémoire (Coalescing)** :

Il faut savoir que l'accès à la mémoire globale se fait par blocs d'espaces contigus de taille fixe, c.-à-d. qu'à la $i$ème lecture de la mémoire, le dispositif va toujours récupérer k éléments $x_i, .., x_{i+k}$ en même temps. Pour cela, il est recommandé que chaque work item accède à l'élément de données qui a le même indice (gid [1]) que le sien. Par exemple, supposons que $k = 4$ et que nous avons lancé lancé 4 work items qui vont lire chacun un élément d'un tableau, alors si nous avons fait de l'accès séquentiel, le dispositif va effectuer un seul accès à la mémoire, mais si chaque work item i accède à l'élément $i+4$, alors le dispositif va devoir lire quatre blocs de la mémoire pour récupérer les quatre éléments.

— **Utilisation efficace de la mémoire** :

Quand la nature des données et des tâches effectuées par le kernel le permet, il faut exploiter efficacement la hiérarchie de la mémoire cela en privilégiant d'abord l'utilisation de la mémoire privée avant l'utilisation de la mémoire locale ou de la mémoire globale, le tout pour éviter la latence d'accès qui caractérise les deux derniers types de mémoires.

— **Utilisation des vecteurs** :

Un *vecteur* est un type de données similaires au tableau, mais ayant des tailles prédéfinies par la norme OpenCL. L'avantage qu'il y a à utiliser cette structure de données est de pouvoir appliquer une opération de calcul sur tous les éléments du vecteur en une seule instruction. Par exemple, le résultat du calcul du fragment

1. Index qui identifie le work item de façon unique et globale

de code suivant correspond à la division par deux des 4 éléments du vecteur *X* en une seule commande, ce qui permet un gain important en performance :

```
float4 X = (float4)(3.0f, 6.0f, 9.0f, 11.0f);
float4 resultat = X * 0.5f;
```

— **Divergence des work items** :

Nous avons vu qu'OpenCL impose que les work items d'un même work group effectuent la même instruction en même temps (SIMD). Cependant, il faut savoir qu'au moment de l'exécution, le dispositif va diviser encore plus le work group en plusieurs ensembles de taille prédéfinie appelés *wraps*. Par exemple, si nous lançons 1024 work items sur 4 work groups, alors chaque unité de calcul va exécuter 32 work items (warp) à la fois jusqu'à ce que le work group soit terminé. La taille du warp ne peut pas être modifiée par le code, elle est fixée par le constructeur du dispositif.

La divergence d'exécution apparaît dans le cas où il y aurait une instruction de branchement conditionnel au niveau de la logique du kernel, de sorte qu'une partie des work items du warp l'évalue à *Vrai* alors que l'autre partie l'évalue à *Faux*. Dans ce cas le CU va exécuter les deux branches l'une à la suite de l'autre. Cela constitue une perte de parallélisme et implique un ralentissement de la performance ; c'est pour cela qu'il faut éviter d'utiliser ce type de branchement autant que possible.

Revenons à l'analogie de l'établissement scolaire et supposons qu'un instructeur (hôte) donne un questionnaire avec deux points à 32 étudiants (warp), avec la condition que si l'élève est un garçon, alors il doit répondre seulement au premier point sinon le deuxième point sera répondu. Dans le contexte réel, cela n'aurait constitué aucun problème, cependant, dans le cas d'un dispositif OpenCL où les instructions sont exécutées en mode SIMD, le warp va exécuter la partie conditionnelle du code (branche) à deux reprises ; lors de la première fois, les work items qui auront la condition à vrai vont exécuter la première branche alors que

les autres work items attendront leur tour pour exécuter la deuxième branche.

CHAPITRE 5

LES INTERFACES clRNG ET clPROBDIST

Ce chapitre contient une description détaillée des structures de données, des fonctions et des générateurs implémentés dans clRNG et clProbDist.

Il faut savoir que la compagnie AMD est parmi les instigateurs qui ont demandé au laboratoire de simulation de l'université de Montréal le développement de ces deux APIs. Cette compagnie a demandé le respect de quelques contraintes durant le développement ; ainsi les interfaces de programmation doivent être écrites en langage C et l'on doit utiliser la version 1.2 de l'environnement OpenCL. La raison derrière cela serait de garder le code le plus proche possible du matériel, ce qui permettra plus tard une meilleure intégration de ces librairies (APIs) avec d'autres langages de plus haut niveau. Nous pensons aussi que le code s'exécutera plus rapidement avec le langage C. Cependant, avec ces limitations, nous ne pouvions pas utiliser les classes et le polymorphisme vu que C99 ne le permettait pas et qu'OpenCL C n'offrait pas de pointeurs vers les fonctions. Puisque le langage C n'offrait pas aussi les espaces de noms (namespaces), nous avons dû inclure dans le nom de chaque fonction le nom du générateur implémenté, ce qui a donné des noms de fonctions assez longs. Vous noterez aussi qu'il y a plusieurs fonctions qui auraient pu être consolidées en une seule fonction, mais il n'y a pas moyen d'utiliser la surcharge de fonctions dans le langage C.

5.1 Définition de clRNG

clRNG est une API qui implémente différents types de générateurs uniformes suivant la même interface de programmation. Cela définit des fonctions et des structures de données qui permettent de générer une séquence de nombres pseudo-aléatoires qui peut être divisée en plusieurs segments (streams) et sous-segments utilisables dans des dispositifs OpenCL.

Un *stream* est implémenté par une structure de données qui contient trois éléments, à savoir, l'état initial du générateur (graine), son état courant et l'état initial du sous-stream courant. À la création de l'objet stream, clRNG initialise ces états internes avec des valeurs par défaut puis calcule la valeur de l'état initial du prochain stream qui est Z états plus loin. La structure de données correspondant à l'état interne varie selon le générateur.

Puisque la version utilisée d'OpenCL ne permet pas l'allocation d'espace mémoire directement à partir du kernel, nous avons dû séparer les fonctions de l'API en deux parties, celles qui s'exécutent uniquement au niveau de l'hôte - telles que pour la création des streams - et celles qui peuvent être utilisées aussi bien au niveau de l'hôte qu'au niveau du dispositif.

Typiquement, on utilise l'API pour créer au niveau de l'hôte plusieurs streams dans un tableau puis on les transfère vers la mémoire globale du dispositif. Cela permet à chaque work item d'utiliser son indice global (gid) pour sélectionner un stream (ayant le même indice dans le tableau) qui sera copié vers la mémoire privée, ce qui permet d'avoir une meilleure performance lors de la génération des nombres aléatoires.

Au niveau du dispositif, quand le stream est copié vers la mémoire privée du work item, clRNG transfère seulement l'état courant du stream ainsi qu'un pointeur vers l'état initial qui reste en mémoire globale. Quant à l'état initial du sous-stream courant il ne sera copié que si la macro `CLRNG_ENABLE_SUBSTREAM` est définie, cette commande permet aussi d'activer plusieurs fonctions de manipulation des sous-streams (telles que l'affectation de l'état du sous-stream courant à l'état courant du stream). Cette stratégie permet d'économiser l'espace mémoire restreint au niveau des PEs. L'utilisation de cette macro n'est pas nécessaire au niveau de l'hôte et les fonctions de manipulation de streams y sont toujours disponibles. Il faut aussi noter que, par défaut, l'API effectue les calculs des nombres en virgule flottante double précision, mais nous pouvons aussi générer les variables aléatoires en simple précision en utilisant la macro

`CLRNG_SINGLE_PRECISION`.

Pour pouvoir utiliser l'API dans un programme, il suffit d'inclure au début du programme qui s'exécute sur l'hôte (ou sur le dispositif) le fichier d'en-tête de l'un des générateurs implémentés ; par exemple, le code du listing 5.1 montre comment activer l'utilisation des sous-streams au niveau du kernel avec le générateur MRG32k3a pour produire des variables aléatoires en virgule flottante simple précision.

```
1  #define CLRNG_ENABLE_SUBSTREAMS
2  #define CLRNG_SINGLE_PRECISION
3  #include <mrg32k3a.clh>

5  __kernel void monKernel(__global clrngMrg32k3aHostStream* Streams,
6                                          __global double* result){
7  //Obtient l'index global du work item
8  int gid = get_global_id(0);

10 //Copie le stream de la mém. globale vers la mém. privée
11 clrngMrg32k3aStream monStream;
12 clrngMrg32k3aCopyOverStreamsFromGlobal(1, &monStream, &Streams[gid]);

14 //Génère et sauvegarde une var. aléatoire uniforme
15 result[gid] = clrngMrg32k3aRandomU01(&monStream);
16 }
```

Listing 5.1 – Exemple simple d'utilisation de clRNG dans un kernel

5.1.1 Les générateurs implémentés

5.1.1.1 MRG32k3a

C'est un MRG combiné implémenté avec un état de six entiers non signés de 64 bits ayant une période $\rho \approx 2^{191}$ qui est divisé en 2^{64} streams de longueur $Z = 2^{127}$. Chaque stream est divisé à son tour en 2^{51} sous-streams de longueur $W = 2^{64}$, le tout sans chevauchement (voir section 2.1.3). La version originale proposée dans [16] a une récurrence implémentée en `double`.Dans notre version, l'état est plutôt représenté par six entiers de 32 bits. Ce changement dans l'implémentation a été fait pour éviter l'utilisation des

nombres en virgule flottante double précision qui ne sont pas toujours disponibles dans les dispositifs de calcul.

5.1.1.2 MRG31k3p

Ce générateur défini dans [27] est un MRG constitué de deux composantes d'ordre 3, son état est constitué des deux vecteurs $s_{1,n} = (x_{1,n}, x_{1,n+1}, x_{1,n+2})$ et $s_{2,n} = (x_{2,n}, x_{2,n+1}, x_{2,n+2})$ qui suivent la récurrence linéaire suivante :

$$x_{1,n} = (2^{22} x_{1,n-2} - (2^7 + 1) x_{1,n-3}) \; mod \; m_1$$
$$x_{2,n} = (2^{15} x_{2,n-1} - (2^{15} + 1) x_{2,n-3}) \; mod \; m_2$$

avec $m_1 = 2^{31} - 1$ et $m_2 = 2^{31} - 21069$. La sortie u_n est définie par :

$$z_n = (x_{1,n} - x_{2,n}) \; mod \; m_1$$
$$u_n = \begin{cases} z_n / 4656612873 & \text{pour } z_n > 0 \\ m_1 / 4656612873 & \text{pour } z_n = 0. \end{cases}$$

MRG31k3p est implémenté avec un état composé de six entiers non signées de 31 bits. Sa période est approximativement égale à 2^{185} qui est divisé en 2^{51} streams de longueur $Z = 2^{134}$. Chaque stream est divisé à son tour en plusieurs sous-streams de longueur $W = 2^{72}$, cela sans aucun chevauchement.

5.1.1.3 LFSR113

L'état de ce générateur est composé de quatre entiers de 31 bits, sa période est approximativement de 2^{113} divisé en 2^{23} streams distincts de longueur $Z = 2^{90}$; chaque stream est divisé à son tour en 2^{35} sous-streams de longueur $W = 2^{55}$ [17].

5.1.1.4 Philox4x32-10

Ce générateur tel que conçu par ses auteurs [9] permet de produire des nombres aléatoires uniformes qui ont passé toutes les batteries de tests du logiciel TestU01. Cependant sa conception n'est pas supportée par une analyse théorique d'équidistribution. Le challenge a été d'adapter Philox à notre interface de programmation pour pouvoir générer des streams et sous-streams de v.a. $\mathscr{U}(0,1)$. Pour cela, nous avons dû premièrement analyser la fonction bijective utilisée pour convertir un simple entier *i* en une valeur aléatoire, ensuite nous avons conçu un objet *clrngPhilox4x32* qui contient un compteur de 128 bits ainsi qu'une clé composée de deux entiers non signés. À chaque incrémentation, notre API envoie la valeur du compteur ainsi que la clé vers la fonction bijective du générateur *Philox4x32-10* pour produire quatre valeurs aléatoires de type entier 32 bits. Ces entiers sont stockés en interne dans notre objet puis distribués l'un après l'autre jusqu'à ce que les quatre soient retournés à l'utilisateur. À la cinquième demande de nombre aléatoire, le compteur sera incrémenté une deuxième fois et quatre nouveaux entiers seront encore générés et ainsi de suite.

Le générateur contient aussi un index à 2 bits utilisé pour indiquer la prochaine valeur à distribuer parmi les 4x32 bits générés. La clé est la même pour tous les streams et elle est initialisée par défaut avec la valeur 0 ; l'implémentation actuelle de clRNG ne permet pas le changement de cette clé. La période du générateur est de 2^{128} qui est divisé en 2^{28} streams distincts de longueur $Z = 2^{100}$, chaque stream est subdivisé en 2^{36} sous-streams de longueur $W = 2^{64}$.

5.1.1.5 cLFSRMRG : Combinaison de LFSR113 et MRG31k3p

En s'inspirant du travail fait dans [10], nous avons défini un nouveau générateur de nombres aléatoires uniformes combinant un LFSR113 avec un MRG31k3p et ayant une plus longue période que chacun des deux générateurs pris à part. Le listing 5.2 montre la fonction permettant de générer la variable aléatoire combinée en utilisant un ou-exclusif bit-à-bit ; le résultat obtenu z est ramené à un nombre entre 0 et 1 en division par 2^{32}. Pour

pouvoir faire cela correctement, nous avons multiplié la sortie du MRG31k3p par 2^{32}. Le code de la fonction `clrngLfsr113_UL` correspond à `clrngLfsr113RandomU01` avec la différence que la valeur retournée est de type entier long non signé. Dans [15], nous trouvons la preuve théorique que la période de deux générateurs combinés est égale au produit des deux périodes de chaque générateur pris à part, ce qui donne pour cLFSRMRG une période $\rho \approx 2^{298}$.

```
1  #include <mrg32k3a.clh>
2  #include <lfsr113.clh>

4  #define DeuxExp32    4294967296                /* 2^32   */
5  #define norm         2.3283064365386962890e-10  /* 1/2^32 */

7  cl_double clLfsrMrgRandomU01(clrngMrg31k3pStream* stream1,
       clrngLfsr113Stream* stream2){

9  cl_ulong z= (cl_ulong) (DeuxExp32 * clrngMrg31k3pRandomU01(stream1))
10                                    ^ clrngLfsr113_UL(stream2);

12 return (z * norm);
13 }
```

Listing 5.2 – Génération de nombres aléatoires uniformes avec cLFSRMRG

À noter que nous avons implémenté seulement la fonction de création des nombres aléatoires uniformes, le reste des fonctions de l'API ne l'est pas encore et cLFSRMRG ne fait pas partie de la version publique de clRNG.

Le listing 5.3 montre le code que nous avons écris pour de tester statistiquement le nouveau générateur avec la librairie TestU01 ; notons à la ligne 10 l'utilisation de la commande `unif01_CreateExternGen01` pour créer un générateur externe qui produira les nombres aléatoires en utilisant `clLfsrMrgRandomU01`. Les lignes 16 à 18 montrent comment appliquer les batteries de tests Small Crush, Crush et Big Crush au nouveau générateur. Notons aussi la création des générateurs MRG31k3p et LFSR113 (lignes 21 à 22), ces derniers seront utilisés ainsi que le générateur combiné par la fonction `unif01_TimerSumGenWr` pour calculer le temps CPU nécessaire à la génération de 10^7 nombres aléatoires (lignes 25 à 27), les résultats sont illustrés par la figure 5.1.

Notons que le temps nécessaire à cLFSRMRG (1.23 secondes) pour générer ces nombres aléatoires est plus grand que la somme des deux générateurs réunis.

Nous avons exécuté le programme 5.3 sur un ordinateur avec un CPU de type AMD A10 2.10 GHz et ayant 8 GB de mémoire vive, ce qui a donné les résultats illustrés par la figure 5.1. Notons que la vitesse d'exécution du nouveau générateur est plus grande que la somme des vitesses d'exécution de chaque générateur pris à part. La figure 5.2 montre que cLFSRMRG a passé tous les tests du logiciel TestU01. Les temps d'exécution des batteries *Small Crush*, *Crush* et *Big Crush* sont respectivement de l'ordre de 42 secondes, 2 heures et 20 heures.

```
-------------  Results of speed test  ---------------

 Host:
 Generator:   Combinaison de LFSR113 avec MRG31k3p en XOR
 Method:      GetU01
 Mean =       0.499969507084592
 Number of calls:  10000000
 Total CPU time:   1.23 sec


-------------  Results of speed test  ---------------

 Host:
 Generator:   ulec_Createlfsr113
 Method:      GetU01
 Mean =       0.500154672454091
 Number of calls:  10000000
 Total CPU time:   0.41 sec


-------------  Results of speed test  ---------------

 Host:
 Generator:   ulec_CreateMRG31k3p
 Method:      GetU01
 Mean =       0.499910873779136
 Number of calls:  10000000
 Total CPU time:   0.59 sec
```

Figure 5.1 – Tests de performance de cLFSRMRG, LFSR113 et MRG31k3p

```c
#include "unif01.h"
#include "ulec.h"
#include "swrite.h"
#include "bbattery.h"
#include <stdio.h>

int main (void){

   unif01_Gen *gen1, *gen2, *gen3;
   gen1 = unif01_CreateExternGen01 ("Combinaison de LFSR113 avec
      MRG31k3p en XOR : ", clLfsrMrgRandomU01);

   //Affichage sommaire des résultats des tests statistiques
   swrite_Basic = FALSE;

   //Exécution des batteries de tests
   bbattery_SmallCrush (gen1);
   bbattery_Crush(gen1);
   bbattery_BigCrush(gen1);

   //Création des générateurs LFSR113 et MRG31k3p
   gen2 = ulec_Createlfsr113 (12345, 12345, 12345, 12345);
   gen3 = ulec_CreateMRG31k3p (123., 123., 123., 123., 123., 123.);

   //Test de performance, le param "True" permet d'utiliser GetU01()
      au lieu de GetBits()
   unif01_TimerSumGenWr (gen1, 10000000, TRUE);
   unif01_TimerSumGenWr (gen2, 10000000, TRUE);
   unif01_TimerSumGenWr (gen3, 10000000, TRUE);

   //Libération des ressources
   unif01_DeleteExternGen01 (gen1);
   unif01_DeleteExternGen01 (gen2);
   unif01_DeleteExternGen01 (gen3);
}
```

Listing 5.3 – Code utilisé pour tester le générateur combiné avec la librairie TestU01

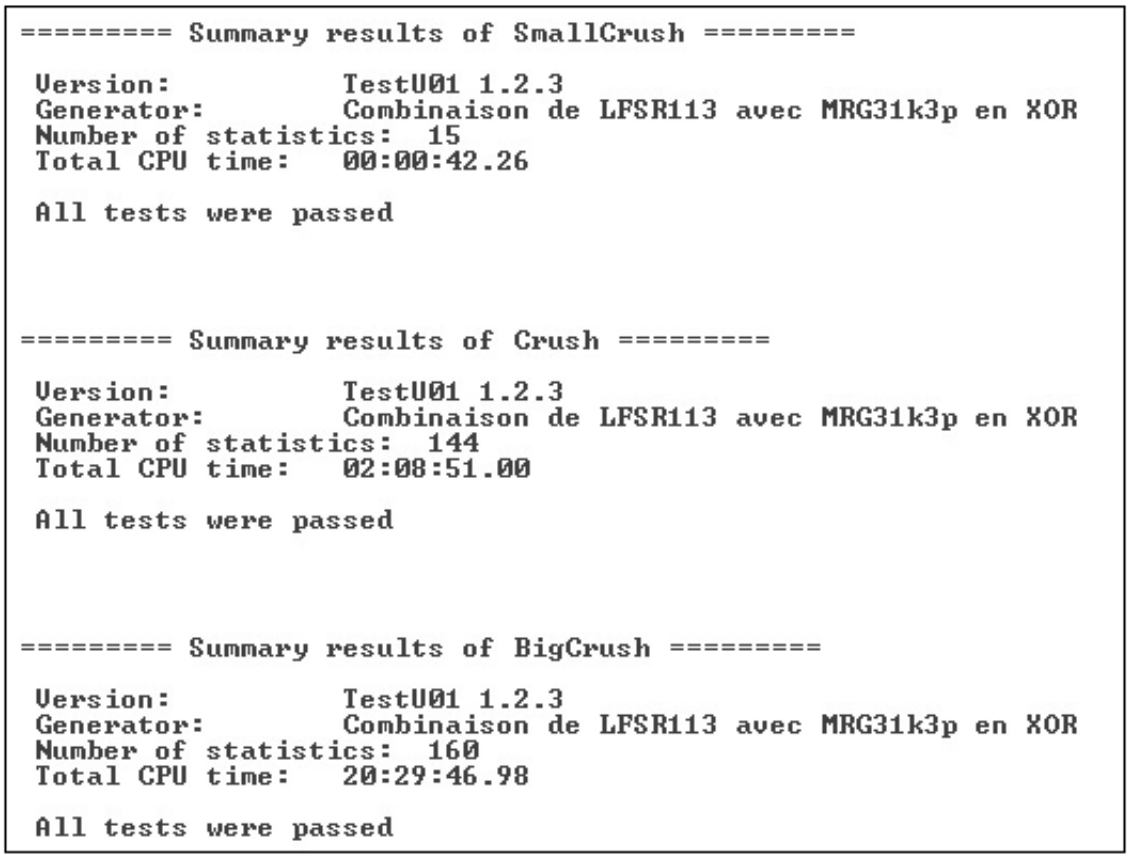

Figure 5.2 – Résultats des batteries de tests statistiques TestU01

5.1.2 L'interface de programmation clRNG

clRNG implémente plusieurs générateurs de nombres aléatoires qui possèdent tous le même ensemble de fonctions et structures de données et les noms d'objets débutent par un préfixe qui identifie le générateur implémenté. Afin de pouvoir faire une description générique de l'API, nous avons remplacé les préfixes des objets de l'API par `clrng`. Nous pouvons regrouper les éléments de l'interface en quatre catégories :

1. Les structures de données
2. Les fonctions qui gèrent la création et la destruction des streams
3. Les fonctions qui génèrent les variables aléatoires uniformes
4. Les fonctions qui permettent la manipulation des états des streams

5.1.2.1 Au niveau de l'hôte

Les structures de données

```
typedef struct { ... } clrngStreamState;
```

— Correspond à l'état d'un stream, sa structure interne dépend du type de générateur.

```
typedef struct { ... } clrngStream;
```

— C'est une structure de données qui contient les informations sur l'état initial du stream, son état courant et l'état initial du sous-stream courant. Pour transférer `clrngStream` vers un kernel, nous définissons une structure similaire appelée `clrngHostStream` qui peut être utilisée pour copier le stream de la mémoire globale vers la mémoire privée. Cette copie contiendra seulement l'état courant et un pointeur vers l'état initial du stream (qui reste en mémoire globale). En plus, si le préprocesseur `CLRNG_ENABLE_SUBSTREAMS` est utilisé, alors l'état du sous-stream courant sera aussi copié en mémoire privée.

```
typedef struct { ... } clrngStreamCreator;
```

— Cette structure contient la valeur par défaut de l'état initial du stream, l'état initial du prochain stream `nextState` ainsi que des matrices qui permettent de faire les sauts en avant et qui sont utilisées dans le cas des générateurs de type MRG seulement [12]. À chaque création d'un stream, l'interface de programmation utilise un `clrngStreamCreator` par défaut pour définir la valeur de l'état initial du stream ainsi que son état courant et l'état du sous-stream courant, puis fait avancer `nextstate` Z états vers le prochain stream en utilisant les matrices de sauts. Cela permet - lors de la prochaine création de stream - d'avoir l'état initial du prochain stream déjà calculé. L'API offre aussi à l'utilisateur la possibilité de définir son propre créateur de streams.

```
typedef enum { ... } clrngStatus;
```

— La plupart des fonctions retournent un pointeur vers un objet `err` de type *clrngStatus* qui indique le succès ou l'erreur de l'exécution.

Création et destruction des streams

```
clrngStreamCreator* clrngCopyStreamCreator(const clrngStreamCreator*
                                           creator, clrngStatus* err);
```

— Cette fonction permet de créer une copie identique du créateur transmis comme paramètre (`creator`); si ce dernier est `null`, alors la fonction renvoie une copie du créateur par défaut.

```
clrngStatus clrngDestroyStreamCreator(clrngStreamCreator* creator);
```

— Cette fonction libère les ressources associées avec l'objet `creator`

```
clrngStatus clrngRewindStreamCreator(clrngStreamCreator* creator);
```

— Réinitialise le créateur avec ses valeurs originales pour qu'il puisse recréer exactement les mêmes streams.

```
clrngStatus clrngSetBaseCreatorState(clrngStreamCreator* creator,
                                const clrngStreamState* baseState);
```

— Initialise la valeur de l'état initial et la valeur de `nextState` du créateur passé en paramètre avec la valeur contenue dans l'état de base `baseState` qui peut être différente de la valeur par défaut.

```
clrngStream* clrngAllocStreams(size_t count, size_t* bufSize,
                                              clrngStatus* err);
```

— Cette fonction alloue l'espace nécessaire pour `count` objets de type stream et retourne un pointeur vers la mémoire créée ; la fonction n'initialise pas les streams ; la taille de la mémoire alloué est retournée en octets dans `bufSize`.

```
clrngStatus clrngDestroyStreams(clrngStream* streams);
```

— Libère les ressources allouées aux objets `streams`

```
clrngStream* clrngCreateStreams(clrngStreamCreator* creator,
            size_t count,size_t* bufSize, clrngStatus* err);
```

— Cette fonction crée un tableau de `count` streams avec le créateur spécifié, cela en réservant la mémoire nécessaire et en initialisant chaque stream puis en retournant un pointeur vers le tableau créé. Un seul stream est créé si `count` vaut 1. La fonction utilisera le créateur par défaut si `creator` est null. `bufSize` contient la taille de l'espace alloué.

```
clrngStatus clrngCreateOverStreams(clrngStreamCreator* creator,
                           size_t count, clrngStream* streams);
```

— Cette fonction est similaire à `clrngCreateStreams` sauf qu'elle ne réserve pas de mémoire, mais réutilise l'espace déjà alloué aux objets `streams` ; cela est pratique si l'on veut exploiter la mémoire déjà allouée à d'autres streams.

```
clrngStream* clrngCopyStreams(size_t count, const clrngStream* streams,
                                                clrngStatus* err);
```

— Retourne une copie conforme de chacun des `count` streams passés en paramètre. Cette fonction fait la réservation de mémoire nécessaire aux copies puis retourne un pointeur sur la structure allouée.

```
clrngStatus clrngCopyOverStreams(size_t count,
                  clrngStream* destStreams, const clrngStream* srcStreams);
```

— Copie `count` streams de `srcStreams` vers `destStreams` ; cette fonction ne réserve pas d'espace mémoire.

```
clrngStream* clrngMakeSubstreams(clrngStream* stream,size_t count,
                                 size_t* bufSize, clrngStatus* err);
```

— Cette fonction retourne un tableau de streams en récupérant `count` sous-streams successifs de `stream`. Chaque élément du tableau aura son état initial et son état courant égaux à l'état initial du sous-stream lui correspondant. La fonction réserve l'espace mémoire et retourne la taille de l'espace alloué dans `bufsize`. Cette fonction a aussi pour effet d'avancer l'état initial et l'état du sous-stream courant de `stream` par `count` sous-streams en avant.

```
clrngStatus clrngMakeOverSubstreams(clrngStream* stream,
                                 size_t count, clrngStream* substreams);
```

— Similaire à `clrngMakeSubstreams` sauf que cette fonction ne réserve pas de mémoire mais réutilise l'espace déjà alloué à `substreams` qui a pu être créé par l'une des fonctions suivantes `clrngAllocStreams`, `clrngMakeSubstreams` ou `clrngCreateStreams`.

Génération des nombres aléatoires

```
cl_double clrngRandomU01(clrngStream* stream);
```

— Génère et retourne un nombre pseudo-aléatoire uniformément distribué sur l'intervalle (0,1) en utilisant `stream` ; l'état du stream est avancé avant de produire le nombre aléatoire. Le nombre généré est de type `cl_double` sauf si l'option `CLRNG_SINGLE_PRECISION` est utilisée. Dans ce cas, il sera de type `cl_float` pour tous les RNG. Cette option affectera aussi le type de retour de toutes les fonctions citées dans cette section et qui génèrent des nombres aléatoires.

```
clrngStatus clrngRandomU01Array(clrngStream* stream,
                                size_t count,cl_double* buffer);
```

— Remplit le tableau pré-alloué `buffer` avec `count` nombres aléatoires uniformes en appelant `clrngRandomU01` `count` fois.

```
cl_int clrngRandomInteger(clrngStream* stream,
                                        cl_int i, cl_int j);
```

— Génère et retourne un nombre pseudo-aléatoire uniformément distribué dans l'ensemble `{i,..,j}` en utilisant la méthode d'inversion suivante :
`i + (cl_int)((j-i+1) * clrngRandomU01(stream)`

```
clrngStatus clrngRandomIntegerArray(clrngStream* stream,
                  cl_int i, cl_int j, size_t count, cl_int* buffer);
```

— Similaire à `clrngRandomU01Array` mais retourne des entiers uniformément distribués dans l'ensemble `{i,..,j}`, ce qui équivaut à appeler `count` fois `clrngRandomInteger()` pour remplir le tableau `buffer`.

Manipulation des streams

```
clrngStatus clrngChangeStreamsSpacing(clrngStreamCreator* creator,
                                      cl_int e, cl_int c);
```

— Cette fonction doit être utilisée seulement dans des cas exceptionnels. Elle change la valeur par défaut Z de l'espacement entre les états initiaux des streams successifs en la nouvelle valeur : $Z = 2^e + c$ si $e > 0$ ou à $Z = c$ si $e = 0$ (nous devons toujours avoir $e \geq 0$ mais la valeur de c peut être négative). Il faut savoir que l'espacement par défaut a été soigneusement sélectionné pour chaque générateur afin d'éviter le chevauchement et la dépendance entre les streams ; il n'est pas recommandé de changer cet espacement. À noter que cette fonction n'est pas implémentée pour le générateur LFSR113.

```
clrngStatus clrngRewindStreams(size_t count, clrngStream* streams);
```

— Pour chaque stream dans `streams`, cette fonction réinitialise l'état courant et l'état initial du sous-stream courant avec l'état initial du stream.

```
clrngStatus clrngRewindSubstreams(size_t count, clrngStream* streams);
```

— Pour chaque stream dans `streams`, cette fonction réinitialise l'état courant du stream avec l'état initial du sous-stream courant.

```
clrngForwardToNextSubstreams(size_t count, clrngStream* streams);
```

— Pour chaque stream dans `streams`, cette fonction réinitialise l'état courant et l'état initial du sous-stream courant avec l'état initial du prochain sous-stream.

```
clrngStatus clrngAdvanceStreams(size_t count, clrngStream* streams,
                                              cl_int e, cl_int c);
```

— Cette fonction doit être utilisée seulement dans des cas exceptionnels. Cette fonction avance l'état courant de chaque stream dans `streams` par k étapes sans modifier ni l'état initial du stream ni l'état initial du sous-stream courant.

$$\begin{aligned} &Si \quad e > 0 \quad alors \quad k = 2^e + c \\ &Si \quad e < 0 \quad alors \quad k = -2^{|e|} + c \\ &Si \quad e = 0 \quad alors \quad k = c. \end{aligned}$$

À noter que c peut prendre des valeurs négatives. Il n'est pas recommandé d'utiliser cette fonction pour modifier la longueur des streams et sous-streams ; il vaut mieux utiliser l'espacement par défaut qui a été soigneusement sélectionné. La fonction n'est pas implémentée pour le générateur LFRS113.

```
clrngStatus clrngDeviceRandomU01Array(size_t streamCount,cl_mem streams,
          size_t nbrCount, cl_mem outBuffer,cl_uint numQueuesAndEvents,
                   cl_command_queue* commQueues, cl_uint numWaitEvents,
                      const cl_event* waitEvents, cl_event* outEvents);
```

— Remplit le tableau contenu dans l'objet mémoire `outBuffer` avec `nbrCount` nombres aléatoires de type `cl_double` (ou de type `cl_float` si l'option `CLRNG_SINGLE_PRECISION` est utilisée). Le tableau sera rempli au niveau du dispositif par `streamCount` work items. Le nombre de variables aléatoires générées doit être un multiple du nombre de work items. Le reste des paramètres correspond aux objets nécessaires pour l'exécution du kernel en utilisant la fonction `clEnqueueNDRangeKernel`

5.1.2.2 Au niveau du dispositif de calcul

Pour pouvoir utiliser la librairie peu importe l'endroit où elle a été déployée, nous avons eu recours aux variables d'environnement. Ainsi l'utilisateur doit ajouter au niveau de son système d'exploitation la variable d'environnement `CLRNG_ROOT` qui doit pointer vers le chemin du répertoire `src` qui existe dans le dossier d'installation de l'API. Le code du kernel doit bien sûr inclure le fichier d'entête du générateur utilisé (par exemple `MRG31k3p.clh`).

Manipulation des streams

```
clrngStatus clrngCopyOverStreams(size_t count,
          clrngStream* destStreams, const clrngStream* srcStreams);
```

— Cette fonction copie `count` streams de `srcStreams` vers `destStreams`, les deux objets se trouvent dans la mémoire privée du work item.

```
clrngStatus clrngCopyOverStreamsFromGlobal(size_t count,
                                           clrngStream* destStreams,
                     __global const clrngHostStream* srcStreams);
```

— Cette fonction copie `count` streams de `srcStreams` qui se trouve en mémoire globale vers `destStreams` qui se trouve en mémoire privée du work item.

```
clrngStatus clrngCopyOverStreamsToGlobal(size_t count,
                              __global clrngHostStream* destStreams,
                                     const clrngStream* srcStreams);
```

— Copie de la mémoire privée `count` streams de `srcStreams` vers `destStreams` qui se trouve en mémoire globale.

```
clrngStatus clrngRewindStreams(size_t count, clrngStream* streams);
```

— Similaire à la version s'exécutant au niveau de l'hôte, cette fonction peut être lente au niveau du dispositif parce qu'elle récupère l'état initial des streams à partir de la mémoire globale.

Note : Les fonctions suivantes sont les mêmes que celles qui s'exécutent au niveau de l'hôte mais elles sont disponibles seulement quand l'option `CLRNG_SINGLE_PRECISION` est utilisée.

```
clrngStatus clrngRewindSubstreams(size_t count, clrngStream* streams);

clrngStatus clrngForwardToNextSubstreams(size_t count,
                                         clrngStream* streams);

clrngStatus clrngMakeOverSubstreams(clrngStream* stream,
                            size_t count, clrngStream* substreams);
```

Génération de nombres aléatoires sur le dispositif de calcul

Les fonctions suivantes sont les mêmes que celles qui s'exécutent au niveau de l'hôte :

```
cl_double clrngRandomU01(clrngStream* stream);

clrngStatus clrngRandomU01Array(clrngStream* stream, size_t count,
                                cl_double* buffer);

cl_int clrngRandomInteger(clrngStream* stream, cl_int i, cl_int j);

clrngStatus clrngRandomIntegerArray(clrngStream* stream, cl_int i,
                        cl_int j, size_t count, cl_int* buffer);
```

5.2 Définition de clProbDist

clProbDist est une interface de programmation développée pour s'exécuter sur les dispositifs compatibles avec OpenCL et offre plusieurs fonctionnalités relatives aux lois de probabilité comme l'évaluation de la fonction de densité ou de masse, l'évaluation de la fonction de répartition et son complément ainsi que la fonction de répartition inverse. Cette API est utilisée pour la génération de nombres aléatoires non uniformes selon la méthode d'inversion (section 2.5.1), mais peut être aussi utilisée pour le calcul des intervalles de confiance, le calcul de la valeur-`p` pour des tests statistiques ou aussi le calcul des densités pour évaluer la fonction de vraisemblance d'un échantillon, etc.

Chaque loi de probabilité est implémentée sous forme d'objets (`struct`) ayant comme composants les paramètres définissant la distribution implémentée. Il existe deux catégories de fonctions : celles qui ont le suffixe `WithObject` dans leur nom et qui manipulent l'objet de la distribution pour retourner leurs valeurs ; et celles qui n'ont pas ce dernier suffixe et qui utilisent directement les paramètres de la loi de probabilité au lieu de l'objet pour effectuer les calculs. Chaque catégorie peut être plus appropriée que l'autre selon le cas d'utilisation, comme nous le verrons dans le prochain chapitre. Cela permet aussi plus de flexibilité dans l'utilisation de l'API.

Le listing 5.4 illustre comment générer une suite de nombres aléatoires selon la loi normale en utilisant l'inversion d'une variable uniforme produite par le générateur MRG32k3a. Notons le passage de l'objet de la distribution `dist` comme paramètre au kernel afin qu'il soit utilisé par la fonction de répartition inverse.

5.2.1 Les distributions implémentées

5.2.1.1 La distribution normale

La fonction de densité (PDF) de la loi normale avec une moyenne μ et une variance σ^2 ($\sigma > 0$) a la forme suivante :

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} \exp\left[\frac{-(x-\mu)^2}{2\sigma^2}\right] \qquad \text{pour } (-\infty < x < +\infty) \text{ et } (\sigma > 0).$$

La fonction de répartition (CDF) est évaluée en utilisant l'approximation de Chebyshev avec 16 décimales de précision[34] ; l'inverse de la fonction de répartition est calculé grâce à des approximations de Chebyshev définies dans [4] avec 16 décimales de précision. L'objet de la distribution est implémenté avec les variables suivantes : `cl_double mu` et `cl_double sigma`.

```
1  #include <mrg32k3a.clh>
2  #include <normal.clh>

4  __kernel void AfficherNbrNonUniform(__global clrngMrg32k3a* stream,
5                                      __global clprobdistNormal* dist){
6  for (int i =0 ; i < 10 ; i++) {

8     cl_double u = clrngMrg32k3aRandomU01(stream);

10    cl_double x = clprobdistNormalInverseCDFWithObject(dist, u, null);

12    printf("Variable aléatoire selon la loi normale : \%5.4f" , x );
13  }
14 }
```

Listing 5.4 – Exemple simple utilisant clProbDist au niveau du kernel

5.2.1.2 La distribution Lognormale

Une variable aléatoire X suit une loi Lognormale avec des paramètres μ et σ^2 si la variable aléatoire $Y = \ln X$ suit une loi normale avec une espérance μ et une variance σ^2. Les formules qui suivent correspondent respectivement à la fonction de densité, la fonction de répartition et la fonction de répartition inverse de la distribution Lognormale :

$$f(x) = \frac{1}{\sigma x\sqrt{2\pi}} \exp\Big[\frac{-(\ln x - \mu)^2}{2\sigma^2}\Big] \qquad (0 < x < +\infty)$$
$$F(x) = \Phi\big(\frac{\ln x - \mu}{\sigma}\big) \qquad (0 < x < +\infty)$$
$$F^{-1}(u) = \exp\Big[\mu + \sigma\Phi^{-1}(u)\Big] \qquad (0 < u < 1).$$

Notons que Φ est la fonction de répartition de la loi normale Standard. La fonction de répartition et son inverse sont évalués en exploitant les implémentations de la distribution normale. L'objet de la distribution est implémenté avec les variables suivantes : `cl_double mu` et `cl_double sigma`.

5.2.1.3 La distribution Exponentielle

Il s'agit d'une loi de probabilité continue avec une espérance notée $1/\lambda$. Les formules qui suivent correspondent respectivement à la fonction de densité, la fonction de répartition et de répartition inverse :

$$f(x) = \lambda e^{-\lambda x} \qquad (x \geq 0)$$
$$F(x) = 1 - e^{-\lambda x} \qquad (x \geq 0)$$
$$F^{-1}(u) = -\ln(1-u)/\lambda \qquad (0 \leq u < 1).$$

L'API exploite directement ces formules dans l'implémentation de la loi Exponentielle. L'objet de la distribution est implémenté avec la variable suivante : `cl_double lambda`.

5.2.1.4 La Distribution Gamma

Il s'agit d'une loi de probabilité continue de variables aléatoires réelles positives, ayant respectivement $\alpha > 0$ et $\lambda > 0$ comme propriétés de forme et propriétés d'intensité.

La fonction de densité a la forme suivante :

$$f(x) = \frac{\lambda^{\alpha}}{\Gamma(\alpha)} x^{\alpha-1} e^{-\lambda x} \qquad (x > 0)$$
$$\Gamma(\alpha) = \int_0^{\infty} x^{\alpha-1} e^{-x} dx.$$

où Γ est la fonction Gamma d'Euler avec $\Gamma(n) = (n-1)!$ pour $n > 0$.

Plusieurs fonctions de cette distribution sont implémentées en utilisant une approximation avec une certaine précision d qui peut être définie par l'utilisateur. La fonction de répartition est implémentée en utilisant une version améliorée de l'approximation définie dans [3]. La fonction de répartition inverse est approximée en résolvons $F(x) - u = 0$ avec la méthode de bissection (dans le cas où $u \leq 10^{-8}$ ou $\alpha \leq 1.5$) ou avec la méthode de Brent-Dekker dans les autres cas [7, 6]. L'objet de la distribution est implémenté avec les variables suivantes : `cl_double alpha`, `cl_double lambda` et `cl_int decprec`.

5.2.1.5 La distribution de Poisson

Il s'agit d'une loi de probabilité discrète avec une moyenne $\lambda > 0$ et ayant la fonction de masse et la fonction de répartition suivantes :

$$p(x) = P(X = x) = \frac{\lambda^x}{x!} e^{-\lambda} \qquad (x > 0)$$
$$F(x) = e^{-\lambda} \sum_{j=0}^{x} \frac{\lambda^j}{j!}.$$

Détail d'implémentation de l'objet de la distribution

L'API peut précalculer et garder en mémoire (dans la structure de l'objet) des tableaux contenant les valeurs de PDF et CDF pour un paramètre λ donné sous la condition que ce dernier ne soit pas très grand ($\lambda \leq 10^5$). Cela a pour effet d'améliorer la per-

formance notamment dans le cas de la génération de nombres aléatoires non uniformes. Cette approche est implémentée dans la fonction `clprobdistPoissonCreate(..)` qui permet la création de la distribution comme décrit dans les algorithmes suivants.

En théorie, la distribution de Poisson a un support infini, mais comme il n'est pas possible de sauvegarder toutes les valeurs de la distribution dans la mémoire de l'ordinateur, notre algorithme crée deux tableaux P et F (qui contiendrons respectivement, les valeurs de la PDF et CDF) avec seulement N_{max} éléments, avec $N_{max} = \lambda + 16(2+\sqrt{\lambda}))$. Cette formule définie heuristiquement au niveau du logiciel SSJ devrait assurer que la probabilité devient extrêmement petite pour tout $i > N_{max}$.

L'algorithme 1 correspond à la fonction de création de la distribution de Poisson qui permet le calcul des valeurs des densités de probabilité $p_i = P[X = i]$. Nous construisons d'abord le tableau P de taille N_{max} qui contient les p_i qui sont non négligeables, cela en commençant le remplissage du tableau P à partir de l'indice $mid = \lfloor \lambda \rfloor$ qui est aussi le mode de la distribution. Nous exécutons ensuite une boucle qui calcule à chaque itération le terme $p_{i-1} = (p_i \times i/\lambda)$, puis nous décrémentons la valeur de i. La somme des termes est maintenue dans la variable *sum* qui sera utilisée à la ligne 19 pour trouver la valeur de la densité de probabilité. La boucle se termine lorsqu'on atteint une itération ayant la valeur du terme inférieur à $\varepsilon = 10^{-22}/p_\lambda$ ou que $i = 0$. À la sortie de la boucle, l'algorithme sauvegarde l'indice i dans x_{min} qui est l'indice du tableau qui contient la plus petite valeur des termes qui est supérieur à ε, soit $P_i < \varepsilon$ pour tout $(i < x_{min})$.

Algorithm 1: Calcul des probabilités p_i

```
1  int mid = ⌊λ⌋;
2  int i = mid;
3  int sum = p_i = 1.0 /* valeur de p_i normalisée à la ligne 20 */
4   double ε = 10^-22 / p_λ;
   /* Calcul de la partie inférieure à l'indice mid     */
5  while (i > 0 and p_i > ε) do
6  |   p_{i-1} = (p_i × i/λ);
7  |   i = i − 1;
8  |   sum = sum + p_i;
9  end
10 int x_min = i;
   /* Calcul de la partie supérieure à l'indice mid     */
11 i = mid;
12 while (p_i > ε) do
13 |   p_{i+1} = (p_i × λ/(i+1));
14 |   i = i + 1;
15 |   sum = sum + p_i;
16 end
17 int x_max = i;
   /* Normalise la somme des probabilités à 1           */
18  for (i = x_min → x_max) do
19 |   p_i = p_i / sum;
20 end
```

La boucle précédente permettait de calculer la partie inférieure du tableau P (i.e. $i < mid$). Pour remplir la partie supérieure, nous utilisons une seconde boucle qui commence les itérations dans le tableau P au même indice que précédemment (ligne 11) et nous calculons à chaque fois la valeur du terme $p_{i+1} = (p_i \times \lambda/(i+1))$ puis nous in-

crémentons la valeur de i. Les itérations se terminent lorsque la valeur du terme devient inférieure à un ε. À la sortie de la boucle, la fonction sauvegarde l'indice de sortie i dans la valeur du paramètre de la distribution x_{max} tel que $(P_i < \varepsilon)$ pour tout $(i > x_{max})$. Notons qu'à partir de la ligne 19, nous calculons les valeurs des probabilités p_i en divisant les éléments du tableau P (qui sont entre les indices x_{min} et x_{max} inclusivement) par la valeur de la somme des termes. Notons aussi que x_{min} et x_{max} seront utilisés dans le calcul de la taille finale des tableaux précalculés PDF et CDF (voir ligne 2 de l'algorithme 3)

Pour calculer le tableau contenant la fonction de répartition F, nous utilisons l'algorithme 2. Notons qu'à partir de la ligne 4 nous utilisons une boucle qui fait le cumule des probabilités se trouvant dans le tableau P et sauvegarde le résultat dans le tableau F, cela tant que la valeur du CDF est inférieure à 0.5. À la sortie de cette boucle, nous définissons la variable x_{med} comme étant l'indice du tableau tel que :

— $CDF[x] = F(x)$ pour tout $(x \leq x_{med})$

— $CDF[x] = \bar{F}(x)$ pour tout $(x > x_{med})$.

Ainsi, le tableau CDF contient dans sa partie supérieure (à partir de la case x_{med}) les valeurs de la fonction de répartition complémentaire.

Algorithm 2: Calcul du tableau F de la distribution de Poisson

1 $F_{x_{min}} = P_{x_{min}}$;
2 $i = x_{min}$;
3 **while** *(*$i < x_{max}$ *and* $F_i < 0.5$*)* **do**
4 $\quad i = i + 1$;
5 $\quad F_i = P_i + F_{i-1}$;
6 **end**
`/* Ceci est la limite entre F et 1-F */`
7 int $x_{med} = i$;
8 $F_{x_{max}} = P_{x_{max}}$;
9 $i = x_{max} - 1$;
10 **while** $i > x_{med}$ **do**
11 $\quad F_i = P_i + F_{i+1}$;
12 $\quad i = i - 1$;
13 **end**

Enfin, la fonction `clprobdistPoissonCreate(..)` crée l'objet de la distribution en allouant assez d'espace mémoire pour contenir les paramètres de la distribution de Poisson ainsi que les tableaux PDF et CDF à partir des tableaux P et F. Notons qu'à la ligne 1 de l'algorithme 3 la taille finale des tableaux est égale à $(x_{max} + 1 - x_{min})$ au lieu de N_{max}. Cet espace sera initialisé avec les valeurs de P et F déjà précalculées. Cette fonction retournera à l'utilisateur un pointeur vers l'objet créé qui sera utilisé comme paramètre au niveau des fonctions de la distribution de Poisson.

Algorithm 3: Création et initialisation de la distribution de Poisson

1 int $len = (x_{max} + 1 - x_{min})$;
2 int $bufSize$ = sizeof(clprobdistPoisson) + 2 * ((len - 1) $\times$ sizeof(double));
3 clprobdistPoisson dist = CréerTableau($bufSize$);
`/* Copie P et F dans PDF et CDF resp. */`
4 **for** $i = 0 \rightarrow len$ **do**
5 dist->PDF_i = $P_{x_{min}+i}$;
6 dist->$CDF_{i+len-1}$ = $F_{x_{min}+i}$;
7 **end**
8 return dist;

Aspect dynamique de la taille de l'objet de la distribution

Puisque la taille des tableaux PDF et CDF change en fonction de λ et puisque OpenCL ne permet pas le changement de taille pour les structures prédéfinies dans le code (i.e. si un tableau est déclaré avec une taille donnée X, ce tableau ne doit pas changer de taille). Pour cela, nous ne pouvions pas utiliser la déclaration standard de tableau ; au lieu de cela, nous avons dû déclarer (dans l'objet de la distribution) les tableaux CDF et PDF comme ayant une taille fixe d'un seul élément (voir le listing 5.5) et en même temps nous avons réservé assez d'espace mémoire pour tous les éléments des deux tableaux. Cela est illustré à la ligne 2 de l'algorithme 3. Nous avons ensuite copié les tableaux P et F dans les buffers pointés par les variables PDF et CDF . De cette façon, le compilateur va toujours détecter une taille fixe d'un élément mais en réalité les tableaux peuvent avoir différentes tailles. Pour pouvoir accéder aux éléments du tableau CDF nous avons créé une fonction qui prend comme paramètre l'indice de l'élément puis fait un décalage équivalent à la taille du buffer PDF pour retourner la valeur $CDF[i + (sizeOf(PDF) - 1)]$.

```
1 struct _clprobdistPoisson {
2     clprobdistPoissonParams params; //lambda, xmin, xmed, xmax, etc.
3   cl_double pdf[1];
4   cl_double cdf[1];
5  }lprobdistPoisson;
```

Listing 5.5 – L'objet de la distribution de Poisson

Génération d'une variable aléatoire selon la loi de Poisson

Pour pouvoir retourner la variable aléatoire non uniforme, nous pouvons utiliser la fonction de répartition inverse qui exploite l'objet de la distribution avec les tableaux précalculés. Cette fonction prend en entrée une v.a. uniforme u puis effectue une recherche dichotomique dans le tableau précalculé CDF pour la plus petite valeur de x qui satisfait à $F(x) \geq u$. Cette recherche dichotomique est illustrée dans l'algorithme 4. Nous commençons par vérifier si la valeur de u est en dessous de la CDF de l'élément du tableau ayant l'indice $(x_{med} - x_{min})$, dans ce cas (et si u est aussi plus grande que le $CDF[0]$) le code entre les lignes 6 à 15 effectue la recherche dichotomique jusqu'à ce que les indices de la boucle i et j deviennent égaux. Ensuite, la valeur de $i + xmin$ est retournée comme variable aléatoire suivant la loi de Poisson. Le code à partir de la ligne 16 est similaire à la partie précédente sauf qu'on considère la recherche avec la v.a. $(1-u)$, cela pour pouvoir effectue la recherche dans la partie supérieure du tableau CDF qui contient la fonction de répartition complémentaire comme on l'a vu dans la section précédente.

Algorithm 4: Recherche dichotomique dans la fonction CDF inverse avec objet

```
1  int i = 0, j = 0, k = 0;
2  if (u ≤ CDF[dist.x_med - dist.x_min]) then
3      if (u ≤ CDF[0]) then return dist.x_min;
4      i = 0 ; j = dist.x_med - dist.x_min;
5      while (i < j) do
6          k = (i + j)/2;
7          if (u > CDF[k]) then
8              i = k + 1;
9          else
10             j = k;
11         end
12     end
13 else
14     u = 1 − u;
15     if (u < CDF[dist.x_max - dist.x_min]) then return dist.x_max;
16     i = dist.x_med − dist.x_min + 1 ; j = dist.x_max − dist.x_min;
17     while (i < j) do
18         k = (i + j)/2;
19         if (u > CDF[k]) then
20             i = k + 1;
21         else
22             j = k;
23         end
24     end
25     i = i − 1;
26 end
27 return i + dist.x_min;
```

Fonctions de la distribution de Poisson n'utilisant pas les tableaux précalculés

Dans certains cas, où l'on veut générer une seule v.a. de Poisson pour un λ donné, il n'est pas efficace de précalculer le tableau CDF dans la distribution de Poisson, au lieu de cela, on peut utiliser la fonction de répartition inverse qui utilise directement les paramètres de la distribution pour effectuer le calcul de la valeur du CDF. Nous distinguons le cas où la valeur de l'espérance de la loi n'est pas trop grande ($\lambda < 700$) et celui où elle est plus grande. Quand λ n'est pas trop grand, on utilise une boucle qui calcule la somme des termes $\frac{\lambda^i}{i!}e^{-\lambda}$ tant qu'elle est inférieure à la probabilité u, pour finalement retourner la valeur de x qui correspond au dernier i qui est visité dans la boucle.

Si λ est grand et que la valeur de x est petite, alors $p(x)$ devient négligeable. Pour cela, une recherche binaire est d'abord exécutée pour trouver une limite inférieure $x_{min} \in 0,..,\lambda$ qui a une probabilité non négligeable. Ensuite, le calcul séquentiel démarrera à partir de $x = x_{min}$.

Quant à la fonction de répartition et son complément qui utilisent directement le paramètre λ, ils ont été implémentés avec les formules définies au début de la section 5.2.1.5, cela pour le cas où $(\lambda \leq 200)$. Mais si la valeur de l'espérance est plus grande, alors nous avons exploité la relation $F(X) = 1 - G_{x+1}(\lambda)$ tel que G_{x+1} est la fonction de répartition de la loi Gamma ayant comme paramètre $(\alpha, \lambda) = (x+1, 1)$.

5.2.2 L'interface de programmation clProbDist

Cette section définit l'interface de clProbDist sous une forme générique avec des noms de fonctions et de structures de données ayant comme préfixe `clprobdist`. En réalité - au niveau de l'implémentation - ces noms contiendraient aussi le type de distribution implémentée ; par exemple la fonction de répartition de la distribution Gamma sera notée `clprobdistGammaCDF(..)`.

Structure de données

```
typedef struct {
      int decprec;
      cl_double supportA, supportB;
      .... /* params de la distribution */
} clprobdistOBJET;
```

— Cet objet représente la distribution de probabilité implémentée et contient des propriétés communes à toutes les distributions (le support de la variable x et la précision des valeurs retournées) ainsi que les paramètres de la loi de probabilité. Par exemple, `clprobdistNormal` est l'objet de la distribution normale qui a `mu` et `sigma` comme paramètres, et `clprobdistPoisson` est l'objet de la distribution de Poisson :

```
typedef enum { /*.. */ } clprobdistStatus;
```

— La plupart des fonctions retournent un pointeur vers un objet qui indique le succès ou l'erreur de l'exécution et ayant comme type `clprobdistStatus`.

Fonction de création de l'objet

Les fonctions suivantes sont disponibles seulement au niveau de l'hôte :

```
clprobdistOBJET* clprobdistCreate(... /* params de la distribution */,
                                  size_t* bufSize, clprobdistStatus* err);
```

— Cette fonction permet de créer l'objet de la distribution et d'initialiser ses paramètres (les tableaux PDF et CDF) puis de retourner un pointeur vers cet objet ; *bufsize* contiendra la taille de l'espace mémoire alloué. Il faut noter que dans le cas où la taille des tableaux est grande, il faut copier l'objet de la distribution en mémoire globale ; dans le cas contraire, l'utilisateur pourra exploiter le préprocesseur `CLPROBDIST_POISSON_IN_LOCAL_MEMORY` pour copier la distribution dans la mémoire locale du CU.

```
clprobdistStatus clprobdistDestroy(clprobdistOBJET* dist);
```

— Cette fonction libère les ressources allouées par la fonction précédente.

Fonctions statistiques utilisant l'objet de distribution

Les fonctions suivantes sont disponibles au niveau de l'hôte et du dispositif et exploitent les paramètres de la distribution inclus dans l'objet *dist* pour faire leurs calculs :

```
cl_double clprobdistDensityWithObject(const clprobdistOBJET* dist,
                                      cl_double x, clprobdistStatus* err);
```

— Cette fonction retourne la densité de x suivant la loi de probabilité implémentée et avec les paramètres définis dans l'objet *dist*.

```
cl_double clprobdistCDFWithObject(const clprobdistOBJET* dist,
                                  cl_double x, clprobdistStatus* err);
```

— Retourne la valeur de la fonction de répartition (CDF) de x.

```
cl_double clprobdistComplCDFWithObject(const clprobdistOBJET* dist,
                                       cl_double x, clprobdistStatus* err);
```

— Retourne la valeur de la fonction de répartition complémentaire de x.

```
cl_double clprobdistInverseCDFWithObject(const clprobdistOBJET* dist,
                                         cl_double u, clprobdistStatus* err);
```

— Retourne la valeur de la fonction de répartition inverse de x.

```
cl_double clprobdistMeanWithObject(const clprobdistOBJETl* dist,
                                   clprobdistStatus* err);
```

— Retourne la valeur de l'espérance à partir du paramètre défini dans l'objet *dist*.

```
cl_double clprobdistVarianceWithObject(const clprobdistOBJET* dist,
                                       clprobdistStatus* err);
```

— Retourne la valeur de la variance à partir du paramètre défini dans l'objet *dist*.

```
cl_double clprobdistStdDeviationWithObject(const clprobdistOBJET* dist,
                                           clprobdistStatus* err);
```

— Retourne la valeur de l'écart type à partir du paramètre défini dans l'objet *dist*.

```
cl_double clprobdistGet[PARAM](const clprobdistOBJET* dist,
                                    clprobdistStatus* err);
```

— Retourne la valeur du paramètre noté entre crochets et inclus dans l'objet *dist* ; par exemple, la fonction `clprobdistNormalGetSigma(dist, err)` retourne l'écart type de la loi normale.

Fonctions statistiques n'utilisant pas l'objet de la distribution

Les fonctions suivantes effectuent les mêmes tâches que les fonctions citées précédemment qui ont le même nom mais qui possèdent le suffixe `withObject`. Notons le passage des paramètres de la distribution dans les signatures des fonctions. Au niveau du code, les fonctions de la section précédente (qui exploite l'objet `dist`) font appel aux fonctions suivantes pour effectuer leurs calculs :

```
cl_double clprobdistDensity(... /* params de la distribution */,
                            cl_double x, clprobdistStatus* err);

cl_double clprobdistCDF(... /* params de la distribution */,
                        cl_double x, clprobdistStatus* err);

cl_double clprobdistComplCDF(... /* params de la distribution */,
                             cl_double x, clprobdistStatus* err);

cl_double clprobdistInverseCDF(... /* params de la distribution */,
                               cl_double u, clprobdistStatus* err);

cl_double clprobdistMean(... /* params de la distribution */,
                                    clprobdistStatus* err);

cl_double clprobdistVariance(... /* params de la distribution */,
                                        clprobdistStatus* err);

cl_double clprobdistStdDeviation(... /* params de la distribution */,
                                            clprobdistStatus* err);
```

CHAPITRE 6

SIMULATIONS MONTE-CARLO

Ce chapitre présente la simulation d'un modèle d'inventaire simplifié qui utilise dans un premier temps clRNG pour générer des variables aléatoires uniformes afin de représenter les aspects stochastiques du modèle puis, dans un second temps, qui utilise clProbdist pour générer les mêmes variables aléatoires mais selon une distribution non uniforme. Par la suite, nous allons décrire un deuxième modèle de simulation d'une option financière de type asiatique (Asian call option). L'objectif recherché n'est pas d'étudier le système d'inventaire ou l'option financière en soi, mais de montrer comment bien utiliser les APIs développées et d'expérimenter différentes configurations des work items avec les générateurs de nombres aléatoires ainsi qu'évaluer la performance des générateurs implémentés.

6.1 Simulations avec clRNG

6.1.1 Description du modèle

Le modèle d'inventaire utilisé est une version modifiée de l'exemple décrit dans la librairie SSJ [21] avec la différence que les exécutions de la simulation se feront en parallèle au niveau d'un GPU puis sur un APU. Nous avons aussi implémenté la même simulation au niveau de l'hôte pour pouvoir comparer le gain de performance. Conceptuellement, les streams seront créés au niveau de l'hôte puis transférés vers le dispositif pour générer les nombres aléatoires uniformes au niveau des PE, où nous pourrons manipuler les streams et sous-streams en parallèle avec la technique des nombres aléatoires communs et indépendants.

Dans le modèle d'inventaire, à chaque jour j nous commençons avec un stock X_j ; durant la journée, nous recevrons une demande D_j et à la fin de la journée, nous aurons vendu $\min(D_j, X_j)$ avec un nombre de ventes perdues égal à $\max(0, D_j - X_j)$. Les de-

mandes effectuées sur des journées successives seront modélisées par des variables aléatoires indépendantes et uniformément distribuées sur l'ensemble `{0,1,..,L}`. Chaque produit vendu générera un revenu c, les produits restant en stock $Y_j = max(0, X_j - D_j)$ auront chacun un coût h par jour. L'inventaire est contrôlé avec une politique (s,S) de sorte qu'en fin de journée, si $Y_j < s$, alors le stock doit être alimenté de $S - Y_j$ unités. La commande d'approvisionnement faite le soir a une probabilité p d'arriver durant la même nuit et a une probabilité $(1-p)$ qu'elle ne soit jamais expédiée. Dans le premier cas, il y aura un coût fixe K et un coût marginal de valeur k par unité reçue.

Nous voulons simuler le modèle avec un stock initial $X_0 = S$; pour une politique (s,S) et durant m journées consécutives. Nous répliquerons cela n fois de façon indépendante pour estimer la valeur de l'espérance du profit par jour et nous évaluerons la performance de différentes configurations de streams et de sous-streams. Cela permettera aussi de comparer différentes politiques (s_i, S_i) en faisant répéter la simulation autant de fois que l'on a de politiques afin d'estimer le profit espéré par jour pour chaque politique et tenter d'identifier la politique optimale.

6.1.2 Description de l'implémentation

Pour bien simuler ce modèle n fois, nous avons a dû penser à la meilleurs stratégie pour utiliser les streams dans le contexte parallèle en gardant à l'esprit qu'il faut aussi pouvoir comparer plusieurs politiques de façon efficace. Nous pouvons utiliser la technique des nombres aléatoires communs qui a l'avantage (si elle est bien implémentée) de réduire la variance du résultat des comparaisons. Cela peut se faire en utilisant deux streams distincts, l'un pour générer les valeurs aléatoires correspondantes aux demandes et l'autre pour générer la probabilité de la réception des commandes. Une autre stratégie serait d'utiliser un seul stream pour tout le processus mais cela aurait produit une plus grande variance [25] et il aurait fallu un plus grand nombre de simulations pour bien estimer la moyenne du profit par jour. À titre d'exemple voir figure 6.2.

Le listing 6.1 montre le code qui implémente une seule exécution du modèle.

À noter que nous exploitons l'objet `stream_demand` pour générer les variables aléatoires en nombre entier pour les demandes et que nous utilisons `stream_order` pour avoir la probabilité de réception des commandes. Typiquement, chaque exécution utilisera soit un sous-stream de chacun des deux streams précédents, soit une nouvelle paire de streams à chaque exécution (ce dernier cas nécessitera $2n$ streams). Ces deux configurations peuvent assurer qu'à chaque fois, les streams commencent au même endroit pour n'importe quelle politique. Cependant l'utilisation des sous-streams a clairement l'avantage d'utiliser moins de ressources en mémoire au niveau du dispositif ; cela est intéressant pour la simulation de modèles plus complexes.

Durant les simulations, nous avons fixé $L = 100$, $c = 2$, $h = 0.1$, $K = 10$, $k = 1$ et $p = 0.95$ et nous avons expérimenté avec différentes valeurs de n, m et (s,S) et avec différentes façons d'utiliser les streams. Le déroulement se fait comme suit : nous choisissons dans un premier temps (n,m,s,S) pour exécuter n fois la fonction `inventorySimulateOneRun` (donnée au listing 6.1), ce qui donne les moyennes des profits $P_1, P_2, ..., P_n$ qui seront utilisées dans le calcul de la valeur moyenne $\overline{P}_n$, la variance empirique S_n^2 et un intervalle de confiance à 95% pour l'espérance du profit, en utilisant une approximation normale. Cet intervalle est donné par la formule $(\overline{P}_n \pm 1.96 S_n/\sqrt{n})$.

Ensuite, dans un deuxième temps, nous nous intéresserons à comparer p politiques distincts $\{(s_i, S_i), i = 0, ..., p-1\}$ pour estimer l'espérance de la moyenne des profits en utilisant CRN et IRN à travers toutes les politiques ; plus particulièrement, nous estimerons la différence entre les espérances de la moyenne des profits des deux politiques. Pour implémenter cela, nous fixerons les valeurs de n et m et exécuterons la simulation des p politiques en série et en parallèle, comme nous le verrons dans la prochaine section.

```
double inventorySimulateOneRun(int m, int s, int S,
                               clrngMrg31k3pStream* stream_demand,
                               clrngMrg31k3pStream* stream_order)
{
int Xj = S, Yj;  //Stock de début et de fin de journée
double profit = 0.0;
for (int j = 0; j < m; j++) {
  // Génère et soustrait la demande du jour
  Yj = Xj - clrngMrg31k3pRandomInteger(stream_demand, 0, L);
  if (Yj < 0)
    Yj = 0; //Demande perdue.
  profit += c * (Xj - Yj) - h * Yj;
  if ((Yj < s) && (clrngMrg31k3pRandomU01(stream_order) < p)){
      //La commande est reçue
      profit -= K + k * (S - Yj);
      Xj = S;
      }
  else
      Xj = Yj;
  }

return profit / m;  // le profit moyen sur m jours.
}
```

Listing 6.1 – Simulation d'une exécution du modèle d'inventaire

6.1.3 Simulations au niveau du CPU

Il existe plusieurs façons d'exécuter les n réplications indépendantes du modèle d'inventaire sur le CPU ; cela est illustré dans le code du listing 6.2 :

1. Nous pouvons utiliser le premier sous-stream d'un seul stream pour générer tous les nombres aléatoires nécessaires à n simulations ; cela est implémenté avec la fonction `inventorySimulateRunsOneStream`.
2. Une deuxième approche est d'utiliser deux streams distincts `stream_demand` et `stream_order` pour générer chaque type de variable aléatoire. À la première exécution, nous utiliserons les deux premiers sous-streams de chaque stream, ensuite, nous faisons avancer les deux streams vers leurs prochains sous-streams respectifs ; ces derniers seront utilisés dans la deuxième exécution et ainsi de suite. Cela est illustré par la fonction `inventorySimulateRunsSubstreams`.
3. Nous pouvons aussi choisir d'utiliser une nouvelle paire de streams à chaque exécution. Nous utiliserons seulement le premier sous-stream de chaque stream ; pour cela il, faut clairement deux tableaux contenant $2n$ streams distincts au total (voir la fonction `inventorySimulateRunsManyStreams`).

Notons que la deuxième approche est typiquement utilisée dans les simulations en nombres aléatoires communs (CRN) ; la troisième approche utilisant n streams pour les demandes et n streams pour les commandes n'est pas adaptée pour la simulation avec des grands n. Nous trouverons dans la figure 6.3 le code utilisé pour créer les tableaux des streams.

```
1  void inventorySimulateRunsOneStream (int m, int s, int S, int n,
2              clrngMrg31k3pStream *stream, double *stat_profit) {
3  //Simulation du modèle d'inventaire sur m jours avec la politique
4  //(s,S) en utilisant un seul stream avec le même sous-stream pour
5  //toutes les n exécutions.
6  for (int i = 0; i < n; i++)
7    stat_profit[i] = inventorySimulateOneRun (m, s, S, stream, stream);
8  }
9  //_____________________________________________________________________
10 void inventorySimulateRunsSubstreams (int m, int s, int S, int n,
11 clrngMrg31k3pStream *stream_demand, clrngMrg31k3pStream *stream_order
12                                              ,double *stat_profit) {
13 //Similaire à inventorySimulateRuns mais en utilisant deux streams
14 //avec leurs sous-streams.
15  for (int i = 0; i < n; i++) {
16     stat_profit[i] = inventorySimulateOneRun (m, s, S, stream_demand,
17                                                      stream_order);
18     clrngMrg31k3pForwardToNextSubstreams(1, stream_demand);
19     clrngMrg31k3pForwardToNextSubstreams(1, stream_order);
20     }
21 }
22 //_____________________________________________________________________
23 void inventorySimulateRunsManyStreams (int m, int s, int S, int n,
24 clrngMrg31k3pStream *streamsDemand, clrngMrg31k3pStream *streamsOrder,
25                                              double *stat_profit) {
26 //Similaire à inventorySimulateRuns mais avec deux tableaux de
27 //n streams chacun et en utilisant une nouvelle paire de streams
28 //à chaque exécution.
29 for (int i = 0; i < n; i++)
30 stat_profit[i] = inventorySimulateOneRun(m, s, S, &streamsDemand[i],
31                                                  &streamsOrder[i]);
32 }
```

Listing 6.2 – Simulation de n exécutions sur le CPU avec trois approches

```
....
//Créer les tableaux de streams et de profit
clrngMrg31k3pStream *streams_demand = clrngMrg31k3pCreateStreams(NULL,
                                                    n, NULL, NULL);
clrngMrg31k3pStream *streams_order = clrngMrg31k3pCreateStreams(NULL,
                                                    n, NULL, NULL);
double *stat_profit = (double *) malloc (n * sizeof (double));

//Lancer la simulation
inventorySimulateRunsManyStreams (m, s, S, n, streams_demand,
                                  streams_order, stat_profit);
//Calculer l'intervalle de confiance
computeCI(n, stat_profit);
....
```

Listing 6.3 – Simulation sur le CPU avec deux tableaux de streams

6.1.4 Simulations au niveau du GPU

6.1.4.1 Simulation de n exécutions indépendantes avec n work items

Pour effectuer les n simulations sur le GPU et bénéficier du mode d'exécution parallèle et avoir de meilleures performances, nous avons utilisé le kernel illustré à la figure 6.4 qui sera exécuté par chacun des n work items simultanément. La figure 6.5 montre le code qui s'exécute au niveau de l'hôte pour préparer et lancer la simulation sur le GPU selon la deuxième et la troisième approche de la section précédente, soit : (a) simulation avec les n sous-streams pour chacun des deux streams et (b) utilisation de $2n$ streams distincts.

Cas (a) : pour implémenter cette approche dans le dispositif, il faut construire deux tableaux contenant chacun n sous-streams successifs à partir de chacun des deux streams (cela en utilisant la fonction `clrngMrg31k3pMakeSubstreams`). Ensuite, les tableaux seront transférés vers la mémoire globale du GPU et chaque work item aura à sa disposition deux sous-streams pour générer les nombres aléatoires. Un sous-stream dans ce cas peut être considéré comme un stream personnel du work item. Cela est illustré par la figure 6.5. Notons que puisque le work item modifie uniquement l'état courant du stream, il est alors nécessaire de maintenir seulement cet état-là dans la mémoire privée du PE.

Une autre façon pourrait être de copier uniquement les deux streams en mémoire globale (au lieu de copier les $2n$ sous-streams) ; dans ce cas, chaque work item devrait appeler la fonction `clrngMrg31k3pForwardToNextSubstreams` autant de fois que la valeur de son index global (`gid`) afin de trouver le bon sous-stream qui lui correspond, ce qui est clairement inefficace et va créera de la divergence (exécution séquentielle) au niveau des CUs parce que chaque work item aurait exécuté un chemin différent pour obtenir son sous-stream.

Cas (b) : Au lieu d'utiliser $2n$ sous-streams, nous pouvons construire deux tableaux

contenant chacun n streams distincts, donc un seul stream sera utilisé par work item et par exécution.

```
1  //Chaque work item exécute le code suivant
2  __kernel void inventorySimulateGPU (int m, int s, int S,
3                          __global clrngMrg31k3pStreams *streams_demand,
4                           __global clrngMrg31k3pStreams *streams_order,
5                                      __global double *stat_profits) {

7  int gid = get_global_id (0); // Index du work item.
8  // Copier les streams en mémoire privée.
9  clrngMrg31k3pStreams stream_demand_d, stream_order_d;
10 clrngMrg31k3pCopyOverStreamsFromGlobal(1, &stream_demand_d,
11                                         &streams_demand[gid]);
12 clrngMrg31k3pCopyOverStreamsFromGlobal(1, &stream_order_d,
13                                         &streams_order[gid]);
14 // Effectuer la simulation
15 stat_profits[gid] = inventorySimulateOneRun(m, s, S,&stream_demand_d,
16                                                    &stream_order_d);
17 }
```

Listing 6.4 – Le kernel exécuté par chaque work item pour calculer le profit moyen

```
2  // Cette fonction (dont nous omettons les détails ici parce qu'ils
3  // ne sont pas très intéressants) effectue n exécutions en parallèle
4  // sur le GPU avec deux tableaux de n streams, puis sauvegarde
5  // la valeur du profit moyen.
6  void inventorySimulateRunsGPU (int m, int s, int S, int n,
7                                 clrngMrg31k3pStreams *streams_demand,
8            clrngMrg31k3pStreams *streams_order,double *stat_profit){
9    //Créer les structures OpenCL nécessaires : (context, program, etc.)
10     ...
11   //Lancer le kernel inventorySimulateGPU au niveau du GPU.
12     ...
13 }

15 //Cas (a) : Simulation de n exécutions avec n work items en utilisant
16 //deux streams et leurs sous-streams.
17 clrngMrg31k3pStream* stream_demand = clrngMrg31k3pCreateStreams(NULL,
18                                                       1, NULL, NULL);
19 clrngMrg31k3pStream* stream_order = clrngMrg31k3pCreateStreams(NULL,
20                                                     1, NULL, NULL);
21 clrngMrg31k3pStream* substreams_demand = clrngMrg31k3pMakeSubstreams(
22                                         stream_demand, n, NULL, NULL);
23 clrngMrg31k3pStream* substreams_order = clrngMrg31k3pMakeSubstreams(
24                                       stream_order, n, NULL, NULL);
25 double *stat_profit = (double *) malloc (n * sizeof (double));
26 inventorySimulateRunsGPU (m, s, S, n, substreams_demand,
27                                        substreams_order, stat_profit);
28 ...

30 //Cas (b) : Simulation de n exécutions avec n work items en utilisant
31 // n streams distincts
32 clrngMrg31k3pStream* streams_demand = clrngMrg31k3pCreateStreams(
33                                                   NULL,n, NULL, NULL);
34 clrngMrg31k3pStream* streams_order = clrngMrg31k3pCreateStreams(
35                                                 NULL, n, NULL, NULL);
36 double *stat_profit = (double *) malloc (n * sizeof (double));
37 inventorySimulateRunsGPU (m, s, S, n, streams_demand, streams_order,
38                                                       stat_profit);
```

Listing 6.5 – Simulation de n exécutions sur le GPU (a) avec deux streams et (b) avec $2n$ streams

6.1.4.2 Simulation de n exécutions indépendantes avec $n_1 \ll n$ work items

Cas (c) : Si nous voulons faire la simulation avec un très grand nombre d'exécutions n (par exemple de l'ordre du million), il est plus intéressant dans le cas des dispositifs ayant une mémoire globale restreinte d'utiliser n_1 work items effectuant chacun n_2 exécutions tels que $n = n_1 n_2$. La figure 6.6 illustre comment cela est implémenté. Nous avons créé deux tableaux contenant chacun n_1 streams que nous transferons ensuite vers la mémoire globale du GPU. Chaque work item effectue n_2 exécutions et à chaque itération, il utilise un nouveau sous-stream de son stream qui lui est affecté (ce qui fait un total de n_2 sous-streams). Cela est illustré par le code du kernel `inventorySimulSubstreamsGPU`. Nous pensons que les sauts n_2 fois vers les prochains sous-streams au niveau de chaque work item risquent d'être plus rapides que de créer les n_2 sous-streams sur l'hôte puis de transférer le tout vers la mémoire globale du GPU. Cependant, le gain en performance peut être non significatif si l'on utilise un APU vu que ce dernier est localisé dans le même circuit intégré que le CPU. Notons que les résultats des n_1 exécutions parallèles sont enregistrés de façon successive dans le tableau de sortie `stat_profit` afin d'optimiser l'accès à la mémoire (voir section 4.4).

Cas (d) : Cas similaire à (b) mais utilise n_1 work items pour faire le calcul. Nous pouvons créer $2n$ streams distincts puis utiliser $2n_2$ streams au niveau de chaque work item au lieu d'utiliser des sous-streams et les streams pourront être créés comme dans le cas (b). Chaque work item sélectionnera un stream différent à chacune des n_2 itérations. Il est clair que cette façon nécessite l'utilisation de plus de mémoire globale comparativement au cas précédent et cela risque aussi d'être moins rapide que dans le cas (c) quand n_2 est grand.

Cas (e) : Nous pouvons obtenir le même résultat que dans le cas (a) mais avec n_1 work items utilisant deux streams avec n sous-streams à partir de chaque stream, c.-à-d. que nous exploitons un sous-stream à chacune des n_2 exécutions au niveau du work

item. Cela peut se faire en créant deux tableaux contenant les *n* sous-streams puis en transférant le tout vers la mémoire globale ; après cela, le work item sélectionnera le bon sous-stream en se basant sur son index global (`gid`). Cette façon requiert aussi beaucoup de mémoire comme dans le cas (d).

Cas (f) : Au lieu d'utiliser un stream par type de variable aléatoire et changer de sous-stream à chaque exécution, nous pouvons considérer l'approche inverse, c.-à-d. utiliser un sous-stream pour chaque type de nombres aléatoires et un seul stream pour chaque exécution. La figure 6.7 montre un kernel qui implémente cela. L'avantage de cette façon apparaît plus clairement dans un modèle de simulation ayant plusieurs variables aléatoires et pas simplement deux variables comme dans le cas de l'inventaire. Cependant, cela nécessite la génération des sous-streams au niveau du work item à chaque exécution.

```
#define CLRNG_ENABLE_SUBSTREAMS
#include <mrg31k3p.clh>

// Chacun des n1 work items exécute ce kernel n2 fois
__kernel void inventorySimulSubstreamsGPU(int m, int s, int S, int n2,
                         __global clrngMrg31k3pStreams *streams_demand,
                          __global clrngMrg31k3pStreams *streams_order,
                                              __global double *stat_profit){
int gid = get_global_id (0); // Index du work item.
int n1 = get_global_size (0); // Nombre total des work items.
// Fait des copies des streams en mémoire privée
clrngMrg31k3pStreams stream_demand_d, stream_order_d;
clrngMrg31k3pCopyOverStreamsFromGlobal(1, &stream_demand_d,
                                        &streams_demand[gid]);
clrngMrg31k3pCopyOverStreamsFromGlobal(1, &stream_order_d,
                                         &streams_order[gid]);
for (int i = 0; i < n2; i++){
  stat_profit[i * n1 + gid] = inventorySimulateOneRun (m, s, S,
                                 &stream_demand_d, &stream_order_d);

  clrngMrg31k3pForwardToNextSubstreams(1, &stream_demand_d);
  clrngMrg31k3pForwardToNextSubstreams(1, &stream_order_d);
  }
}
// Cette fonction effectue n = n1 * n2 exécutions sur le GPU
// avec 02 tableaux de n1 streams puis sauvegarde le profit quotidien
void inventorySimulateRunsSubstreamsGPU (int m, int s, int S, int n1,
                          int n2,clrngMrg31k3pStreams *streams_demand,
          clrngMrg31k3pStreams *streams_order, double *stat_profit){
// Créer les structures OpenCL : context, program, queue, etc.
...
// Lancer le kernel inventorySimulateGPU sur le GPU.
...
}
// (c) : Simule n = n1 * n2 exécutions avec n1 work items
double *stat_profit = (double *) malloc (n * sizeof (double));
clrngMrg31k3pStream* streams_demand = clrngMrg31k3pCreateStreams(NULL,
                                                        n1, NULL, NULL);
clrngMrg31k3pStream* streams_order = clrngMrg31k3pCreateStreams(NULL,
                                                        n1, NULL, NULL);
inventorySimulateRunsSubstreamsGPU (m, s, S, n1, n2, streams_demand,
                                          streams_order, stat_profit);
computeCI (n, stat_profit); //Calcule l'intervalle de confiance
...
```

Listing 6.6 – Simulation avec n_1 work items utilisant chacun n_2 sous-streams

```
1  __kernel void inventorySimulateGPU (int m, int s, int S,
2                              __global clrngMrg31k3pStreams *streams,
3                                       __global double *statProfits){

5  int gid = get_global_id (0); // Index du work item.
6  // Copie d'un seul stream en mémoire privée
7  clrngMrg31k3pStreams stream_demand_d, stream_order_d;

9  clrngMrg31k3pCopyOverStreamsFromGlobal(1, &stream_demand_d,
10                                                  &streams[gid]);
11 clrngMrg31k3pCopyOverStreamsFromGlobal(1, &stream_order_d,
12                                                  &streams[gid]);
13 //Générer le sous-stream de la 2e variable aléatoire
14 clrngMrg31k3pForwardToNextSubstreams(1, &stream_order_d);
15 //Effectuer la simulation
16 statProfits[gid] = inventorySimulateOneRun(m, s, S, &stream_demand_d,
17                                                    &stream_order_d);
18 }
```

Listing 6.7 – Un kernel avec inversion du rôle du stream et sous-stream

6.1.5 Résultats de l'exécution

Cette section présente les résultats des expérimentations de la simulation pour les cas (a) à (d) avec $n = 2^{22}$, $m = 10$ et $m = 100$ et pour trois choix de n_1. Les données correspondent au temps d'exécution de la simulation sur deux GPUs différents et sur un CPU. Une comparaison est aussi faite entre le facteur d'accélération de chaque cas et la configuration de base qui correspond à l'exécution avec un seul stream sur le CPU. Le premier GPU utilisé est un dispositif de calcul graphique incorporé dans un APU de type *AMD Radeon R7 graphics* avec une fréquence d'horloge de 720 Mhz, 2Go de mémoire globale, 8 unités de calcul (CU) et un total de 512 PE. Le second GPU est une carte graphique discrète de type *AMD Radeon HD 7900* avec une fréquence d'horloge de 925Mhz, 3Go de mémoire globale, 32 unités de calcul et 2048 PE. Ces deux dispositifs seront nommés ci-après GPU-A et GPU-D.

Le tableau 6.I donne les résultats statistiques (moyenne, variance et intervalle de confiance) pour la première approche sur le CPU ; les résultats des autres approches sont approximativement les mêmes. Le tableau 6.II illustre pour chaque cas que nous avons implémenté la performance en secondes et le facteur d'accélération qui est défini comme le temps d'exécution sur le CPU divisé par le temps d'exécution de l'approche considérée. Tous les temps correspondent à la moyenne de trois essais.

Les résultats montrent que la meilleure performance est au niveau des approches (c) et (d) autour de $n_1 = 2^{16}$ et $n_1 = 2^{18}$. Le cas (c) offre le plus grand gain en performances sur le dispositif GPU-D, ce qui confirme les attentes formulées dans la description des approches. Notons que plus on augmente le nombre de jours m dans la simulation, plus le facteur d'accélération augmente. Notons aussi que le meilleur choix de n_1 est plus petit sur le GPU discret que sur le GPU-A, cela parce que ce dernier possède 4 fois moins de CU et a moins de mémoire vive et de vitesse d'horloge que le GPU-D, ce qui se reflète naturellement au niveau de la performance des deux dispositifs.

Tableau 6.I – Résultats pour n simulations et une politique sur le CPU

m	n	Moyenne $\bar{P}_n$	variance S_n^2	I.C 95%
100	2^{22}	36.583	9.0392	[36.5808, 36.5865]

Tableau 6.II – Temps d'exécution en secondes avec le facteur d'accélération pour les approches (1) à (3) sur le CPU et les approches (a) à (d) sur le GPU-A et le GPU-D.

	Temps d'exécution en secondes (facteur d'accélération)				
	n_1	$(m,n) = (10, 2^{22})$		$(m,n) = (100, 2^{22})$	
CPU-hôte (1)		4.32	(1)	43.24	(1)
CPU-hôte (2)		7.21	(0.6)	46.85	(0.9)
CPU-hôte (3)		4.43	(1.0)	44.25	(1.0)
GPU-A (a)		0.370	(12)	0.682	(63)
GPU-A (b)		0.287	(15)	0.638	(68)
GPU-A (c)	2^{14}	0.177	(24)	0.709	(61)
GPU-A (c)	2^{16}	0.170	(25)	0.674	(64)
GPU-A (c)	2^{18}	0.176	(25)	0.677	(64)
GPU-A (d)	2^{14}	0.243	(18)	0.593	(73)
GPU-A (d)	2^{16}	0.227	(19)	0.549	(79)
GPU-A (d)	2^{18}	0.231	(19)	0.524	(82)
GPU-D (a)		0.219	(20)	0.259	(167)
GPU-D (b)		0.217	(20)	0.269	(161)
GPU-D (c)	2^{14}	0.041	(103)	0.095	(456)
GPU-D (c)	2^{16}	**0.039**	**(110)**	**0.086**	**(501)**
GPU-D (c)	2^{18}	0.046	(93)	0.093	(466)
GPU-D (d)	2^{14}	0.230	(19)	0.339	(128)
GPU-D (d)	2^{16}	0.219	(20)	0.282	(153)
GPU-D (d)	2^{18}	0.275	(16)	0.268	(161)

6.1.6 Comparaison de plusieurs politiques d'inventaire

Dans cette section, nous nous intéressons à évaluer comment l'API peut être utilisée pour comparer plusieurs politiques du modèle d'inventaire en utilisant les nombres aléatoires communs et indépendants.

6.1.6.1 Simulation en série

Supposons que l'on veuille simuler plusieurs politiques (s_k, S_k) pour $k = 0, .., p-1$ sur un dispositif de calcul avec la méthode CRN, une façon de faire serait de répéter de façon séquentielle p fois la simulation sur le GPU, cela en utilisant n exécutions, n_1 work items et n_2 itérations par work item et en changeant à chaque fois la politique. Le listing 6.8 réutilise la fonction `inventorySimulateRunsSubstreamsGPU` du listing 6.6 pour faire la simulation et utilise `clrngMrg31k3pRewindStreams` pour réinitialiser les streams à leurs états initiaux après chaque simulation de politique. Cela est combiné avec la fonction `clrngMrg31k3pResetNextSubstreams` utilisée dans le kernel `inventorySimulSubstreamsGPU` du listing 6.6 pour assurer qu'à chaque simulation de politique, les mêmes nombres aléatoires seront utilisés exactement aux mêmes endroits à travers les n exécutions.

À noter que l'instruction `clrngMrg31k3pRewindStreams` peut ne pas être utilisée dans le contexte d'exécution du GPU parce que chaque work item modifie seulement une *copie* de son stream qui est restée inchangée au niveau de l'hôte. Par contre, cette redondance est utile quand on veut exécuter la simulation au niveau du CPU ; dans ce cas, `clrngMrg31k3pRewindStreams` permettra effectivement de réinitialiser les streams.

Comparaison de deux politiques : CRN vs IRN

Le code du listing 6.8 m'a permis d'estimer l'espérance de la différence des moyennes des profits quotidiens $\bar{P}_n$ avec les paramètres suivants : $(s_0, S_0) = (80, 198)$ et $(s_1, S_1) = (80, 200)$, $m = 100$, $n = 2^{22}$. Le résultat est $\bar{P}_n = 0.05860$, $S_n^2 = 0.190$ et l'intervalle

de confiance à 95% pour l'espérance de la différence (en utilisant une approximation normale) est $(0.05818, 0.05902)$. Pour évaluer l'avantage de l'utilisation de la technique CRN dans la comparaison des politiques, nous avons effectué une autre simulation mais avec des variables aléatoires indépendantes (IRN). Pour cela, nous avons remplacé l'instruction `clrngMrg31k3pRewindStreams` - qui permet de réinitialiser les streams - par l'instruction `clrngMrg31k3pCreateOverStreams` qui crée un nouveau stream à chaque exécution. Les résultats obtenus avec les mêmes paramètres que précédemment sont $\bar{P}_n = 0.0569$, $S_n^2 = 18.1$ avec un intervalle de confiance à 95% pour la différence qui est égale à $(0.0528, 0.0610)$. Notons que la variance dans le cas IRN est approximativement 95 fois plus grande que dans le cas du CRN, ce qui veut dire que nous avons besoin de 95 fois moins de simulation dans le cas des nombres aléatoires communs pour atteindre la même précision.

6.1.6.2 Simulation en parallèle

L'approche précédente est bien adaptée quand on a un très grand nombre de simulations ; cependant, dans le cas où n n'est pas grand, nous pourrions simuler les p politiques en parallèle comme illustré dans le listing 6.9. Nous avons utilisé $n_1 p$ work items avec n_1 streams pour effectuer n_2 simulations avec n_2 sous-streams. Le code du listing 6.8 et du listing 6.9 produit exactement le même résultat.

Le tableau de sortie `stat_profit` du listing 6.9 contient les résultats des profits quotidiens des np simulations. Ces données sont organisés dans le tableau comme suit : les premiers $n_1 p$ éléments contiennent les résultats de la première exécution de chaque work item ; ensuite les seconds $n_1 p$ éléments contiennent les résultats de la deuxième exécution de chaque work item et ainsi de suite. À l'intérieur de chaque bloc de $n_1 p$ valeurs, nous avons les premiers n_1 valeurs successives de la première politique ; les n_1 valeurs qui suivent sont pour la deuxième politique, etc. Les résultats d'une exécution pour une politique sont toujours dans des blocs de n_1 valeurs successives dans le tableau de sortie ; aussi tous les résultats de calculs en parallèle ayant le même numéro d'itération i sont dans un bloc de $n_1 p$ valeurs successives.

```
// Assumer que pour k=0,..,p-1, (s[k], S[k]) définit une
// politique k donnée.

// Simule n = n1 * n2 exécutions sur n1 work items utilisant n2
// sous-streams, pour p politiques. On réutilise les mêmes 2n1
// streams pour toutes les politiques.

clrngMrg31k3pStream* streams_demand = clrngMrg31k3pCreateStreams(NULL
                                                    , n1, NULL, NULL);
clrngMrg31k3pStream* streams_order = clrngMrg31k3pCreateStreams(NULL
                                                  , n1, NULL, NULL);
double *stat_profit = (double *) malloc (p * n * sizeof (double));

for (int k = 0; k < p; k++) {
       inventorySimulateRunsSubstreamsGPU (m, s[k], S[k], n1, n2,
               streams_demand, streams_order, &stat_profit[k*n]);

       clrngMrg31k3pRewindStreams(n1, stream_demand);
       clrngMrg31k3pRewindStreams(n1, stream_order);
       computeCI (n, &stat_profit[k*n]);
}

// On peut comparer plusieurs politiques, pour le simple cas de k =2
double *stat_diff = (double *) malloc(n * sizeof (double));
for (int i = 0; i < n; i++)
     stat_diff[i] = stat_profit[n+i] - stat_profit[i];
computeCI (n, stat_diff);
 ...
```

Listing 6.8 – Simulation en CRN de n exécutions pour p politiques sur le dispositif avec n_1 work items et n_2 itérations par work item

Comparaison de plusieurs politiques et estimation de la fonction de coût

L'utilisation de la technique de CRN permet d'estimer la politique optimale parmi l'ensemble des (s_k, S_k). Pour cela, nous pouvons utiliser le code du listing 6.8 ou du listing 6.9 pour simuler un grand nombre de politiques appartenant au produit cartésien de l'ensemble $s = \{50, 51, .., 61\}$ par $S = \{156, 157, .., 167\}$; ceci donne 144 politiques à simuler. Nous avons estimé l'espérance du profit moyen par jour pour l'ensemble de ces politiques avec CRN puis avec IRN en remplaçant la fonction `clrngMrg31k3pRewindStreams` dans le listing 6.8 par la fonction `clrngMrg31k3pCreateOverStreams` pour obtenir $2n_1$ nouveaux streams pour chaque politique.

Les paramètres de la simulation sont $m = 100$, $n = 2^{18}$ et $n_1 = 2^{12}$ work items. Les données des figures 6.1 et 6.2 montrent que la fonction de profit change de façon beaucoup plus lisse dans le cas du CRN. Le point optimal au niveau du graphique $(s,S) = (55,159)$ est notre meilleur estimateur de la vraie valeur de la politique optimale pour le cas du CRN.

```
#define CLRNG_ENABLE_SUBSTREAMS
#include <mrg31k3p.clh>

__kernel void inventorySimulPoliciesGPU(int m, int p, int *s, int *S,
               int n2, __global clrngMrg31k3pStreams *streams_demand,
                        __global clrngMrg31k3pStreams *streams_order,
                                          __global double *stat_profit) {
// Chacun des n1*p work items exécute les n2 itérations suivantes
int gid = get_global_id(0); // Index du work item.
int n1p = get_global_size(0); // Nombre total des work items.
int n1 = n1p / p; // Nombre des streams.
int k = gid / n1; // Index de la politique que le WI va utiliser.
int j = gid % n1; // Index du stream que le work item va utiliser.

// Fait une copie des streams en mémoire privée.
clrngMrg31k3pStream stream_demand_d, stream_order_d;
clrngMrg31k3pCopyOverStreamsFromGlobal(1, &stream_demand_d,
                                       &streams_demand[j]);
clrngMrg31k3pCopyOverStreamsFromGlobal(1, &stream_order_d,
                                        &streams_order[j]);
for (int i = 0; i < n2; i++) {
 stat_profit[i * n1p + gid] = inventorySimulateOneRun(m, s[k], S[k],
                                  &stream_demand_d, &stream_order_d);
 clrngMrg31k3pForwardToNextSubstreams(1, &stream_demand_d);
 clrngMrg31k3pForwardToNextSubstreams(1, &stream_order_d);
 }
}

// Cette fonction (détails non fournis ici) effectue n = n1*n2
//exécutions sur le GPU avec deux tableaux de n1 streams
// et sauvegarde le profit quotidien
void inventorySimulateRunsPoliciesGPU (int m, int p, int *s, int *S,
               int n1, int n2, clrngMrg31k3pStreams *streams_demand,
          clrngMrg31k3pStreams *streams_order,ouble *stat_profit) {
//Création des structures qui contiennent le "context", "program",
   etc.
...
//Lancement du kernel inventorySimulPoliciesGPU au niveau du GPU.
...
}
```

Listing 6.9 – Simulation en CRN de n exécutions pour p politiques sur le dispositif avec $n_1 p$ work items et n_2 itérations par work item

	156	157	158	**159**	160	161	162	163	164	165	166	167
50	37.94537	37.94888	37.95166	37.95319	37.95274	37.95318	37.94887	37.94584	37.94361	37.94074	37.93335	37.92832
51	37.95740	37.96169	37.96379	37.96524	37.96546	37.96379	37.96293	37.95726	37.95295	37.94944	37.94536	37.93685
52	37.96725	37.97117	37.97402	37.97476	37.97492	37.97387	37.97100	37.96879	37.96184	37.95627	37.95154	37.94626
53	37.97356	37.97852	37.98098	37.98243	37.98187	37.98079	37.97848	37.97436	37.97088	37.96268	37.95589	37.94995
54	37.97593	37.98241	37.98589	37.98692	37.98703	37.98522	37.98290	37.97931	37.97397	37.96925	37.95986	37.95186
55	37.97865	37.98235	37.98740	**37.98940**	37.98909	37.98790	37.98483	37.98125	37.97641	37.96992	37.96401	37.95343
56	37.97871	37.98269	37.98494	37.98857	37.98917	37.98757	37.98507	37.98073	37.97594	37.96989	37.96227	37.95519
57	37.97414	37.98035	37.98293	37.98377	37.98603	37.98528	37.98239	37.97858	37.97299	37.96703	37.95981	37.95107
58	37.96869	37.97207	37.97825	37.97944	37.97895	37.97987	37.97776	37.97358	37.96848	37.96170	37.95461	37.94622
59	37.95772	37.96302	37.96630	37.97245	37.97234	37.97055	37.97010	37.96664	37.96122	37.95487	37.94695	37.93871
60	37.94434	37.94861	37.95371	37.95691	37.96309	37.96167	37.95860	37.95678	37.95202	37.94540	37.93785	37.92875
61	37.92200	37.93169	37.93591	37.94085	37.94401	37.95021	37.94751	37.94312	37.94000	37.93398	37.92621	37.91742

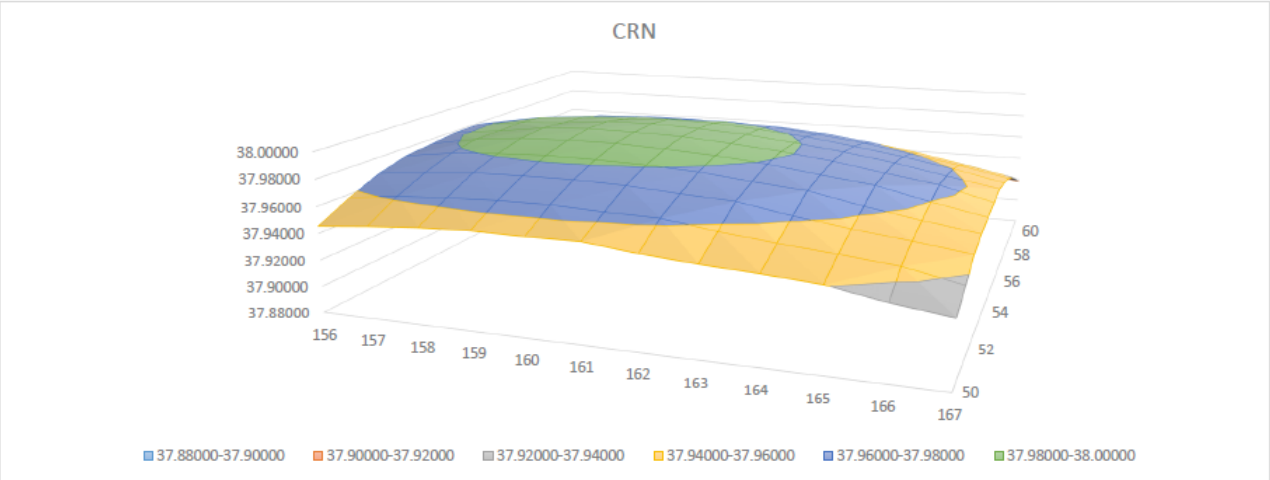

Figure 6.1 – Comparaison de 144 politiques avec CRN

	156	157	158	159	160	161	162	163	164	165	166	167
50	37.94537	37.94888	37.94736	37.95314	37.95718	37.97194	37.95955	37.95281	37.96711	37.95221	37.95325	37.92063
51	37.95740	37.96650	37.95732	37.97337	37.98137	37.94273	37.96965	37.97573	37.95425	37.96074	37.94185	37.93139
52	37.96725	37.96166	37.97192	37.99236	37.98856	37.98708	37.98266	37.94671	37.95961	37.97238	37.95982	37.94465
53	37.97356	37.96999	37.97977	37.97611	37.98929	37.99089	38.00219	37.97693	37.98191	37.97217	37.95713	37.95575
54	37.97593	37.98520	37.99233	38.00043	37.99056	37.97440	37.98008	37.98817	37.98168	37.97703	37.97145	37.96138
55	37.97865	37.99460	37.97297	37.98383	37.99527	38.00068	38.00826	37.99519	37.96897	37.96675	37.95770	37.95672
56	37.97871	37.98670	37.97672	37.97440	37.99550	37.97120	37.96967	37.99717	37.97736	37.97275	37.97968	37.96523
57	37.97414	37.97797	37.98816	37.99192	37.96780	37.98415	37.97774	37.97844	37.99203	37.96531	37.97226	37.93934
58	37.96869	37.97435	37.96250	37.96581	37.97331	37.95655	37.98382	37.97144	37.97409	37.96631	37.96764	37.94759
59	37.95772	37.94725	37.97110	37.97905	37.97504	37.96237	37.98182	37.97656	37.97212	37.96762	37.96429	37.93976
60	37.94434	37.95081	37.94275	37.95515	37.98134	37.95863	37.96581	37.95548	37.96573	37.93949	37.93839	37.92030
61	37.92200	37.93006	37.92656	37.93281	37.94999	37.95799	37.96368	37.94849	37.95400	37.92439	37.90535	37.93375

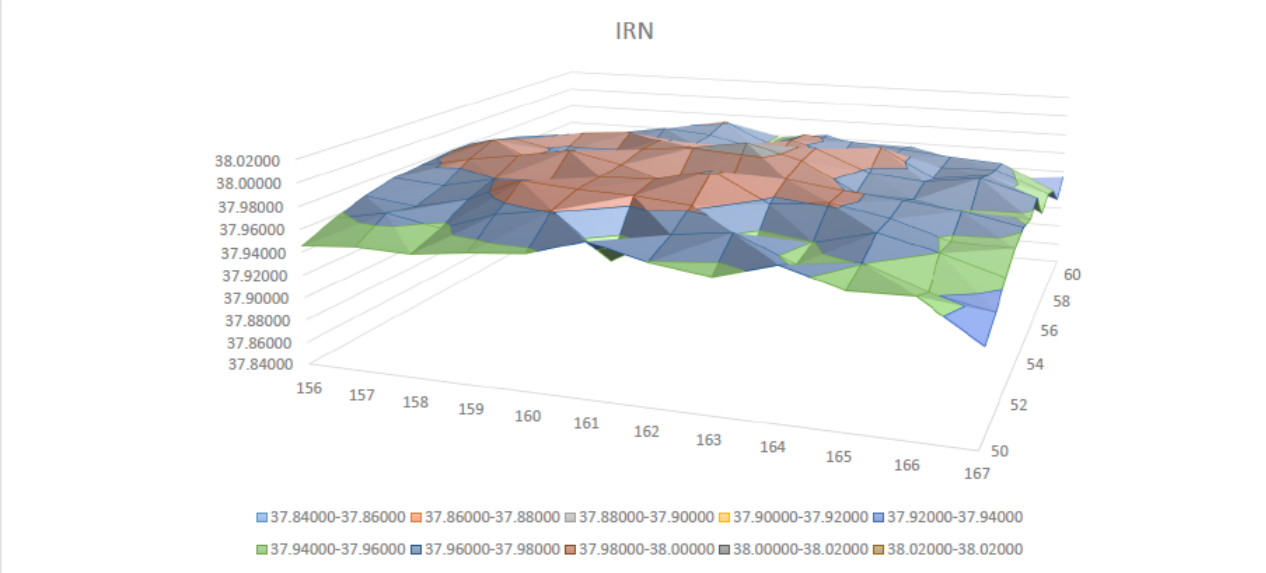

Figure 6.2 – Comparaison de 144 politiques avec IRN

6.2 Simulations avec clProbDist

6.2.1 Le modèle d'inventaire avec des nombres aléatoires non uniformes

Nous avons conçu le modèle d'inventaire décrit à la section précédente pour générer les demandes selon la loi uniforme afin d'expérimenter différentes configurations de streams et de sous-streams. Mais en réalité, la loi de Poisson est plus appropriée pour exprimer la probabilité qu'un nombre donné d'événements survienne dans un intervalle de temps donné. Pour cela, on a adapté le modèle pour que les demandes suivent une distribution de Poisson. Nous avons expérimenté la simulation avec une distribution qui a une moyenne λ fixe sur toutes les périodes puis avec λ qui change d'une période à l'autre suivant une loi Gamma. Nous avons considéré aussi le cas où les demandes suivent une loi normale standard tronquée à zéro. Ces configurations permettront de montrer comment utiliser efficacement les générateurs non uniformes de clProbDist ainsi que d'évaluer leur performance.

Cas (a) : la demande suit une loi de Poisson avec λ fixe

Le code du listing 6.10 ressemble à celui écrit dans le listing 6.1 avec la différence que les demandes sont générées selon une loi de Poisson avec une moyenne fixe. Notons que la distribution est passée en paramètre au kernel dans la mémoire locale ou globale selon l'utilisation de l'option `CLPROBDIST_POISSON_IN_LOCAL_MEMORY`. L'API ne permet pas de copier la distribution de Poisson vers la mémoire privée vu que la taille des tableaux précalculés peut être grande.

Cas (b) : la demande suit une loi de Poisson avec des $\lambda_i \sim$ Gamma (α, β)

Nous avons implémenté ce cas en créant au niveau de l'hôte un tableau de m variables aléatoires selon une loi Gamma ayant des paramètres α et β, puis nous transférons ce tableau vers le dispositif de calcul. Chaque work item choisira pour la période (journée) i l'élément λ_i (pour $1 \leq i \leq m$) comme étant le paramètre de la loi de Poisson à utiliser. Cela est montré dans le listing 6.11 ; notons que le tableau des moyennes peut être transféré soit vers la mémoire globale ou locale selon l'utilisation ou pas de la macro

`CLPROBDIST_POISSON_IN_LOCAL_MEMORY`.

Il faut noter que nous avons exploité au niveau du kernel la fonction de répartition inverse qui utilise directement le paramètre λ et non pas la version de la fonction qui utilise l'objet, cela afin d'éviter de créer m distributions qui auraient différé seulement par les valeurs de leurs moyennes et auraient demandé beaucoup d'espace mémoire.

Cas (c) : la demande suit une loi normale standard tronquée

Pour générer une variable aléatoire X selon une loi normale standard tronquée sur l'intervalle $[a, +\infty]$ et avec des paramètres σ et μ, nous utilisons la formule suivante :

$$X = \Phi^{-1}\Big(\Phi(a) + U(\Phi(+\infty) - \Phi(a))\Big)\sigma + \mu$$

où Φ est la fonction de répartition de la loi normale standard et Φ^{-1} son inverse. Le code du listing 6.12 montre mon implémentation de ce cas ; notons que l'objet de la distribution normale est transféré en mémoire privée. Le $+\infty$ est implanté avec le plus grand `double` qui puisse être représenté sur le dispositif de calcul (`DBL_MAX`).

```
#define CLRNG_ENABLE_SUBSTREAMS
#include <mrg32k3a.clh>

__kernel inventorySimulateOneRunFixedPoissonLambda(int m, int s, int
    S, clrngMrg32k3aStream* stream_demand, clrngMrg32k3aStream*
    stream_order,
#ifdef CLPROBDIST_POISSON_IN_LOCAL_MEMORY
  __local clprobdistPoisson* poissonDist
#else
  __global clprobdistPoisson* poissonDist
#endif)
{
double Xj = S, Yj;  // Stock au début et fin de journée.
double demand, profit = 0.0;
for (int j = 0; j < m; j++) {
 // Génère une variable uniforme
 cl_double u = clrngMrg32k3aRandomU01(stream_demand);

 //Génère la demande avec l'objet distribution et la var. uniforme
 demand = clprobdistPoissonInverseCDFWithObject(poissonDist, u, null);

 // Soustrait la demande du jour.
 Yj = Xj - demand;
 if (Yj < 0)
  Yj = 0; //demande perdue
 profit += c * (Xj - Yj) - h * Yj;

 double prob = clrngMrg32k3aRandomU01(stream_order);
 if ((Yj < s) && (prob < p)) {
    profit -= K + k * (S - Yj);
    Xj = S;
  }
  else
      Xj = Yj;
  }
 return profit / m;
}
```

Listing 6.10 – Modèle d'inventaire avec une loi de Poisson ayant un λ fixe

```
1  //Au niveau de l'hôte :
2  //Création du tableau de m éléments selon la loi Gamma.
3  clprobdistPoisson* gammaDist = clprobdistGammaCreate(alpha, beta,
4                                                  decPrec, null, null);
5  clrngMrg32k3aStream* stream = clrngMrg32k3aCreateStreams(null, 1,
6                                                          null, null);
7  double * lambdaArr = (double*)malloc(m*sizeof(double));

9  for (size_t i = 0; i < m; i++){
10 double u = clrngMrg32k3aRandomU01(stream);
11 lambdaArr[i] = clprobdistGammaInverseCDFWithObject(gammaDist, u,null);
12 }

14 .....
15 //Au niveau du dispositif :
16 //La demande suit une loi de Poisson avec Lambda qui change chaque
       jour
17 __Kernel inventorySimulateDynamicLambdaPoisson (int m, int s, int S,
18                                    clrngMrg32k3aStream* stream_demand,
19                                    clrngMrg32k3aStream* stream_order,
20                              #ifdef CLPROBDIST_POISSON_IN_LOCAL_MEMORY
21                                    __local double* lambdaArray
22                              #else
23                                    __global double* lambdaArray
24                              #endif)
25 {
26 ...
27 for (int j = 0; j < m; j++) {
28     cl_double u = clrngMrg32k3aRandomU01(stream_demand);
29     demand = clprobdistPoissonInverseCDF(lambdaArray[j], u, &err);
30     ...
31     // Soustrait la demande du jour.
32     ...
33     }
34  return profit / m;
35 }
```

Listing 6.11 – Modèle d'inventaire avec une loi de Poisson avec λ dynamique

```
double inventorySimulateOneRunNormal(int m, int s, int S,
                     clrngMrg32k3aStream* stream_demand,
                      clrngMrg32k3aStream* stream_order,
                           clprobdistNormal* normalDist)
{
double Xj = S, Yj;  // Stock au début et fin de journée.
double demand, profit = 0.0;

//Créer la distribution normale tronquée sur [0, DBL_MAX[.
double fa = clprobdistNormalCDFWithObject(normalDist, 0, null);
double fb = clprobdistNormalCDFWithObject(normalDist, DBL_MAX, null);
double fbfa = fb - fa;

double sigma = clprobdistNormalGetSigma(normalDist, null);
double mu    = clprobdistNormalGetMu(normalDist, null);

for (int j = 0; j < m; j++) {
//Générer la demande
cl_double u = clrngMrg32k3aRandomU01(stream_demand);

demand = clprobdistNormalInverseCDFWithObject(normalDist,
                       fa + fbfa * u, null) * sigma + mu;

//Soustrait la demande du jour.
Yj = Xj - demand;
if (Yj < 0) Yj = 0;
profit += c * (Xj - Yj) - h * Yj;
double prob = clrngMrg32k3aRandomU01(stream_order);
if ((Yj < s) && (prob < p)) {
  profit -= K + k * (S - Yj);
  Xj = S;
  }
else
  Xj = Yj;
}
return profit / m;
}
```

Listing 6.12 – Modèle d'inventaire avec une loi normale standard tronquée

6.2.2 Simulation d'une option financière

Cette section décrit la simulation d'une option financière asiatique en utilisant des nombres aléatoires non uniformes avec l'interface clProbdist sur des dispositifs parallèles. Là aussi l'objectif est d'évaluer les fonctionnalités de l'API sur un autre modèle de simulation ; cela nous permettra aussi d'expérimenter différentes configurations et d'étudier la performance des générateurs non uniformes.

De façon similaire au modèle précédent, nous devrons créer un stream au niveau de l'hôte puis on le transférer vers le dispositif pour pouvoir générer les variables aléatoires uniformes qui seront utilisées par clProbdist pour produire des variables aléatoires non uniformes selon la loi normale avec la méthode d'inversion.

6.2.2.1 Description du modèle

Considérons un contrat financier pour un actif donné (une once d'or ou un baril de pétrole, etc.) dont la valeur au temps t vaut $S(t)$. Nous supposons que cette valeur progresse selon un processus stochastique $S(t), t \geq 0$ et avec une loi de probabilité déterminée. On définit $g(S(t_1),..,S(t_d))$ comme étant le *gain net* que le propriétaire du contrat va recevoir en temps $T = t_d$. La valeur du gain dépend uniquement des valeurs de l'actif sur les temps d'observations ; la *valeur actuelle* v du montant à recevoir dans t unités de temps équivaut au produit de cette valeur par le *facteur d'actualisation* e^{-rt}. L'objectif est d'estimer la valeur actuelle équitable du contrat financier $v(s_0,T)$ qui, sous certaines conditions qui définissent la nature du marché (voir chapitre 7 de [28]), s'écrit :

$$v(s_0,T) = \mathbb{E}^* \left[e^{-rt} \times g(S(t_1),..,S(t_d)) \right]$$

où $\mathbb{E}^*$ correspond à l'espérance mathématique sous une certaine mesure de probabilité $\mathbb{P}^*$. Dans le modèle de Black and Scholes, $S(t)$ progresse selon un mouvement brownien

géométrique suivant la formule. :

$$S(t) = S(0)e^{(r-\sigma^2/2)t+\sigma B(t)}$$

où r correspond au *taux d'intérêt*, σ est une constate représentant la *volatilité* et $\{B(t), t \geq 0\}$ représente le mouvement brownien standard.

Nous pouvons simuler le processus et $g(S(t_1),...,(S(t_d))$ pour estimer la valeur de $v(s_0,T)$ avec la méthode Monte-Carlo en générant le chemin $S(t_1),..,S(t_d)$ selon $\mathbb{P}^*$ et en calculant la valeur de $X = e^{-rt}g(S(t_1),..,S(t_d)$, puis en répétant cela n fois pour obtenir n réalisations indépendantes de X notées $X_1,..,X_n$, nous aurons ainsi l'estimateur final

$$\bar{X} = 1/n \sum_{i=1}^{n} X_i.$$

Simulation d'une option asiatique : il s'agit d'un contrat financier qui donne droit à son détenteur d'acquérir un capital à l'expiration du contrat (temps T) lorsque la moyenne arithmétique des valeurs de l'actif sur les périodes sous-jacentes t_d est plus grande qu'une valeur prédéterminée K (appelé *le prix d'exercice*). Dans le cas où la moyenne est en dessous de K, le contrat n'aura pas de valeur. Ainsi le formule du gain net s'écrit :

$$g(S(t_1),...,S(t_d)) = \max\left[0, \frac{1}{d}\sum_{j=1}^{d} S(t_j) - K\right].$$

6.2.2.2 Description de la simulation

Pour simuler le modèle de l'option asiatique et estimer $v(s_0,T)$, nous aurons besoin de générer le chemin des $S(t_j)$ suivant un mouvement brownien décrit par la formule de Black and Scholes avec l'utilisation du logarithme :

$$\ln(S(t_j)) = \ln(S(t_{j-1})) + (r - \sigma^2/2)(t_j - t_{j-1}) + \sigma\sqrt{t_j - t_{j-1}}Z_j$$

où Z_j est une variable aléatoire suivant la loi normale standard.

Le listing 6.13 montre l'utilisation d'un stream de nombres aléatoires uniformes pour générer d variables suivant la loi normale en utilisant la méthode d'inversion par la fonction `clprobdistNormalInverseCDFWithObject`. Celle-ci retourne la valeur du logarithme du gain pour une seule exécution. Ensuite, nous pouvons calculer un estimateur de $v(s_0,T)$ en multipliant le gain net par le facteur d'actualisation e^{-rT}. Le listing 6.14 montre le kernel exécuté par les work items. Durant l'implémentation, nous avons choisi d'utiliser des streams différents à travers les work items et un nouveau sous-stream par itération, cela grâce à la fonction `clrngMrg32k3aForwardToNextSubstreams`, ce qui donne un total de $n = n_1 n_2$ simulations. Cette configuration des streams est similaire à ce qui a été fait dans le cas (c) du modèle d'inventaire ; l'approche a montré son efficacité par rapport à d'autres cas. Il faut noter aussi que pour optimiser l'accès en mémoire au niveau du dispositif, nous avons enregistré les valeurs du gain net dans des positions successives au niveau du tableau de sortie `stat_payOff` (voir le concept de Coalescing au niveau de la section 4.4).

Nous avons expérimenté avec la simulation pour $n = 2^{15}$ et $n_1 = 2^{10}$ sur le même matériel décrit dans la section 6.1.5, ce qui a donné une estimation de la valeur de l'option égale à 13.125 avec une variance $\sigma^2 = 515.64$ et un intervalle de confiance à 95% égal à (12.879, 13.371) ; le temps d'exécution a été de 0.17s sur le CPU et 0.07s sur GPU-A.

```
// Génération du process S.
void generatePath(int d, __global double* logS
        __global double* muDelta,__global double* sigmaSqrtDelta,
       clrngMrg32k3aStream* stream, clprobdistNormal* normalDist,){

for (int j = 0; j < d; j++){
 double u = clrngMrg32k3aRandomU01(stream);
 double Z = clprobdistNormalInverseCDFWithObject(normalDist, u, null);

 logS[j + 1] = logS[j] + muDelta[j] + sigmaSqrtDelta[j] * Z;
 }
}
```

Listing 6.13 – Génération du processus brownien géométrique

```
//Chaque work item exploite un stream différent avec utilisation de
//n2 sous-streams au niveau des réplications du work item
__kernel void AsianOptGPU(__global clrngMrg32k3aHostStream* streams,
                          __global clprobdistNormal* g_normalDist,
                __global double* stat_payOff, __global double* logS,
          __global double* muDelta, __global double* sigmaSqrtDelta)
{
int gid = get_global_id(0); // Index du work item.
int n1 = get_global_size(0); // Nombre total des work items

//Fait une copie du stream dans la mémoire privée
clrngMrg32k3aStream stream_d;
clrngMrg32k3aCopyOverStreamsFromGlobal(1, &stream_d, &streams[gid]);

//Fait une copie de la distribution normale dans la mémoire privée
clprobdistNormal p_normalDist = *g_normalDist;

//Effectue n2 exécutions
for (int i = 0; i < n2; i++) {
  generatePath(d, logS, muDelta, sigmaSqrtDelta, &stream_d,
                                          &p_normalDist);

  stat_payOff[i * n1 + gid] = getPayoff(d, strike, discount, logS);

  clrngMrg32k3aForwardToNextSubstreams(1, &stream_d);
 }
}
```

Listing 6.14 – Kernel qui implémente l'option asiatique

6.2.3 Tests de performance de clProbDist

Pour tester la performance de l'interface clProbDist, nous avons implémenté des kernels qui génèrent n nombres aléatoires avec la loi de Poisson puis avec la loi normale, ceci pour mesurer le temps nécessaire à $n_1 = 2^{20}$ work items d'effectuer cette opération triviale. L'exécution s'est réalisée sur les GPUs décrits à la section 6.1.5. Les cas suivants ont été considérés, chaque mesure a été exécutée trois fois et les résultats sont donnés en secondes.

Cas (a) : Distribution de Poisson avec plusieurs λ et en utilisant la fonction CDF avec objet ou en utilisant directement λ :
Ce cas génère $n = 2^{26}$ nombres aléatoires en exploitant premièrement la version de la fonction de répartition qui utilise l'objet de la distribution comme paramètre et qui exploite les tableaux précalculés pour retourner la valeur du CDF. Ensuite, nous expérimentons la fonction de répartition qui utilise directement le paramètre λ ; dans ce cas, nous ne précalculons pas de tableau, mais nous recalculons la CDF à chaque fois qu'une valeur est générée. Nous notons (1) et (2) ces deux approches.

Le tableau 6.III montre le temps d'exécution en secondes des deux approches (1) et (2). Nous notons que la génération de nombres aléatoires non uniformes en utilisant la fonction inverse du CDF `clprobdistPoissonInverseCDF(lambda,u, null)` qui n'utilise pas les tableaux précalculés est beaucoup plus lente que l'approche utilisant les tableaux. Ceci était prévisible parce que dans la première approche on fait une recherche dichotomique sur le tableau CDF déjà calculé, alors que dans la seconde approche, nous calculons la somme des termes de la distribution de Poisson jusqu'à ce que la valeur de cette somme soit inférieure u (voir détails dans la section 5.2.1.5). Notons aussi que le GPU-D offre de meilleures performances que le GPU-A, cela parce que le premier dispositif graphique a 4 fois plus de CU, plus de mémoire et plus de fréquence d'horloge. Notons aussi que plus la valeur de λ augmente, plus il y a un ralentissement dans l'exécution. Cela s'explique par le temps nécessaire aux work items de trouver la bonne valeur de la variable aléatoire quand le tableau CDF est très grand. L'utilisation

de la fonction CDF paramétrée (colonne 2) au niveau du GPU-A avec un grand $\lambda = 500$ a fait "crasher" le dispositif de calcul qui n'a pas pu donner de résultat, nous pensons que cela est dû à un dépassement de capacité de la mémoire (overflow) survenu lors du calcul de la variable aléatoire non uniforme avec la fonction CDF n'exploitant pas les tableaux précalculés. Mais lorsqu'on utilise un tableau précalculé, ce ralentissement est relativement minime, ce qui est très rassurent et suggère que le temps de recherche dans le tableau n'augmente pas beaucoup lorsque λ augmente.

Lambda	(1) CDF utilisant l'objet		(2) CDF utilisant Lambda	
	GPU-A	GPU-D	GPU-A	GPU-D
2	0.258	0.084	0.901	0.078
	0.237	0.076	0.879	0.081
	0.236	0.080	0.765	0.077
100	0.295	0.086	5.248	0.360
	0.295	0.091	5.249	0.355
	0.293	0.087	5.249	0.360
500	0.313	0.089	-	1.270
	0.314	0.089	-	1.270
	0.313	0.094	-	1.270

Tableau 6.III – Temps en secondes de génération de 2^{26} variables aléatoires par la distribution de Poisson avec plusieurs λ et en utilisant la fonction CDF avec objet ou en utilisant directement λ

Cas (b) : Utilisation de la distribution de Poisson en mémoire locale vs globale : Dans ce cas, nous avons généré 2^{30} nombres aléatoires avec la fonction CDF utilisant l'objet de la distribution. On veut comparer la performance d'exécution lorsque cet objet est copié en mémoire locale ou en mémoire globale (voir section 4.3 pour la différence entre ces deux mémoires).

Le tableau 6.IV montre que le temps (en secondes) de génération des nombres aléatoires non uniformes est plus lent quand nous copions l'objet de la distribution dans la mémoire globale, ce qui est aussi prévisible vu que les work items accèdent plus rapidement à leur mémoire locale qu'à leur mémoire globale. Nous pensons que l'utilisation de la mémoire privée ne constitue pas une bonne approche si l'on veut utiliser les ta-

bleaux précalculés vu que ces derniers peuvent être assez grands. Notons aussi que la performance du GPU discret est meilleure que celle du GPU-A (cela peut être attribué à la différence dans le nombre de CU et de taille mémoire entre les deux dispositifs). Le listing 6.15 illustre le code utilisé pour exécuter le cas (b). Notons l'utilisation de la commande `barrier(CLK_LOCAL_MEM_FENCE)` qui permet de synchroniser les work items. En effet, lors de l'utilisation de la mémoire locale, seul le work item qui a un index égale à 0 devra copier la distribution de Poisson de la mémoire globale vers la mémoire locale ; tous les autres work items attendront devant la barrière et une fois que la copie sera effectuée, les work items continuerons leurs exécutions.

(1) Mémoire locale		(2) Mémoire globale	
GPU-A	GPU-D	GPU-A	GPU-D
3.277	0.576	4.312	0.745
3.234	0.568	4.312	0.753
3.234	0.576	4.312	0.745

Tableau 6.IV – Temps en secondes de génération de 2^{30} variables aléatoires par la distribution de Poisson copiée en mémoire locale vs globale

Cas (c) : La distribution normale en mémoire privée vs constante vs globale :
Dans ce cas, nous avons aussi généré $n = 2^{30}$ nombres aléatoires non uniformes en utilisant la distribution normale ; nous voulons comparer le temps nécessaire à cela quand nous copions puis exploitons l'objet de la distribution dans la mémoire privée, la mémoire constante et la mémoire globale. Donc, ici, nous utilisons la fonction de l'inverse du CDF qui manipule l'objet de la distribution (voir ligne 29 du listing 6.15). Nous notons à partir du tableau 6.V que les temps d'exécution en secondes dans les colonnes (1), (2) et (3) sont similaires à travers le même type de GPU ; nous pensons que cela est dû à l'optimisation effectuée par le dispositif de calcul notamment en utilisant la mémoire cache. Nous notons aussi que le GPU-A est deux fois plus rapide que le GPU-D ; cela peut être dû au fait que le GPU discret prend plus de temps à effectuer le calcul d'inversion.

```
1  __kernel void performancePoissonGlobalLocal(__global
       clrngMrg32k3aHostStream* streams,
2  #if CLPROBDIST_POISSON_OBJ_MEM == CLPROBDIST_MEM_TYPE_LOCAL
3    __global clprobdistPoisson* g_poissonDist,
4    __local  clprobdistPoisson* poissonDist
5  #else
6    _CLPROBDIST_POISSON_OBJ_MEM clprobdistPoisson* poissonDist
7  #endif
8  )
9  {
10   // Chacun des n1 work items exécute le code suivant.
11   int gid = get_global_id(0); // Id du work item.
12   int group_size = get_local_size(0);

14   // Fait une copie du stream en mémoire privée.
15   clrngMrg32k3aStream stream_d;
16   clrngMrg32k3aCopyOverStreamsFromGlobal(1, &stream_d, &streams[gid]);

18 #if CLPROBDIST_POISSON_OBJ_MEM == CLPROBDIST_MEM_TYPE_LOCAL
19   //Le premier work item de chaque work group copie la distribution
         de la mém. globale vers la mém locale.
20   if (gid % group_size == 0)
21     clprobdistPoissonCopyOverFromGlobal(poissonDist, g_poissonDist);

23   //Tous les work items attende ici jusqu'à ce que le premier work
         item ait fini de copier.
24   barrier(CLK_LOCAL_MEM_FENCE);
25 #endif

27   //Effectue la génération de n2 nbr aléatoires
28   for (int i = 0; i < param_n2; i++) {
29     clprobdistPoissonInverseCDFWithObject(poissonDist,
         clrngMrg32k3aRandomU01(&stream_d), &err);
30   }
31 }
```

Listing 6.15 – Kernel utilisant la distribution de Poisson en mémoire locale vs globale

(1) Mémoire privée		(2) Mémoire constante		(3) Mémoire globale	
GPU-A	GPU-D	GPU-A	GPU-D	GPU-A	GPU-D
0.043	0.049	0.023	0.048	0.023	0.048
0.023	0.044	0.023	0.043	0.023	0.043
0.023	0.043	0.022	0.043	0.023	0.043

Tableau 6.V – Temps en secondes de génération de 2^{30} variables aléatoires par la distribution normale copiée en mémoire privée vs constante vs globale

6.2.4 Analyse de la divergence des WI utilisant la distribution de Poisson

Dans le cas de la distribution de Poisson, nous nous sommes intéressés à l'étude de la performance des work items (WI) dans un warp de taille w pour générer une v.a. de Poisson en utilisant le tableau précalculé. Pour rappel, nous utilisons la fonction `clprobdistPoissonInverseCDFWithObject(dist, u)` pour générer une v.a. non uniforme X en utilisant la méthode d'inversion avec une variable aléatoire uniforme U, tel que :

$$X = F^{-1}(U) = \min\{x \mid F(x) \geq U\}.$$

L'algorithme d'inversion est implémenté sous forme de recherche dichotomique dans le tableau CDF de la distribution, comme c'est détaillé à la section 5.2.1.5. Nous voulons savoir dans quelle mesure notre implémentation de la recherche va créer de la divergence entre les WI, notamment à cause de l'utilisation des instructions de branchements (if). Nous sommes partis de l'hypothèse que cela aurait un effet négatif sur la vitesse d'exécution des WI dans le warp. La section 4.4 contient une explication détaillée du phénomène de la divergence.

6.2.4.1 Calcul du nombre d'itérations dans la recherche dichotomique

Définissions Y comme étant le nombre d'itérations effectué par la recherche dichotomique dans le tableau CDF pour générer **une seule** variable de Poisson X. On compte une iteration dans le premier if de l'algorithme 4, lorsque la fonction commence la recherche d'un coté ou de l'autre de x_{med}. Ensuite, ce nombre d'itération est incrémenté à chaque itération de la boucle *while*. L'exemple du paragraphe suivant illustre en détail un cas concret de cette recherche dichotomique. Il est clair que Y est aussi une variable aléatoire. Nous nous intéressons dans un premier temps à calculer la fonction de masse du nombre d'itérations Y. Pour cela, nous avons modifié la fonction `clprobdistPoissonInverseCDFWithObject(dist,u, out y)` telle

que pour chaque valeur x_i de X qui peut être retournée, la fonction retourne aussi le nombre y_i d'itérations requis dans le paramètre `out y`. Ainsi, cette dernière valeur rapporte simplement combien de fois nous avons effectué de bissection. Cela a donné la tableau 6.VI.

Par exemple avec $\lambda = 2$, en utilisant une uniforme U. On commence par regarder la valeur de la CDF à $x_{med} = 2$ et on compare avec U. Si $U \leq F(2)$ nous vérifions d'abord si $U \leq F(0)$ dans ce cas nous retournons directement la valeur du paramètre de la distribution $x_{min} = 0$ et nous reportons l'exécution d'une seule itération ($Y = 1$). Autrement, si $U > F(0)$, nous initialisons deux variables $i = 0$ et $j = x_{med} = 2$ et nous exécutons une boucle qui calcule à chaque itération la variable $k = (i + j)/2$. Nous testons si $U > F(k)$. Dans le cas où cette condition serait vraie nous affectons la valeur de $k+1$ à i sinon nous affectons la valeur de k au variable j. Enfin, nous incrémentons la valeur de la variable qui compte le nombre d'itérations puis nous exécutons cette boucle tant que : $i < j$.

Dans le cas où $U \leq F(2)$, puisque le tableau CDF contient les valeurs de la fonction de répartition inverse dans sa moitié supérieure ($x > 2$), nous remplaçons la valeur de U par $1 - U$. Puis, de façon similaire, nous vérifions d'abord si $U < F(x_{max})$, auquel cas nous reportons une seule itération et nous retournons la valeur $x_{max} = 22$. Autrement, si $U \geq F(x_{max})$, nous initialisons la valeur de $i = x_{med} + 1 = 3$ et nous initialisons la valeur de $j = x_{max}$, puis nous exécutons une boucle qui calcule la valeur de $k = (i + j)/2$, ensuite nous vérifions la condition $U < F(k)$. Si cette dernière est vraie alors i aura la valeur de $k + 1$ sinon j aura la valeur k. Puis nous incrémentons le compteur d'itération et nous réexécutons la boucle tant que ($i < j$). Enfin, la fonction retourne la valeur de i et le nombre d'itération y.

Ainsi, nous pouvons calculer la fonction de masse de Y de façon exacte comme le montre le tableau 6.VII. Nous notons dans le tableau 6.VII (λ=2) que l'algorithme de recherche aura besoin, dans 67% du temps, d'effectuer au plus 3 itérations avant de trouver la v.a. X. Pour le cas de λ= 10, le tableau 6.VIII montre que pour 80% des cas la recherche nécessitera entre 5 et 6 itérations. Pour $\lambda = 100$ le nombre d'itérations varie entre 7 et 8 de façon assez proche, tandis que pour un grand $\lambda = 1000$, l'algorithme aura besoins de 9 itérations 93% des fois. Notons que le reste des valeurs de y_i ont

des probabilités exactement égales à zéro. Cela s'explique par le fait que la recherche dichotomique a toujours besoin de soit $\lfloor \log_2 k \rfloor$ ou $\lceil \log_2 k \rceil$ itérations pour trouver la valeur recherchée, telle que k est la taille de la partie du tableau CDF dans laquelle on débute la recherche [5].

Par exemple, la tableau 6.IX contient le nombre d'itérations effectué par la recherche dichotomique pour chacune des valeurs de λ, dans le cas où on ne s'arrête pas à cause de la condition $U \leq CDF[0]$ et le cas similaire lorsque l'on va à droite. Ainsi, les valeurs du tableau ont été calculées en appliquant les deux formules logarithmiques précédentes (plus la valeur 1) sur les tailles k_L et k_M qui correspondent respectivement au nombre d'éléments du tableau CDF qui sont inférieurs et supérieurs à la médiane x_{med} pour chaque λ. L'ajout de la valeur 1 à la formule logarithmique s'explique par le fait que la recherche dichotomique effectuée par la fonction `clprobdistPoissonInverse-CDFWithObject(dist,u, out y)` sur les k éléments du tableau exécute une itération au début de la recherche pour déterminer quel côté du tableau choisir, cette étape n'est pas pris en compte dans le calcul avec la formule logarithmique dans le tableau 6.IX puisque nous prenons en considération seulement k_L et k_M.

6.2.4.2 Fonction de répartition du maximum des Y_i

a) Cas de l'exécution du warp sans (ou avec peu de) divergence

Pour un warp contenant w WI et dans le cas où il n'y a pas de divergence, le temps que prendra le warp pour finir son exécution sera égal au temps pris par le WI le plus lent. Notons dans ce cas $M = \max(Y_1, ..., Y_w)$ comme étant le maximum des Y_i qui sera inférieur ou égal à y si et seulement si tous les Y_i sont inférieurs à ce y et on écris :

$$P(M \leq y) = P(Y_i \leq y : \forall i) = \prod_{i=1}^{w} P(Y_i \leq y)$$

donc, nous pouvons calculer le CDF du maximum des y_i à partir du CDF $F_Y(y)$ de Y comme suit : $F_M(y) = P(M \leq y) = (F_Y(y))^w$.

b) Cas de l'exécution du warp avec divergences des WI

x_i	$P(X = x_i)$	y_i
0	0.135335283	1
1	0.270670566	3
2	0.270670566	2
3	0.180447044	6
4	0.090223522	5
5	0.036089409	5
6	0.012029803	6
7	0.003437087	6
8	0.000859272	5
9	0.000190949	5
10	0.000038190	5
11	0.000006944	5
12	0.000001157	6
13	0.000000178	6
14	0.000000025	5
15	3.39126E-09	5
16	4.23908E-10	6
17	4.98715E-11	6
18	5.5413E-12	5
19	5.833E-13	5
20	5.83E-14	5
21	5.6E-15	5
22	5.0E-16	5

Tableau 6.VI – La fonction de masse de X pour une loi de Poisson avec $\lambda = 2$ et avec les nombres d'itérations y_i

y_i	$P(Y = y_i)$
1	0.135335283
2	0.270670566
3	0.270670566
5	0.127408314
6	0.195915269

Tableau 6.VII – La fonction de masse de Y pour une loi de Poisson avec $\lambda = 2$

λ= 10	
y_i	$P(Y = y_i)$
1	0.000045400
4	0.402922375
5	0.180071975
6	0.406131320
7	0.010828930

λ= 100	
y_i	$P(Y = y_i)$
1	2.0E-16
7	0.662142131
8	0.337287869

λ= 1000	
y_i	$P(Y = y_i)$
8	0.022333385
9	0.924949831
10	0.052716784

Tableau 6.VIII – La fonction de masse de Y pour une loi de Poisson avec différents λ

Dans le cas de la divergence, le warp aura deux chemins d'exécution (branches) à effectuer ; un premier qui sera exécuté par une partie des WI qui ont évalué la condition

	$\lambda = 2$		$\lambda = 10$		$\lambda = 100$		$\lambda = 1000$	
	$k_L = 3$	$k_H = 20$	$k_L = 10$	$k_L = 35$	$k_L = 71$	$k_L = 92$	$k_L = 249$	$k_L = 259$
$1 + \lfloor \log_2 k \rfloor$	2	5	4	6	7	7	8	9
$1 + \lceil \log_2 k \rceil$	3	6	5	7	8	8	9	10

Tableau 6.IX – Le nombre d'itération calculé en fonction de λ

à vrai et un autre chemin pour le reste des WI. Cela correspond à une exécution en série pour la portion de code soumise à la formule conditionnelle. Cela implique que le temps que prendra le warp pour finir son exécution sera potentiellement plus important que le temps pris par l'exécution parallèle des WI n'ayant pas de divergences (cas précédent).

6.2.4.3 Analyse du gaspillage du temps dans le warp

Pour savoir dans quelle mesure le phénomène de la divergence des WI affectera la performance du warp dans la génération des variables aléatoires nous avons effectué les étapes suivantes :

1. Nous exécutons, au niveau du GPU-A, un seul work item (w=1) qui génère un million de variables aléatoires selon la loi de Poisson avec tableau pour $\lambda = 2$, et nous mesurons le temps d'exécution en secondes. La valeur du temps trouvée constitue le temps de référence qui sera utilisé pour calculer le pourcentage d'augmentation du temps d'exécution (ralentissement) de la simulation provenant de la divergence ou d'autre forme "d'overhead." Nous ferons cela avec différentes valeurs de λ et de w.
2. Nous répétons l'étape (1) et nous fixons la taille du warp à $w = 2$ et nous modifions la valeur de λ de 2 à 2000. Cela en mesurant le temps d'exécution.
3. Nous effectuons la même chose que précédemment mais pour $w = 32$ puis pour $w = 64$, c.-à-d. nous exécutons d'abord 32 work items et chacun produira à lui seul un million de variables aléatoires pour différentes λ, et nous mesurons le temps d'exécution. Enfin nous effectuons la même chose avec 64 work items.

Dans les tous les cas, nous nous sommes assurés d'utiliser la même séquence de variables aléatoires uniformes pour chacun des WI, cela pour permettre de comparer

seulement l'impact qu'a eu la modification de λ ou w.

Cette expérimentation a donné les 15 temps d'exécutions illustrés dans la colonne (a) du tableau 6.X, dans lequel nous notons aussi les colonnes suivantes :

— La colonne (b) correspond au pourcentage de ralentissement des work item(s) pour la même taille de warp, mais pour différentes valeurs de λ. Ainsi, dans le cas d'un seul WI, nous notons que lorsque la valeur de λ passe de 2 à 2000 le temps d'exécution augmente de 70%. Cela s'explique par le fait qu'en augmentant λ, la taille N_{max} du tableau précalculé augmente aussi selon la formule $N_{max} = (\lambda + 16 \times (2 + \sqrt{\lambda}))$ - voir section 5.2.1.5, ce qui fait que le WI passera plus de temps à faire la recherche dichotomique.

— La colonne (c) représente le pourcentage de ralentissement de la simulation pour la même valeur de λ, mais pour différentes valeurs de w (taille du warp). Nous notons par exemple dans la première ligne du tableau (pour $\lambda = 2$) que le warp ayant 64 WI a eu besoin de 96% plus de temps pour générer les 64 millions v.a. comparativement au cas d'un seul WI (qui a généré 10^6 variables aléatoires). Le ralentissement dans ce cas est dû à la divergence des WI lors de l'exécution parallèle. Plus la taille du warp est grande, plus il y a risque qu'un WI ralentisse les autres WI qui sont dans le même warp.

Globalement, nous notons que la génération des v.a. de Poisson avec tableau ne crée pas beaucoup de divergence, parce que si l'on prend le cas de $w = 64$ et en supposant que chaque work item avait divergé complètement par rapport à chaque autre work item du même warp, alors cela aurait donné un facteur de ralentissement de 64 par rapport au temps qu'aurai pris un seul work item, alors que dans le cas présent pour $\lambda = 2000$, si l'on divise le temps pris par 64 WI sur le temps pris par un seul WI (23.68 / 68.33) nous obtenons un facteur de ralentissement d'environ 2.9. Cela indique aussi que le warp n'a pas perdu beaucoup de parallélisme lors de l'exécution de la recherche dichotomique.

Tableau 6.X – Temps d'exécution (en secondes) et pourcentages de ralentissement d'un warp suivant différents λ et différents w pour la génération de v.a. de Poisson avec tableau

	w= 1 WI		w= 32 WI			w= 64 WI		
λ	**(a) Temps d'exéc.**	(b) Ralent. par λ	**(a) Temps d'exéc.**	(b) Ralent. par λ	(c) Ralent. par w	**(a) Temps d'exéc.**	(b) Ralent. par λ	(c) Ralent. par w
2	**13.89**		**26.24**		89%	**27.26**		96%
10	**16.80**	21%	**34.50**	31%	105%	**36.11**	32%	115%
100	**20.18**	45%	**47.01**	79%	133%	**50.20**	84%	149%
1000	**22.81**	64%	**59.38**	126%	160%	**65.10**	139%	185%
2000	**23.68**	70%	**62.59**	139%	164%	**68.33**	151%	188%

CHAPITRE 7

CONCLUSION

Nous avons présenté dans ce mémoire la conception et le développement des interfaces de programmation *clRNG* et *clProbDist* pour la génération de nombres aléatoires sur dispositifs de calcul parallèles. Nous avons vu comment grâce à OpenCL nous avons pu faire abstraction de la dissimilarité du matériel et implémenter différents types de générateurs qui produisent des nombres aléatoires uniformes et non uniformes. Ces générateurs ont des streams et sous-streams qui peuvent être créés au niveau du hôte et utilisés par les work items au niveau des dispositifs de calcul. Nous avons aussi évalué différentes configurations pour chaque API sur plusieurs dispositifs. Ces interfaces de programmation sont déjà disponibles sur Internet et peuvent être exploitées par les utilisateurs. Nous pensons que ce travail a permis de produire d'excellents outils nécessaires surtout aux simulations dans le contexte parallèle. Ce travail nous a permis d'apprendre les détails d'implémentation et de tests de différents types de générateurs. Notre étude de l'environnement OpenCL est définitivement un atout qui sera utilisé dans nos futurs projets afin d'optimiser la performance des applicatifs. Toute la démarche de conception et d'implémentation nous a permis de nous imprégné du processus de recherche scientifique dans la collecte d'informations ainsi que dans l'expérimentation, l'analyse et la présentation des résultats.

À l'avenir, il serait intéressant de modifier les algorithmes utilisés dans la génération de nombres aléatoires pour qu'ils exploitent encore plus le parallélisme au niveau des dispositifs, par exemple pour effectuer le calcul matriciel de façon parallèle au niveau du PE ou le remplacement des boucles à l'intérieur des kernels par des appels récursifs grâce à l'utilisation de la nouvelle version d'OpenCL 2.0 ; il faudra expérimenter la différence de performance dans ce cas. Nous avons noté que le temps nécessaire au lancement des kernels était un peu long parce que l'environnement OpenCL prend du temps à compiler toute l'API à chaque fois, même si le work item avait besoin de quelques fonctions

seulement ; pour remédier à cela, les deux APIs doivent être divisées en plusieurs petits modules qui peuvent être inclus dans les kernels de façon indépendante. Pour clProbDist, il faut offrir aux utilisateurs plus de choix dans les lois de probabilité implémentées. Nous pourrons aussi revoir et adapter la conception des API pour qu'elles puissent être exploitées sur d'autres types de dispositifs tels que les FPGA et MPPA.

BIBLIOGRAPHIE

[1] M. Anker. Pseudo Random Number Generators on Graphics Processing Units, with Applications in Finance. *Mémoire de maîtrise à l'Université d'Edinburgh*, 2013.

[2] R. Banger et K. Bhattacharyya. OpenCL programming by example. ISBN-10 : 1849692343, 2013.

[3] G. P. Bhattacharjee. The incomplete gamma integral. *Applied Statistics*, (19):285–287, 1970.

[4] J. M. Blair, C. A. Edwards et J. H. Johnson. Rational Chebyshev approximations for the inverse of the error function. *Mathematics of Computation*, (30):827–830, 1976.

[5] G. Brassard et P. Bratley. Algorithmics : theory & practice. *Prentice-Hall, ISBN-10 : 0130232432*, 1988.

[6] R. P. Brent. An algorithm with guaranteed convergence for finding a zero of a function. *Computer Journal*, (14):422–425, 1971.

[7] R. P. Brent. Algorithms for minimization without derivatives. *Prentice-Hall, Englewood Cliffs*, 1973.

[8] P. Coddington. Tests of random number generators using Ising model simulations. *International Journal of Modern Physics C*, 7(3):295–303, 1996.

[9] R. Dror, M. Moraes, D. Shaw et J. Salmon. Parallel random numbers : as easy as 1, 2, 3. *In Proceedings of 2011 International Conference for HPC, Networking, Storage and Analysis*, 2011.

[10] J. Granger-Piché. Générateur pseudo-aleatoires combinant des récurrences linéaires et non linéaires. *Mémoire de maîtrise à l'Université de Montréal*, 2001.

[11] D. Kunzman. Programming heterogeneous systems. *Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW)*, pages 2061–2064, 2011.

[12] P. L'Ecuyer. Random numbers for simulation. *Communications of the ACM*, 33 (10):85–97, 1990.

[13] P. L'Ecuyer. Uniform random number generation. *Annals of Operations Research*, 53(1):77–120, 1994.

[14] P. L'Ecuyer. Combined multiple recursive generators. *Annals of Operations Research*, 44(5):816–822, 1996.

[15] P. L'Ecuyer. Maximally equidistributed combined tausworthe generators. *Mathematics of computation*, 65(213):203–213, 1996.

[16] P. L'Ecuyer. Good parameters and implementations for combined multiple recursive random number generators. *Annals of Operations Research*, 47(1):159–164, 1999.

[17] P. L'Ecuyer. Tables of maximally-equidistributed combined LFSR generators. *Mathematics of computation*, 68(225):261–269, 1999.

[18] P. L'Ecuyer. Variance Reduction's Greatest Hits. *Proceedings of the 2007 European Simulation and Modeling Conference.*, pages 5–12, 2007.

[19] P. L'Ecuyer. clRNG : A library for uniform random number generation in OpenCL. `http://simul.iro.umontreal.ca/clrng/indexe.html/`, 2015. [En ligne ; accédé le 07-Août-2015].

[20] P. L'Ecuyer et P. Hellekalek. Random number generators : selection criteria and testing in random and quasi-random point sets. *Lectures Notes In Statistics*, 138: 223–266, 1998.

[21] P. L'Ecuyer, L. Meliani et J. Vaucher. SSJ : a framework for stochastic simulation in Java. *Proceedings of the 34th conference on Winter simulation : exploring new frontiers.*, pages 234–242, 2002.

[22] P. L'Ecuyer, D. Munger et N. Kemerchou. clRNG : a random number API with multiple streams for OpenCL. `http://www.iro.umontreal.ca/~lecuyer/myftp/papers/clrng-api.pdf`, 2015. [En ligne ; accédé le 07-Août-2015].

[23] P. L'Ecuyer, B. Oreshkin et R. Simard. Random numbers for parallel computers : requirments and methods, with emphasis on GPUs., submitted 2014, revised april 2015.

[24] P. L'Ecuyer et R. Simard. TestU01 : a C Library for empirical testing of random number generators. *ACM Transactions on Mathematical Software*, 33(4), 2007.

[25] P. L'Ecuyer, R. Simard, E. Jack Chen et W. D. Kelton. An object-oriented random-number package with many long streams and substreams. *Annals of Operations Research*, 50(6):1073–1075, 2002.

[26] P. L'Ecuyer, R. Simard et S. Wegenkittl. Sparse serial tests of uniformity for random number generators. *SIAM Journal on Scientific Computing*, 24(2):652–668, 2002.

[27] P. L'Ecuyer et R. Touzin. Fast Combined Multiple Recursive Generators with Multipliers of the form a=$\pm 2^q \pm 2^r$. *Proceedings of the 32nd conference on Winter simulation*, pages 683–689, 2000.

[28] C. Lemieux. Monte Carlo and Quasi-Monte Carlo sampling. *Springer Series in Statistics, ISBN : 978-0-387-78164-8*, 2009.

[29] N. Nandapalan, R. Brent, L. Murray et A. Rendell. High-performance pseudo-random number generation on graphics processing units. *Parallel Processing and Applied Mathematics*, 7203:609–618, 2012.

[30] B. Natarajan. AMD Releases clRNG, an OpenCL Random Number Library. `http://developer.amd.com/community/blog/2015/05/05/amd-releases-clrng-an-opencl-random-number-library/`, 2015. [En ligne ; accédé le 07-Août-2015].

[31] J. Passerat-Palmbach. Contributions to parallel stochastic simulation : Application of good software engineering practices to the distribution of pseudorandom streams in hybrid Monte-Carlo simulations. *Thèse de PhD, Université Blaise Pascal-Clermont-Ferrand II*, 1(30), 2013.

[32] AMD Accelerated Parallel Processing. OpenCL programming Guide. *AMD technical guide*, 2013.

[33] M. Scarpino. OpenCL in action : how to accelerate graphics and computation. ISBN-10 : 1617290173, 2012.

[34] J. L. Schonfelder. Chebyshev expansions for the error and related functions. *Mathematics of Computation*, (32):1232–1240, 1978.

[35] H. Sutter. The Free lunch is over : a fundamental turn toward concurrency in software. `http://www.gotw.ca/publications/concurrency-ddj.htm`, 2009. [En ligne ; accédé le 07-Août-2015].

[36] D. Thomas, L. Howes et W. Luk. A comparison of CPUs, GPUs, FPGAs, and massively parallel processor arrays for random number generation. *Proceedings of the ACM/SIGDA international symposium on Field programmable gate arrays*, pages 63–72, 2009.