RÉPUBLIQUE ALGÉRIENNE DÉMOCRATIQUE ET POPULAIRE
Ministère de l'Enseignement Supérieur et de la Recherche Scientifique

Université Mohamed Khider -BISKRA-
Faculté des Sciences Exactes, et des Sciences de la Nature et de la Vie
Département d'informatique
Laboratoire **LESIA**

Thèse

Présentée pour l'obtention du diplôme de **DOCTORAT LMD** en

Informatique

Spécialité : **Techniques de l'image et de l'intelligence artificielle**

Exploitation des potentialités des cartes graphiques à l'optimisation des algorithmes évolutionnaires

---

Application en Vie Artificielle

Présentée par : NOUR EL-HOUDA BENALIA

Jury :

Pr. Mohamed Chaouki BATOUCHE	Université de Constantine 2	Président
Pr. NourEddine DJEDI	Université de Biskra	Directeur de thése
Pr. Mohamed BENMOHAMED	Université de Constantine 2	Examinateur
Pr. Foudil CHERIF	Université de Biskra	Examinateur
Dr. Mohamed Chaouki BABAHENINI	Université de Biskra	Examinateur

Année Universitaire 2014/2015

À Mes cher parents,

Mes frères Et Mes soeurs,

Et à la mémoire de notre chère collègue Amina Benatti

Remerciements

Je tiens premièrement à me prosterner en remerciant Allah le tout puissant de m'avoir donné le courage et la patience pour terminer ce travail.

Je tiens tout d'abord à exprimer ma reconnaissance approfondi à mon professeur *Mr. Noureddine Djedi*, mon directeur de thèse, qui s'est toujours montré à l'écoute et était très disponible tout au long de la réalisation de cette thèse. Je lui remercie infiniment et chaleureusement encore pour son implication à la réalisation de ce travail de recherche et pour le support qu'il m'a apporté, sa patience, et la pertinence de ses conseils m'ont été d'une aide précieuse tout au long de ce travail.

Je remercie également mon professeur, *Mr. Yves DUTHEN*, pour m'avoir accueilli au sein de l'équipe Vortex du laboratoire de recherche IRIT de Toulouse en France et pour l'aide et les conseils qu'il m'a fournis tout au long de mes séjours au sein de son équipe.

Je voudrais aussi remercier les membres du laboratoire de recherche en informatique (LESIA) à commencer par mon encadreur et le chef du laboratoire, *Mr. Foudil CHERIF*, sans oublier mes collègues : Nesrine OUANNES, Saida KALWACH, Sara KHEMLICH, Marwa GRID, Khadidja ABID, Samah BENMERBI, Nafissa REZKI, sans oublier ma soeur et ma copine d'étude Ahlem BENTRAH. Un remerciement à une très chère amie tunisienne Soukaina CHOUARI.

Mes remerciements vont aussi aux enseignants du département d'informatique de l'université de Biskra.

Mes remerciements vont particulièrement aux membres de jury : *Pr. Mohamed Chaouki Batouche*, *Pr. Mohamed Benmohamed*, *Pr. Foudil cherif*, et *Dr. Mohamed Chaouki Babahenini* pour m'avoir honoré par leur évaluation de ce travail.

Enfin, mes remerciements qui ne vont jamais être suffisants à mes parents pour leur soutien et patience avec moi tout au long de la réalisation de ce travail. Mes remerciements aussi à ma soeur Douâa et mon frère Ahmed Yacine pour leur aide, et je ne peux pas oublier bien sûr ma petite soeur Hanâa et mon petit frère Abd El-Raouf. Rien de tout ceci ne serait possible sans leurs sacrifices et leurs encouragements.

ملخص

تعتبر هذه الأطروحة من بين الدراسات الأولى لتأثير استغلال حوسبة المرئيات على بعض جوانب الحياة الاصطناعية مثل الروبوتات التطورية من خلال توفير سلسلة من تجارب جديدة في مجال عمل الإنسان الآلي. تهدف الروبوتات التطورية إلى إنشاء برامج السيطرة على التكيف من خلال استغلال المبادئ التطورية. تحقيق الهدف النهائي للإنسان الآلي التطوري يتطلب كميات هائلة من القدرة الحاسوبية، التي كانت حتى وقت قريب موردة أساسا من المعالجات المركزية.

قلوب وحدات المعالجة المركزية هي الأمثل لتنفيذ التعليمات البرمجية المتتابعة على حساب التنفيذ المتوازي، مما يجعلها غير فعالة نسبيا عندما يتعلق الأمر إلى تطبيقات الحوسبة عالية الأداء. الطلب المتزايد للرسوم ثلاثية الابعاد عالي الأداء في الوقت الحقيقي للسوق ساعد على تطوير معالج لغرافيك الى معالج الحوسبة المتوازية المتقدمة, متعددة المهام, متعددة القلوب, مع قوة حوسبة خارقة ومع عرض لنطاق الترددي العالي جدا للذاكرة.

كان لزيادة القوة والمرونة وانخفاض أسعار هذه الوحدات نتيجة غير مقصودة لاستخدامها في ميادين أخرى غير الغرافيك. يسمى هذا الاستخدام الحساب العام الغرض للبرمجة. بدافع من المتطلبات الحسابية الضخمة التي تتصل بمجالات البحوث التي تدخل في موضوعات الحياة الاصطناعية، نقترح في هذه الأطروحة تطبيق مفاهيم الحساب العام الغرض للبرمجة في هذا المجال. ويمكن الآن لهذه الموارد الحاسوبية الواسعة من وحدات المعالجة الحديثة استخدامها من قبل نماذج من الروبوتات التطورية لأنها تميل إلى أن تكون متوازية بطبيعتها.

وقد وضعت نماذج مختلفة تطورية مثيرة للاهتمام لمعالجة مسائل علمية هامة حول الكمبيوترمثل إجراءات الاكتساب وانشاء الروبوتات. على الرغم من أنها سمحت لنا أن نفهم بشكل أفضل الجوانب العلمية الهامة، إلا أن التعقيد والتنفيذ لم يتحسن بشكل جيد في السنوات الأخيرة. في كثير من الأحيان يلجا التقليل من المهام التجريبية وحجم هذه النماذج لتجنب التدريبات المفرطة الوقت التي تنمو باطراد مع عدد من بيانات التدريب.

تمثل خوارزميات التحسين مثل الخوارزميات التطورية طرق فعالة لحل المشاكل المعقدة في مجال العلوم والصناعة, على الرغم من أن هذه الاستدلالات تقلل كثيرا من الوقت على حساب استكشاف الفضاء بحثا عن حل , إلا أن تكلفة الأخيرة باهظة عندما يتم حل حالات كبيرة جدا من المشكلة. خاصية التوازي الكامن لخوارزميات التحسين تجعلها تبدو مناسبة تماما لتعمل على الأجهزة متوازية على نطاق واسع مثل وحدات معالجة لغرافيك.

في هذه الأطروحة، وضعنا هذه المعلومات حيز الاختبار عن طريق إجراء التجارب في نهج شامل مرتبط بالروبوتات التطورية. ونحن نحاول أن نعرف كيف يتم تكييف وحدات معالجة الغرافيك لهذه المهمة , مع تحديد أجزاء التقنية التطورية التي يجب أن تؤدى عليها.

تقدم هذه الأطروحة العديد من الحالات حيث أدى استغلال وحدات معالجة الصورة على خوارزميات الحياة الصناعية والروبوتات التطورية خصوصا في تطوير نماذج واسعة النطاق مع تعقيد لم يسبق لها مثيل لتحقيق خبرات جديدة.

الكلمات المفتاحية : الحساب العام الغرض للبرمجة بوحدات معالجة الغرافيك ،وحدات معالجة الغرافيك، الحياة الاصطناعية، الروبوتات، الخوارزميات التطورية بالتوازي، والشبكات العصبية المتكررة.

Résumé

Cette thèse constitue l'une des premières études sur l'impact de la programmation orientée GPU sur quelques aspects de la vie artificielle, telle que la robotique évolutionnaire. A cette occasion, nous fournissons une série d'expérimentations nouvelles dans le domaine de la conception de robots humanoïdes. La robotique évolutionnaire est un domaine qui s'intéresse à la création de programmes réalisant des contrôles adaptatifs pour des robots artificiels en exploitant les principes évolutionnistes. Atteindre l'objectif ultime de robots évolutionnaires viables nécessitera une puissance de calcul considérable, cette puissance étant jusqu'alors fournie principalement par des processeurs (CPUs) standards.

La demande sans cesse croissante du marché hautes performances pour les graphiques 3D temps réel a permis de faire subir aux architectures massivement parallèles des processeurs graphiques (GPUs) une évolution spectaculaire en matière de puissance de calcul graphique. Cette augmentation de la puissance et la flexibilité qu'elle dénote, conjuguées au faible coût de ces GPUs ont eu comme conséquence inattendue de voir leur utilisation s'étendre à des domaines étrangers à celui pour lequel ils ont été conçus, le graphisme. Par son extension, cet usage a reçu un nom, c'est le GPGPU (General Purpose computation on GPU). Motivés par les besoins de calcul considérables liés aux domaines de recherche s'inscrivant dans les thématiques de la vie artificielle, nous nous proposons dans cette thèse d'exploiter les concepts GPGPU à des applications spécifiques du domaine de la vie artificielle.

Différents modèles évolutionnaires intéressants ont été élaborés pour traiter des questions scientifiques importantes pour l'acquisition et la génération des actions de robots. Bien qu'il ne nous est pas aisément offert de mieux comprendre les aspects scientifiques les plus importants, la compréhension de leur complexité et la multiplication de leur utilisation ont fait émerger une évolution favorable au cours de ces dernières années.

En raison de leur parallélisme inhérent, les algorithmes évolutionnaires semblent bien adaptés pour être exécutés sur des architectures massivement parallèles telles que les GPUs.

Dans cette thèse, nous mettons cette assertion à l'épreuve en effectuant des expériences complètes pour une approche en rapport avec la robotique évolutionnaire. Cette thèse présente plusieurs cas où l'application du concept du GPU Computing sur des algorithmes de la vie artificielle et spécialement ceux de la robotique évolutionnaire a abouti à l'élaboration de modèles à grande échelle avec une complexité inédite permettant la réalisation de nouvelles expérimentations.

Mots Clés : GPU, GPGPU, Vie artificielle, Algorithmes évolutionnaires parallèles, Robotique évolutionnaire, Réseaux de Neurones Récurrents.

Abstract

This thesis is considered among the first studies of the impact of GPU computing on some aspects of artificial life as evolutionary robotics by providing a series of new experiments in the field of action of humanoid robots. Evolutionary robotics is concerned with the creation of adaptive control programs for robots by exploiting the evolutionary principles. Achieving the ultimate objective of evolutionary robots require huge amounts of computing power, which was until recently mainly supplied by standard CPU processors.

The increasing demand for high performance 3D graphics, real-time market evolved the graphics processor (GPU) into a highly parallel computing processor, multi-threaded, many-core, extraordinary computing power and with a very high bandwidth memory.

The increased power and flexibility and the low prices of these GPUs took the unintended consequence for their use in other areas than the graphics. This use is named GPGPU, General Purpose computation on GPU or GPU Generic Programming. Motivated by the huge computational requirements related to research areas falling within the artificial life themes, we propose in this thesis to apply the concepts of specific GPGPU applications in this field.

Various interesting evolutionary models have been developed to address the important scientific questions about the acquisition and generation actions robots computerized. Although they have allowed us to better understand important scientific aspects, complexity and implementation were not well improve in recent years. The experimental tasks and the scale of these models are often minimized to avoid excessive time trainings that grow exponentially with the number of the training data.

Because of their inherent parallelism, the evolutionary algorithms seem well suited to running on massively parallel hardware such as graphics processing units. In this thesis, we put this claim to the test by performing experiments in a comprehensive approach linked to the evolutionary robotics.

We try to know how the graphics processing units are adapted to the task and what parts of evolutionary technique that must be performed on them.

This thesis presents several cases where the application of GPU Computing on algorithms of artificial life and especially evolutionary robotics has led to the development of large-scale models with unprecedented complexity for the realization of new experiences.

Keywords : GPU, GPGPU, Artificial Life, Parallel Evolutionary Algorithms, Evolutionary Robotics, Recurrent Neural Network.

Publications et Communications

Les publications et les communications relatives à cette thèse sont les suivantes :

Publications

1. Benalia, N. H., Djedi, N., Bitam, S., Ouannes, N., Duthen, Y. (2015)(***in press***) 'An Improved CUDA-based Hybrid Metaheuristic for Fast Controller of an Evolutionary Robot', *Int. J. Embedded Systems*, Vol. x, No. x, pp.xxx–xxx.

Communications

1. Benalia, N. H., Djedi, N. (2012). Exploiting graphic cards potentialities for optimising phylogenies problem by using evolutionary algorithm. The 4th International Conference on Metaheuristics and Nature Inspired Computing, (META' 2012), Port El-Kantaoiui (Sousse, Tunisia), October 27-31, 2012.
2. Benalia, N. H., Ouannes, N., Djedi, N. (2014). Implementation of an improved parallel metaheuristic on GPU applied to humanoid robot simulation. In the International Conference on Multimedia Computing and Systems (ICMCS' 2014), pages 42-47. IEEE, 2014.
3. Benalia, N. H., Ouannes, N., Djedi, N. (2014). An Improved CUDA based Hybrid Metaheuristic for Fast Controller of an Evolutionary Robot. International Conference on Information Technology for Organization Development(IT4OD' 2014). IEEE, 2014 (**Best paper Award**).
4. Benalia, N. H., Djedi, N., Ouannes, N., Duthen, Y. (2014). GPU based Metaheuristic for Fast Controller of an Evolutionary Robot. The 5th International Conference on Metaheuristics and Nature Inspired Computing, (META'2014), Marrakech, Morroco, October 27 - 31, 2014.
5. Benalia, N. H., Ouannes, N., Djedi, N. (2014). High Performance Bio-inspired Techniques for Evolutionary Controller −Case Study of Evolutionary Robot using GPGPU Algorithms−, 1ère Conférence sur l'Ingénierie Informatique (C2i' 2014)à l'école militaire, Alger, 16-17 Décembre 2014.

Workshops

1. Benalia, N. H., Djedi, N. (2013). Execution Configuration Optimisations for phylogenies problem on the Fermi architecture. Le Premier workshop Images, Graphiques et Vie Artificielle, 08,09-10 Juin, BISKRA, 2013.
2. Benalia, N. H., Ouannes, N., Djedi, N., Duthen, Y. (2014). Optimization Considerations for GPU Based Metaheuristic Approach Applied to an Evolutionary Robot. Le Deuxiéme workshop Images, Graphiques et Vie Artificielle, 16,17-18 Juin, BISKRA, 2014.

Table des matières

Table des figures

Liste des tableaux

Chapitre 1

Introduction Générale

1.1 Cadre de la Thèse

Le parallélisme est un concept de plus en plus populaire qui contribue grandement à satisfaire la demande croissante en performance dans le domaine informatique. Depuis quelques années, on assiste à une démocratisation à grande échelle d'ordinateurs possédant des potentialités de calculs parallèles du fait qu'ils soient devenus de plus en plus abordables. De plus, un certain volume d'efforts a été consenti dans le but de les rendre encore et en même temps plus accessibles et davantage plus conviviaux. Cette croissance rapide de la technologie des architectures parallèles a touché beaucoup de domaines parmi lesquels ceux liés aux problèmes de la vie artificielle.

L'efficacité des algorithmes évolutionnaires à traiter divers problèmes a été démontrée dans plusieurs travaux de recherche fondamentale et appliquée. Le succès de ces méthodes s'explique par un ensemble de facteurs, notamment par leur potentiel d'adaptation aux différentes contraintes associées à des problèmes spécifiques, par leur facilité d'implantation dans des programmes d'application divers et par la qualité des solutions qu'elles peuvent produire en un temps relativement court. Le domaine des algorithmes évolutionnaires est toutefois encore jeune et de nombreux efforts sont présentement mis en œuvre dans le but d'améliorer la performance de ces méthodes de résolution.

Cette thèse représente une tentative dans la voie d'explorer les implications de programmation sur les GPUs pour la résolution de problèmes du domaine de la vie artificielle tels que la robotique évolutionnaire, en procédant à l'accélération de techniques évolutionnistes utilisées. Le domaine de la vie artificielle est associé au développement de comportements de créatures artificielles inspirées de la vie naturelle. Le développement de tels comportements dépend fortement de la nature de l'environnement de simulation et des ressources dont il dispose. Dans ce contexte, un nombre important d'algorithmes traitant ce type de problèmes se sont avérés idéalement parfaits pour le modèle de programmation parallèle du GPU, et en conséquence que leur accélération peut ouvrir de nouvelles possibilités dans la conception de solutions aux problèmes auxquels s'intéresse la vie artificielle. En outre, l'utilisation des GPUs pour des problèmes de la vie artificielle telle que la robotique évolutionnaire ne permet pas uniquement d'accélérer les modèles existants, mais offre également la possibilité d'ouvrir de nouveaux horizons pour des expérimentations qui n'ont pas été tentées auparavant en raison du volume de calcul intensif qu'elles engendrent.

1.2 Motivations

1.2.1 Émergence des architectures multicœurs

Dans les processeurs multicœurs, l'utilisateur peut désormais adresser et contrôler plusieurs threads en parallèle, comme ce fut le cas dans les anciens systèmes multiprocesseurs utilisés dans les environnements professionnels (les serveurs multi-utilisateurs, les serveurs web, les clusters de haute performance (HPC), etc.). Le nombre de cœurs a par la suite évolué pour atteindre le nombre de 4. En utilisant le Multithreading simultané (SMT), 8 threads peuvent être exécutés en parallèle et en concurrence. L'inconvénient de ces architectures réside dans le fait que pour atteindre sa puissance de crête de calcul, de tels processeurs doivent effectuer plusieurs tâches simultanément.

L'exploitation de ces ressources est assurée par le *Scheduler* des tâches, qui permet d'assigner une tâche pour chacun des différents cœurs. Dans les systèmes modernes, le Scheduler distribue les tâches d'un utilisateur sur tous les cœurs constituant le CPU (navigateur Web, le programme d'écoute de la musique, des démons système).

Cette technique présente plusieurs avantages, et offre une forte réactivité du système d'interaction de l'utilisateur ainsi qu'une exploitation optimale des concepts matériels et logiciels déjà existants. En effet, la planification des tâches standards multiprocesseur peut être adaptée à ce cas.

Cependant, certaines questions restent ouvertes : l'équilibrage de charge, en particulier, est un problème difficile. En raison de l'aspect mono-utilisateur, le nombre de tâches lourdes n'est pas forcément assez élevé. L'utilisateur exécute une tâche principale, qui est en général la plus consommatrice en terme de temps CPU.

Une tâche aussi large utilise toute la puissance de traitement disponible uniquement si elle est programmée pour être exécutée sur un système multiprocesseur. Mais dans le cas général, cette exigence n'est pas observée, car la plupart des applications sont toujours développées de façon séquentielle. Afin de surmonter ce problème, certains mécanismes ont été mis en œuvre afin d'augmenter la puissance d'un des cœurs au détriment des autres.

Les architectures multicœurs apportent un avantage réel dans les clusters de calcul, où le parallélisme entre les nœuds de calcul, entre les multiples processeurs et entre les noyaux sont exploités. Si les deux derniers niveaux de parallélisme sont juste un peu différents, des clusters d'ordinateurs avec un processeur unique sont aujourd'hui assez rares et chaque application à l'aide de telles architectures de Clustering doit utiliser des paradigmes de parallélisation différents, un pour les machines à mémoire partagée et un autre pour les machines à mémoire distribuée (par exemple, OpenMP, MPI).

1.2.2 Émergence du calcul général GPGPU sur les architectures GPUs

Les processeurs graphiques (GPUs) ont été introduits par NVIDIA dès 1999. La demande toujours croissante pour des graphiques hautes performances et temps réel découlant principalement de l'industrie du jeu ont fait subir au GPU une évolution permettant de le classer en tant que supercalculateur massivement parallèle, possédant des performances de calcul extrêmement élevées en virgule flottante. Peu de chercheurs ont réalisé que cette performance cachée pourrait révolutionner de nombreux algorithmes scienti-

fiques. Le problème auquel se sont confrontés les développeurs d'applications sur GPU résidait dans le fait que ces instruments n'ont pas été conçus pour le calcul généraliste et donc ne peuvent pas être programmables de la même manière qu'on le fait avec les CPUs. Ainsi, le GPU constituait un moyen de réaliser uniquement des graphismes en manipulant des Shaders, des textures, des points, des pixels, etc.

Pour écrire un nouveau code qui pourrait fonctionner sur GPU, ces programmeurs devaient représenter leurs problèmes mathématiques comme des textures et les transmettre vers le GPU comme entrée au GPU en utilisant les APIs graphiques tels qu'OpenGL ou DirectX. Le GPU serait alors capable de produire des sorties sous forme de pixels et d'attributs graphiques, qui devaient être décodés pour être utilisés en tant que résultat. L'utilisation de cette technique a montré à ceux qui l'ont adoptée qu'elle pouvait arriver à des accélérations beaucoup plus compétitives que les CPUs classiques. Cependant, l'ensemble du processus de programmation GPU était extrêmement lourd et exigeait des connaissances supplémentaires sur le fonctionnement des APIs graphiques, ce qui a découragé énormément l'utilisation des processeurs graphiques pour des fins de calculs généralistes.

Le 15 février 2007, représente la date de publication de la première plateforme de calcul parallèle nommée CUDA (Compute Unified Device Architecture). Cette plateforme a offert aux développeurs la possibilité de disposer d'un accès direct à l'ensemble des instructions virtuelles ainsi qu'à la mémoire des éléments de calcul parallèles dans le GPU. Grâce à la plateforme CUDA, le GPU est devenu un supercalculateur parallèle à faible coût et aux performances inégalées. C'est à partir de cette date que des chercheurs d'autres domaines se sont intéressés à la programmation GPGPU pour accélérer des applications gourmandes et exigeantes en termes de puissance de calcul.

1.2.3 Programmation GPGPU pour les méthodes bio-inspirées, les phénomènes biologiques, et les techniques évolutionnistes

Les méthodes inspirées de la nature sont privilégiées pour leur robustesse et leur très bonne exploration de l'espace de recherche. Ces méthodes s'intéressent principalement aux algorithmes évolutionnaires, qui touchent de près aux algorithmes génétiques, aux stratégies d'évolution, à la programmation génétique, à l'optimisation évolutionnaire, à l'optimisation par colonies de fourmis ainsi qu'aux approches émergentes (BOIDS, optimisation par essaim particulaire).

De par le passé, l'utilisation des algorithmes bio-inspirés a été relativement limitée du fait de la nécessité de disposer de ressources informatiques très importantes. La progression, qualifiée de spectaculaire, de la puissance de calcul des ordinateurs, a permis à des modèles réels inspirés de phénomènes ou de comportements naturels observés dans la nature d'être appliqués pour la résolution de problèmes complexes. Plusieurs exemples peuvent être cités, l'évolution naturelle, les cellules nerveuses, les essaims d'insectes, les populations de plantes ou d'animaux et divers anticorps. Dans tous ces cas, typiquement, chaque agent est plus ou moins indépendant, et dans une certaine mesure, les actions individuelles de ces agents, réalisées naturellement simultanément, font émerger un comportement, très souvent plus complexe que la somme de toutes ces actions. La simulation de tels comportements est souvent réalisée séquentiellement sur un seul ordinateur, générant par conséquent des temps de calcul colossaux dus souvent et selon le cas, au nombre

d'individus dans les populations simulées, à la taille des réseaux de neurones, à la taille des essaims, etc.

Cependant, dans la plupart des cas, les simulations peuvent être réalisées facilement en parallèle (plutôt que séquentiellement) mais les coûts qu'une telle entreprise peut générer, rend des actions de ce type très souvent irréalisables. Tout récemment, on observe un regain d'intérêt à des efforts de parallélisation en utilisant les architectures graphiques dont disposent les GPUs qui offrent des opportunités de parallélisation et d'accélération à faible coût.

Certaines des premières utilisations des GPUs concernent les algorithmes évolutionnaires (AEs). Les AEs sont inspirés de l'évolution naturelle. Ils permettent de trouver des solutions qui ne sont pas forcément optimales, mais très souvent satisfaisantes à de nombreux problèmes inverses complexes, mais nécessitent très souvent des puissances de calcul très élevées.

Ceci rend les algorithmes inspirés de la nature en général et les AEs, en particulier, intéressants dans le contexte de la programmation GPGPU, du fait de leur nature intrinsèquement parallèle. En effet, ces algorithmes réalisent une exploration/exploitation de l'espace de recherche par l'évolution d'un ensemble de solutions possibles. En réalité, un nombre restreint des étapes qui les constituent nécessitent une communication globale. Comme ceci représente le même cas dans de nombreux domaines de calcul intensif, de nombreux travaux ont été entrepris et d'autres sont en cours, dans le but d'aboutir à leur parallélisation sur diverses architectures, et à fortiori sur des architectures GPUs. Ce constat est principalement dû au fait que ces algorithmes sont de nature intrinsèquement parallèle, et peuvent utiliser le nombre très élevé de cœurs de traitement disponibles dans les GPUs modernes.

Finalement, le caractère SIMD du parallélisme GPGPU ne représente pas une limitation pour les AEs et leur aspect synchrone les rend de bons candidats pour une utilisation efficace des GPUs.

1.3 Problématique, Objectifs, et Contributions

L'utilisation des cartes graphiques massivement parallèles (GPUs) pour le calcul scientifique est aujourd'hui, une réalité inéluctable, de par la puissance que les GPUs incarnent et les coûts très loin et bien en deçà des solutions massivement parallèles classiques. Ainsi, dans le domaine de l'évolution artificielle et des algorithmes évolutionnaires, cette éventualité constitue un challenge de par la puissance de calcul dont on peut disposer aujourd'hui dans les cartes graphiques.

Pour toutes ces raisons, de plus en plus de problèmes, jadis inabordables, tant ils nécessitaient des ressources informatiques indisponibles en termes de facteurs puissance/coût, trouvent aujourd'hui espoir dans l'exploitation des possibilités des cartes graphiques dans la résolution de problèmes, au demeurant non graphiques, et ce, en exploitant les architectures massivement parallèles du pipeline des cartes graphiques.

En terme général, l'objectif principal de la recherche entreprise dans cette thèse est d'étudier l'impact de la programmation GPU dans la résolution des problèmes de la robotique évolutionnaire et de mener de nouvelles expérimentations pour prendre en charge la complexité inhérente à la robotique évolutionnaire. La présente entreprise représente l'une des premières thèses à explorer cette problématique.

La robotique évolutionnaire (RE) est un axe relativement nouveau dans le domaine de l'intelligence artificielle pour laquelle l'idée principale est de créer des programmes de contrôle adaptatif pour des robots autonomes en exploitant les principes évolutionnistes. Les robots sont considérés comme des organismes artificiels qui développent leurs propres systèmes de contrôle voire leurs morphologies en étroite interaction avec l'environnement et sans intervention humaine. La RE s'inspire des mécanismes de l'auto-organisation biologique et intègre les aspects liés à l'évolution, au développement, aux systèmes neuronaux et aux développements morphologiques.

Le problème majeur dans un tel domaine réside dans le processus et les mécanismes permettant la conception de contrôleurs pouvant évoluer en un temps raisonnable.

Cependant, l'exploitation des modèles parallèles n'est pas un issue facile, et plusieurs questions sont liées à une exploitation efficace de la hiérarchie mémoire de l'architecture des GPUs, à la distribution des données à traiter entre le CPU et le GPU, à l'optimisation du transfert de données entre les différents mémoires ainsi qu'aux capacités de la mémoire. Les objectifs généraux de ce travail de recherche obéissent à deux axes :

Le premier étant d'étudier les relations entre le calcul parallèle et les algorithmes évolutionnaires afin de mieux comprendre les bénéfices que peuvent apporter les algorithmes évolutionnaires parallèles dans la résolution de problèmes de la vie artificielle.

Le deuxième, quant à lui, consiste à identifier et mettre en œuvre des solutions parallèles performantes à un problème lié à la robotique évolutionnaire pratique. L'aboutissement de ces objectifs passe alors par une connaissance adéquate du parallélisme, des architectures GPUs, des algorithmes évolutionnaires et des problèmes de la vie artificielle telle que la robotique évolutionnaire.

Cette étude est considérée parmi les premières investigations sur l'impact du GPU Computing dans la robotique évolutionnaire. La contribution principale de cette thèse est de faire face aux challenges de l'évolution parallèle dans un problème de robotique évolutionnaire de large instance sur l'architecture de GPU. Notre objectif est la reconception des modèles existants pour optimiser et réussir leur déploiement sur GPU. Pour atteindre cet objectif, nous proposons dans ce document une vision inédite et une conception nouvelle pour la construction d'un contrôleur de robotique évolutionnaire sur une architecture GPU. Notre challenge est de sortir avec une conception à base de GPU adéquate pour toute la hiérarchie des modèles parallèles. Différentes contributions avec des issues principales sont traitées dans ce document :

1. Cette étude a permis de proposer un système efficace de coopération entre la partie de simulation évolutionnaire dans le CPU et celle dans le GPU, qui exige l'optimisation du transfert de données entre le CPU et le GPU.

2. Cette étude nous a conduits de proposer un système de contrôle de parallélisme efficace qui permet d'associer les unités de calcul avec les processus élémentaires de la tâche globale, en considérant les différentes parties de la simulation.

3. Cette étude a permis de proposer un mappage efficace des différentes structures de la simulation sur la hiérarchie de la mémoire.

1.4 Structure du manuscrit

Dans cette section, nous proposons un résumé des différents chapitres composant ce manuscrit tout en précisant les principaux résultats obtenus.

Le deuxième chapitre décrit la théorie et les concepts généraux des algorithmes évolutionnaires, ainsi que leur complexité et leurs méthodes d'analyses.

Le troisiéme chapitre commence par présenter les concepts du parallélisme, une classification des architectures parallèles ainsi que les modèles et les langages de programmation qui leur sont associés. Par la suite, nous décrivons les modèles parallèles des algorithmes évolutionnaires ainsi que les paradigmes informatiques considérés comme des outils permettant leur parallélisation. Par ailleurs, une étude approfondie concernant l'architecture des GPUs, leur émergence, leurs caractéristiques matérielles et logicielles, en mentionnant leur utilisation au sein des méthodes bio-inspirées. Une discussion des travaux existants concernant la parallélisation des algorithmes évolutionnaires dans les architectures classiques et modernes sur les GPU spécialement, est également abordée à la fin de ce chapitre.

Le quatrième chapitre présente une analyse des sources de parallélisation à prendre en considération dans le problème étudié, puis il s'intéresse principalement à la présentation de notre proposition qui s'exprime dans le domaine de la robotique évolutionnaire en étudiant les différentes classes et parties du problème traité, la partie simulation dans le moteur physique, et la partie évolutionniste avec leur re-conception sur GPU. La gestion de la hiérarchie de mémoire ainsi que la distribution des tâches entre le CPU et le GPU pour le processus de simulation est également présentée en fin de chapitre.

Le cinquième chapitre reprend les détails des sous-systèmes déjà décrits dans le chapitre précédent. Celui de la distribution des tâches entre le CPU et le GPU ainsi que celui lié à la gestion mémoire en ont reçu une attention particulière.

Enfin, le document est achevé avec une conclusion qui permet de faire un récapitulatif de toutes les études effectivement entreprises dans cette thèse et suivi par la présentation d'une future vision sur le rôle et la contribution des architectures GPUs dans le domaine évolutionnaire.

Chapitre 2

Intelligence artificielle bio-inspirée : Algorithmes évolutionnaires, Notions, Complexité et Analyse

2.1 Introduction

L'intelligence artificielle (IA) bio-inspirée [50] tente à produire synthétiquement des systèmes qui présentent de l'intelligence en s'inspirant des processus des systèmes naturels de la vie. Des exemples de l'intelligence artificielle bio-inspirée incluent les comportements reproduits en robotique, les réseaux de neurones artificiels (RNAs), les algorithmes évolutionnaires (AEs), l'optimisation par essaim de particules et l'optimisation par colonies de fourmis.

La biologie est une branche de la science qui cherche à déterminer les lois de la nature inhérentes à la structure et au comportement des organismes vivants. Le biologiste postulera une théorie ou un modèle pour expliquer certains phénomènes naturels, et vérifiera ensuite expérimentalement à quel point le modèle prédit ce qui est observé dans la nature. Bien qu'il n'existe pas de contraintes de conception dans le domaine de l'IA pour se limiter à appliquer les mêmes mécanismes à ceux utilisés par les systèmes de vie naturels, la « fausseté » d'un modèle particulier à reproduire l'intelligence naturelle peut être un guide utile pour mesurer le succès d'un modèle d'IA bio-inspirée. En outre, on tente de reproduire les capacités d'intelligence du niveau humain qui pourraient conduire potentiellement à de nouvelles percées technologiques. Un des buts de ce chapitre est de souligner l'importance de ce processus de validation par rapport à des systèmes du monde réel dans l'évaluation des modèles bio-inspirés de l'intelligence artificielle. Nous pouvons acquérir des connaissances en les déduisant de comparaisons avec les observations des processus naturels afin de munir la conception de nos systèmes d'une vie artificielle intelligente [50], [196].

La plupart des modèles développés jusqu'à présent sont faiblement liés à leur équivalent de la vie réelle (tels que les réseaux de neurones artificiels et les stratégies évolutionnaires comme sont les exemples les plus évidents). De nombreux systèmes bio-inspirés de l'IA ont négligé l'importance de maintenir la comparaison avec le monde réel. Une fois une technique jugée efficace, l'approche diverge souvent de son équivalent du monde réel, et les améliorations de la méthode se concentrent souvent davantage sur l'amélioration des

performances de la machine et moins sur la façon de tenir compte des phénomènes naturels mieux observés afin d'améliorer le réalisme. Il est souvent important de revoir les objectifs de conception de la recherche originale en tirant plus d'inspiration à partir des systèmes de la vie réelle. Ceci peut être fait en examinant les phénomènes qui ne sont pas couverts par le modèle en explorant des solutions alternatives, ou en cherchant à étendre le modèle pour en améliorer le réalisme et analyser la partie fausse du modèle [196].

Dans cette optique, une des directions les plus explorées actuellement est liée aux améliorations avancées par les AEs existants, qui prennent leur inspiration du processus évolutif naturel. Il est, de ce fait intéressant de se concentrer sur les améliorations significatives de performance qui peuvent être faites en apportant des modifications inspirées des mécanismes biologiques non inclus dans le modèle original ou en apportant des modifications pour les adapter aux nouvelles technologies existantes telles que les architectures parallèles récentes. Ainsi, le point d'intérêt de cette thèse est l'évolution artificielle en utilisant les AEs ainsi que leur redéfinition pour les nouvelles technologies parallèles. Ce chapitre est consacré à la présentation de la théorie des AEs, l'analyse de leur complexité, et surtout les problèmes nécessitant de grands espaces de recherche, suivie par la présentation des méthodes d'analyse existantes ainsi que leurs sources de performance [89], [144], [193], [90].

Les AEs ont été utilisés, souvent avec succès, dans de nombreux domaines informatiques tels que l'optimisation, l'apprentissage, l'adaptation et autres. Malgré la réussite de plusieurs applications avec les AEs, la théorie des AEs est encore à ses commencements. De plus, l'idée fondamentale d'un AE est l'obtention d'un problème robuste indépendant de l'heuristique de recherche avec un comportement efficace de plusieurs problèmes à partir d'une large variété de problèmes. Cette variété de problèmes rend l'analyse des AEs plus difficile que l'analyse d'un problème d'algorithmique spécifique. Néanmoins, le progrès dans la conception et dans l'application des AEs gagnerait considérablement en exploitant davantage des fondements théoriques. Actuellement, nous sommes capables d'analyser les AEs sans croisement et les AEs avec croisement sur quelques fonctions. Les fonctions ne sont pas des exemples à partir d'applications du monde réel, mais des exemples de fonctions décrivant quelques issues typiques de fonctions ou sont choisies pour voir les comportements extrêmes des AEs [193], [41], [40], [94].

Ce chapitre s'intéresse aussi à étudier quelques méthodes d'analyses des AEs existants ainsi que leur complexité et leurs performances. Ainsi, l'objectif principal consiste à mettre l'accent sur le temps prévu dans leur exécution et la probabilité de réussite avec une borne de temps raisonnable [183], [91].

2.2 Évolution naturelle : principe des algorithmes évolutionnaires

L'évolution naturelle est un élément fondamental dans la biologie moderne, et les scientifiques s'entendent remarquablement sur ce sujet.

Les théories originales de l'évolution et de la sélection naturelle ont été proposées presque simultanément et indépendamment par *Charles Robert Darwin* et *Alfred Russel Wallace* au 19ème siècle, combinée avec le sélectionnisme par *Charles Weismann* et de la génétique par *Gregor Mendel*, elles sont acceptées de façon ubiquitaire dans la communauté scientifique, et sont répandues chez le public en général [164].

Ce corpus cohérent, souvent appelé le néo-darwinisme, agit comme une grande théorie unificatrice pour la biologie : il est en mesure d'expliquer les merveilles de la vie, et, plus remarquablement, il le fait à partir d'un nombre limité de concepts relativement simples et intuitivement plausible. Il décrit l'ensemble du processus de l'évolution par des notions telles que la reproduction, la modification, la concurrence, et la sélection.

L'évolution peut être facilement décrite comme une séquence d'étapes, certaines plus souvent déterministes et certaines plus souvent aléatoires. Une telle idée de forces aléatoires est formée par des pressions déterministes inspirantes et n'est pas surprenante, elle a été exploitée pour décrire des phénomènes tout à fait sans rapport avec la biologie. Des exemples notables incluent des alternatives conçues lors de l'apprentissage, les idées s'efforcent de survivre dans notre culture [136], ou même les univers possibles.

L'évolution peut être considérée comme un processus d'amélioration qui tente de parfaire des caractéristiques inexpérimentées. L'ensemble du paradigme néo-darwiniste peut donc être considéré comme un outil d'optimisation puissant capable de produire d'excellents résultats à partir de zéro, ne nécessitant pas un plan, et avec l'exploitation d'un mélange d'opérateurs aléatoires et déterministes [90].

Dans le calcul de l'évolution, une solution unique candidate est appelée individu ; l'ensemble des solutions candidates qui existent à un moment donné est appelé population, et chaque étape du processus d'évolution est appelée génération. La capacité d'un individu à résoudre un problème donné est mesuré par une fonction de fitness [194], qui classe la probabilité d'une solution et propage ses caractéristiques pour les générations futures.

Le mot *génome* désigne l'ensemble du matériel génétique de l'organisme, bien que sa mise en oeuvre réelle soit différente d'une approche à l'autre. Le *gène* est l'unité fonctionnelle de l'héritage, ou opératoire, elle représente le plus petit fragment du génome qui peut être modifié pendant le processus d'évolution. Les *gènes* sont positionnés dans le génome à des positions spécifiques appelées *loci*, pluriel de locus. Les gènes alternatifs qui peuvent se produire à un locus donné sont appelés *allèles* [107].

Les processus naturels qui conduisent à des mutations, la reproduction, la concurrence et la sélection sont émulées par des opérateurs. Les opérateurs agissent sur les gènes, les individus seuls, des groupes ou des populations entières, et produisent généralement une version modifiée de l'entité qu'ils manipulent. Les biologistes ont besoin de faire la distinction entre le *génotype* et le *phénotype* : le premier est l'ensemble de la constitution génétique d'un organisme, le deuxième présentant l'ensemble des propriétés observables qui sont produites par l'interaction entre le génotype et l'environnement. Dans de nombreuses implémentations, les praticiens du calcul évolutif n'en font pas une distinction précise. La valeur numérique qui représente l'aptitude d'un individu est parfois assimilée à son phénotype [164], [107].

Pour générer les descendants pour la prochaine génération, la plupart des AEs mettent en oeuvre la reproduction sexuée et asexuée. La première est généralement appelée recombinaison ; elle nécessite deux participants ou plus, et implique la possibilité pour les descendants d'hériter des caractéristiques différentes de parents différents. Lorsque la recombinaison est réalisée par un simple échange de matériel génétique entre les parents, elle prend souvent le nom de croisement. La deuxième est nommée réplication, pour indiquer que la copie d'un individu est créée, ou, plus couramment, de mutation, pour souligner que la copie n'est pas exacte. Presque aucun AE ne prend en compte le genre, d'où les individus n'ont pas des rôles distincts de reproduction. Dans certaines implémentations, la mutation a lieu seulement après la recombinaison sexuelle. On remarque que certains AEs

ne stockent pas une collection d'individus distincts, et donc la reproduction est effectuée en modifiant les paramètres statistiques qui décrivent la population actuelle. Tous les opérateurs exploités lors de la reproduction peuvent être cumulativement appelés opérateurs évolutionnaires, ou opérateurs génétiques soulignant qu'ils agissent au niveau génotypique [44].

2.2.1 Qu'est-ce qu'un algorithme évolutionnaire (AE) ?

De grandes inventions ont été le résultat de la bionique [44], à savoir l'application des principes biologiques ou naturels pour l'étude et la conception des systèmes inspirés des humains. Nous avons imité les chauves-souris pour inventer le radar, les poissons pour inventer les sous-marins, etc. L'évolution naturelle des espèces pourrait être regardée comme un processus pour apprendre à s'adapter à l'environnement et l'optimisation de l'aptitude de l'espèce. Ainsi, on pourrait imiter le point de vue de la génétique moderne, à savoir le principe de "*la survie du plus apte*", dans la conception, l'optimisation ou l'apprentissage des algorithmes.

Comme son nom le suggère, un AE a un lien avec la biologie. D'un point de vue biologique, l'idée est de formaliser le modèle de l'évolution qu'on a à l'esprit, et de synthétiser cette idée à travers un algorithme [89]. Les AEs appartiennent à la famille des méta-heuristiques, qui comprend aussi, par exemple, les essaims de particules [43] et l'optimisation par colonies de fourmis [38], qui sont inspirés à partir d'autres structures et processus biologiques, ainsi que des méthodes classiques comme le recuit simulé [189], qui est inspiré d'un processus de la thermodynamique.

Comme il est connu, cette grande catégorie d'algorithmes est subdivisée en quatre grandes familles comme mentionnées dans la table 2.1. Les détails exacts de chaque famille diffèrent sur des points considérables, et ne sont pas équivalents en général. Chaque membre de la famille des AEs a un grand nombre de variantes décrites dans la littérature et en tant que tel, la généralisation ci-dessous devrait être acceptée avec soin [159].

Cette catégorie d'algorithmes est considérée généralement comme des solveurs généraux de problèmes. Ceci implique qu'ils sont conçus selon une idée générale concernant la manière de la mise en oeuvre. Dans le cas des AEs, cette idée provient d'une compréhension de l'évolution naturelle. Le plus important encore, est qu'ils ne sont pas conçus de manière adaptée vers un type spécifique de problème d'optimisation. La façon de faire l'optimisation tout en étant inconscient de l'instance du problème concret est appelée "*optimisation à boite noire*" [90].

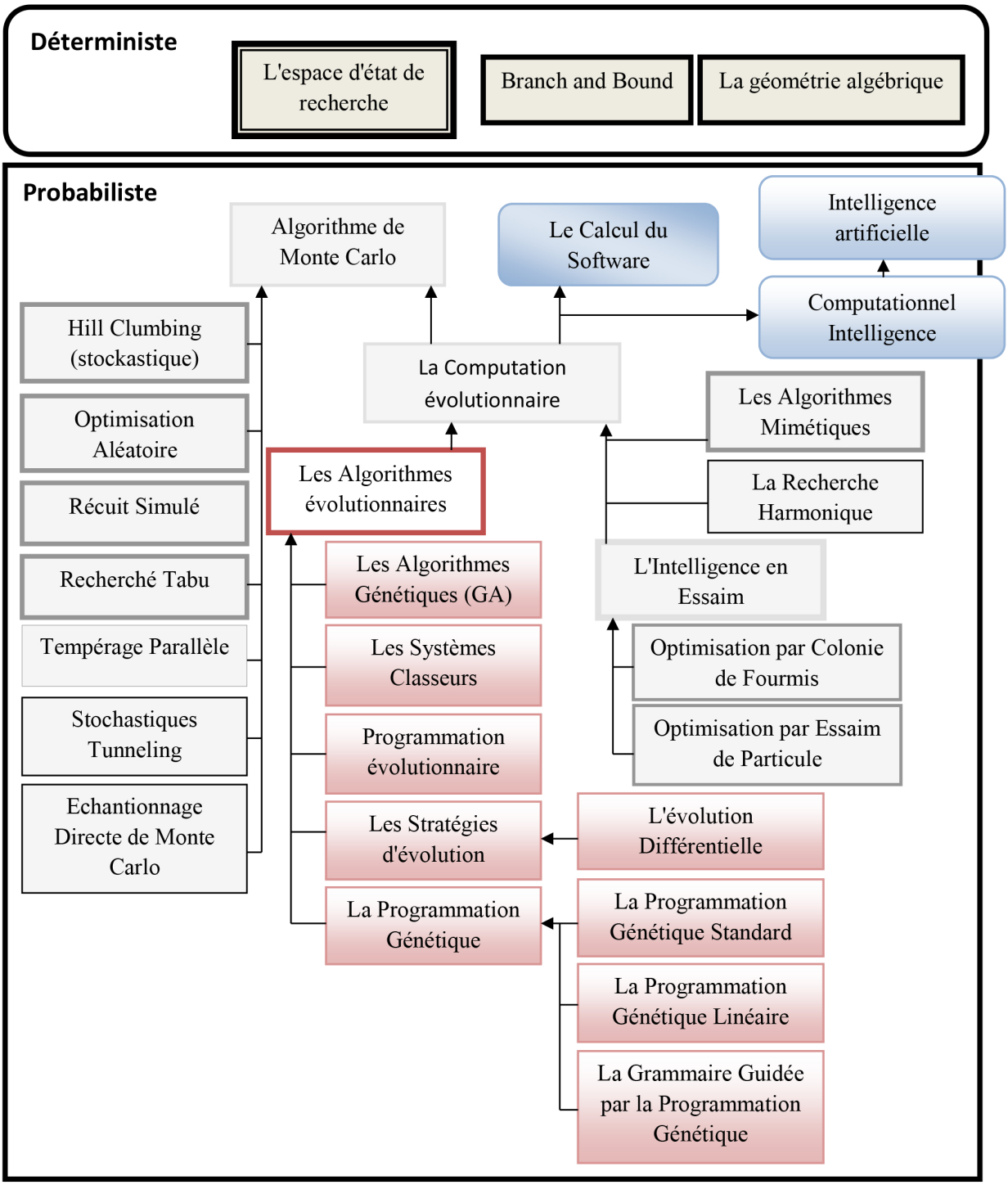


FIGURE 2.1 – Position des algorithmes évolutionnaires dans l'espace des méthodes de recherche aléatoires [195].

Le schéma suivant représente la chronologie de quelques AEs :

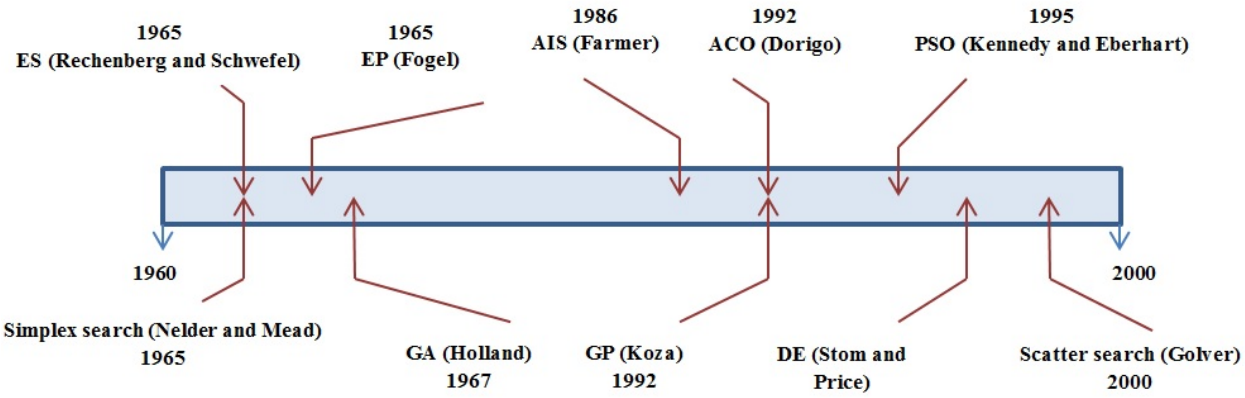


FIGURE 2.2 – Chronologie de quelques différents AEs.

La variante	La description
Les Stratégies d'évolution (SE)	Introduites par Ingo Rechenberg en 1960, elles sont destinés à optimiser les systèmes de valeurs réelles.
Les Algorithmes Génétiques (AG)	Introduits par John Holland en 1960, ils sont destinés a optimisé les systèmes avec des représentations binaire et réelle.
La Programmation Evolutionnaire (PE)	Introduite par Lawrence Fogel en 1960, conçue pour évoluer des machines d'état fini.
La Programmation Génétique (PG)	Introduite par John Koza en 1960, conçue pour faire évoluer les arbres Parse de langage informatiques fonctionnels tels que LISP.

TABLE 2.1 – Variantes des algorithmes évolutionnaires [159].

Les AEs sont décomposés en plusieurs parties interdépendantes : *La Population, la Fonction de Fitness (la fonction d'évaluation, la fonction objective)* et, *le Moteur de Variation*. Par la suite, nous introduisons les notions de base et les concepts des AEs en les transférant de leurs homologues biologiques, voir la table (2.2).

2.2.1.1 Population

Les AEs maintiennent un groupe d'individus (solutions), appelé population, afin d'optimiser ou de réaliser un apprentissage sur le problème, et ce, d'une manière parallèle.

a. Individu :

Un individu, qui est un organisme vivant en biologie, correspond à une solution candidate en informatique. Dans les deux domaines, un individu est décrit par un chromosome.

En biologie, un chromosome est constitué d'acide désoxyribonucléique (ADN) et de nombreuses protéines histones, alors que dans la science informatique, l'information génétique est codée comme une séquence d'objets informatiques tels que : les bits, les caractères, les nombres, etc. La plupart des organismes vivants possèdent plusieurs chromosomes, à titre d'exemple, les humains ont 46 chromosomes, organisés en 23 paires homologues. Cependant, en informatique, cette complication est ignorée et toutes les informations génétiques sont combinées dans un seul chromosome. Chaque individu représente une solution potentielle du problème à optimiser. Il est encodé par un ensemble de gènes qui le décrivent [107]. Ces gènes constituent la représentation génotypique de l'individu, qui est transformée en une représentation phénotypique pour l'évaluation.

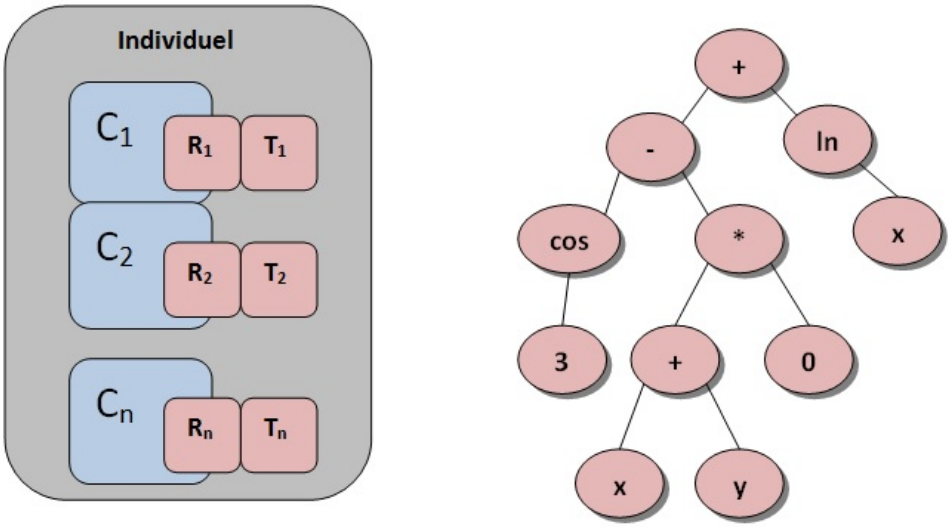


FIGURE 2.3 – Représentation de deux individus pour deux problèmes d'algorithmes évolutionnaires : gauche, (a) Représentation du génome pour les examens horaires. Droite, (b) Une représentation d'un individu pour la PG [124].

b. Population :

Une population est un simple ensemble d'organismes, généralement de la même espèce. À cause de la complexité des génomes biologiques, il est généralement prudent de supposer qu'il n'existe pas deux individus d'une population partageant la même configuration génétique - les jumeaux homozygotes étant la seule exception. En outre, même les individus génétiquement identiques, différent à cause des caractères acquis, (ne sont jamais parfaitement identiques), ce qui conduit à des phénotypes différents. Cependant, en informatique, il faut prévoir la possibilité d'avoir des individus identiques à cause de la variabilité limitée d'un chromosome. En conséquence, une population d'un AE est un sac ou un multi-ensemble d'individus [107].

c. Générations :

Dans l'évolution biologique et simulée, la production se réfère à la population sur

certains repères dans le temps. Une nouvelle génération est créée par la reproduction, et ceci par la génération des descendants d'un ou plusieurs organismes, dans lesquels le matériel génétique des individus parents peut être recombiné. La même procédure est utilisée en informatique, à la seule différence que le processus de création d'un descendant agit directement sur les chromosomes et que le nombre de parents peut excéder deux. La reproduction sera entamée avec plus de détails dans les opérateurs de variation [159], [124].

d. Initialisation de la population :

Concernant l'initialisation de la population, un générateur aléatoire uniforme est utilisé pour initialiser les différentes valeurs des gènes de la population initiale et permet d'échantillonner uniformément l'espace de recherche à grande échelle.

Des stratégies plus sophistiquées peuvent être utilisées, et des connaissances d'experts peuvent être insérées à ce niveau. Bien sûr, cela va induire un biais, qui pourrait empêcher l'algorithme de trouver une bonne solution si la population est initialisée dans une mauvaise partie de l'espace de recherche [124].

Le choix initiale et crucial pour un pratiquant d'un AE est la taille de la population. Le raisonnement de base explique que la population doit être choisie en fonction de la longueur de la chaine du génome. Une population trop grande conduit à une perte de temps de calcul avec plusieurs évaluations indifférenciées de la fonction. Alors qu'une population de petite taille peut conduire une couverture insuffisante de l'espace de recherche [159].

Plusieurs travaux ont entamé l'analyse de cette question dans une série d'investigations tels que ceux réalisés par Goldberg et ses co-auteurs [64], [62], [63], [78], Reeves [157] et Gao [56]. Comme résultat, il suffit de dire qu'une large population $p \gg n$, n'est généralement pas nécessaire et que la diversité est souhaitable, et de plus elle doit être maintenue. Cette hypothèse est largement défendue [61].

2.2.1.2 Fonction de Fitness

Dite aussi fonction d'évaluation, elle représente la partie clé d'un AE. Elle permet d'évaluer un individu et est spécifique du problème sur la représentation du génome. Cette fonction doit permettre la comparaison entre les différents individus, éventuellement avec un ordre non total. Pour poursuivre l'analogie avec la biologie, la fonction évalue la façon d'adaptation d'un individu à son environnement. Les fonctions d'évaluation représentent l'espace de recherche. Elles sont généralement multimodales (c.-à-d., elles possèdent plusieurs optima locaux) lorsqu'elles sont utilisées dans les AEs. La fonction d'évaluation décrit la fitness que l'algorithme tente d'optimiser, par échantillonnage en utilisant une population. En effet, la population se propage sur l'espace de la fitness, et les opérateurs de variation appliqués sur ces individus servent à les mouvoir en essayant de les déplacer vers les parties les plus intéressants de l'espace de recherche [159], [124].

Les fonctions d'évaluation représentent le problème à résoudre. Elles possèdent différents temps d'exécution. Le temps passé à l'évaluation de la population représente souvent la partie majeure dans l'exécution d'un AE. Typiquement, le temps d'évaluation de la programmation génétique est un calcul très intensif, comme pour la régression symbolique, un individu est comparé à un ensemble d'entrainement qui peut aller vers des milliers de cas [124].

Inversement, pour résoudre les problèmes avec contraintes, l'évaluation d'un individu consiste à associer des pénalités à chaque fois qu'une contrainte est en panne (la totalité

du temps, la densité de la solution). Ces calculs peuvent être très légers, ce qui signifie que l'évaluation de la population peut représenter une proportion négligeable par rapport au temps total d'exécution de l'AE. Ces considérations sont relativement importantes, car si l'évaluation est très coûteuse [124], il est intéressant d'utiliser des opérateurs de variation complexes qui permettront de sauver lors des évaluations.

À l'opposé, le temps nécessaire à trouver une solution peut être plus long si ces opérateurs coûteux sont utilisés, si l'évaluation n'est pas coûteuse. Dans un tel cas [124], produire autant d'individus que possible et en comptant sur l'évolution artificielle pour sélectionner les meilleurs individus peut être plus efficace.

2.2.1.3 Moteur de variation

Les individus seront soumis à un certain nombre d'opérations de variation pour imiter les changements génétiques du gène, et qui sont fondamentaux à la recherche de l'espace des solutions.

Les opérateurs de variation sont utilisés pour créer de nouveaux individus. L'idée sous-jacente est que la création d'un individu à partir des parents sélectionnés pour leurs qualités devrait être en mesure de créer un meilleur enfant. Les opérateurs de variations prennent comme entrée un nombre de parents et produisent un nombre d'enfants comme sortie [136], [194].

Les trois sous-sections suivantes sont de brefs résumés des principaux opérateurs de l'évolution comme vus par la communauté des AEs. Le lecteur doit examiner des livres d'introduction pour plus de détails et d'exemples [136], [46].

a. Opérateurs de sélection :

Un AE possède deux classes distinctes d'opérateurs de sélection, la sélection pour la reproduction et la sélection pour la survie. Le choix de l'opérateur à imposer est abordé dans la discussion du contexte. Les deux opérateurs ont une base dans les théories de l'évolution de la biologie [159].

La fonction de fitness évalue la qualité des individus, ce qui permet une comparaison facile des individus avec un tri de la population pour ne prendre que le meilleur.

Toutefois, un tel mécanisme, où les meilleurs sont utilisés de façon déterministe, présente des problèmes de convergence prématurée et n'a pas de sens pour la sélection des parents. Dans la phase de réduction, tout en gardant dans la nouvelle population des individus qui ne sont pas les meilleurs, mais qui sont encore "assez bon", peut aider à maintenir une certaine diversité dans la population, ce qui empêche une telle convergence prématurée. L'utilisation des opérateurs stochastiques pour la sélection a été proposée très tôt (Holland [46]). L'idée est d'utiliser une sélection stochastique, qui est sollicitée selon la valeur de la fonction de fitness. La « sélection de fitness proportionnelle » ou « la roue de roulette » suit ce principe en attribuant une probabilité, à un individu, qui est proportionnelle à sa valeur de fitness.

Plusieurs sélecteurs modernes incluent la très bonne sélection, celle de tournoi, qui choisit au hasard t individus de la population et renvoie le meilleur. Ce sélecteur de tournoi est largement utilisé de nos jours [124].

b. Opérateurs de croisement :

Une instance des opérateurs de variation est l'opérateur de croisement appelé aussi opérateur de recombinaison. Cet opérateur prend habituellement plusieurs parents afin de produire un ou plusieurs enfants après la recombinaison des gènes des parents. Le croisement devrait tenir compte de la structure de l'individu. Plusieurs types de croisement existent, adaptés à et/ou inspirés par les différentes représentations des individus. La figure (2.4) présente l'opérateur de « croisement uniforme », qui fait le choix aléatoire des gènes à partir de trois parents pour produire l'enfant. En effet, ce croisement uniforme est initialement adapté aux algorithmes génétiques et leurs représentations binaires. Un opérateur typique de la stratégie d'évolution pourrait être le croisement barycentrique, où le gène de l'enfant est le centre de gravité pondéré des gènes des parents, mais des opérateurs plus évolués existent : le croisement binaire simulé (SBX [33]) crée un gène réel estimé dont la valeur est proche de celle de l'un des deux parents. Enfin, l'un des parents peut être choisi comme le seul parent et l'enfant qui en résulte est donc un clone du parent choisi [159], [124], [136], [194].

Le croisement dans la programmation génétique est aussi différent, et également étudié par Koza dans [103], Koza and Rice dans [104], Koza et al. dans [105]. Le problème ici est de croiser deux arborescences représentant des fonctions mathématiques, par exemple.

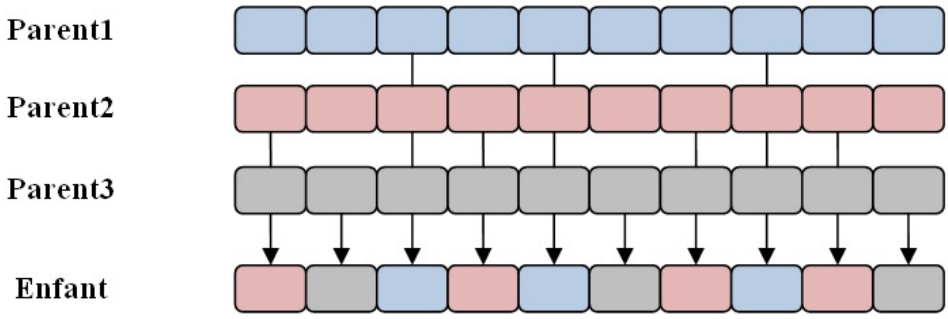


FIGURE 2.4 – Le croisement uniforme entre trois parents [124].

c. Opérateur de mutation :

L'opérateur de mutation est peut-être l'opérateur unaire le plus élémentaire dans l'AE, qui prend généralement l'enfant récemment créé comme paramètre. Il s'inspire de la mutation de base observée en génétique biologique, en raison des différents effets de transcriptions et des changements spontanés dans les chromosomes observés dans les primitives d'études évolutionnaires. La mutation vise à déplacer légèrement l'enfant dans l'espace de recherche afin d'explorer éventuellement des zones inaccessibles par un croisement seul. Pour une représentation basée sur des nombres réels, le procédé ajoutant un bruit gaussien à un ou plusieurs des gènes individuels, est une méthode courante, mais dans les SEs [5], un opérateur de mutation auto-adaptative est utilisé. De la même manière, une matrice de covariance peut être ajoutée à la mutation avec un vecteur d'écart-type, ce qui donne une description complète de la distribution normale multidimensionnelle généralisée pour muter les descendants.

Cette mutation auto-adaptative très efficace est largement utilisée dans les SEs, où chaque gène individuel a une variance spécifique. Le schéma de mutation le plus commun pour les génomes binaires consiste à utiliser une probabilité bit-flip appliquée contre chaque bit indépendamment. Donc, le nombre réel de bits modifiés chez un individu n'est pas fixe. Le taux de mutation typique est 1/n, en donnant un bit unique prévu permettant la variabilité. D'autres ont utilisé un taux de 1/nm, donnant un bit unique attendu dans l'ensemble de la population à être renversé dans une génération donnée [159].

2.2.1.4 Récapitulatif

Récapitulation

Individuel	L'organisme vivant	La Solution Candidate
Chromosome	ADN protéine histone brin	séquence d'objets de traitement
	Décrit le «plan de construction» et donc (quelques-uns) des traits d'un individu sous forme codée.	
	Généralement plusieurs chromosomes par individu	En général un seul chromosome par individu
Géne	Partie du chromosome	Objet de calcul (par exemple, un bit, un caractère, un numéro, etc.)
	Unité fondamentale de l'héritage, qui détermine une caractéristique partielle d'un individu.	
Allèle	Forme ou « valeur » de gène.	Valeur d'un objet de calcul.
	Dans chaque chromosome il y a au plus une forme / valeur d'un gène.	
Locus	Position d'un gène	position d'un objet de calcul
	Pour chaque position dans un chromosome, il y a un gène exactement.	
Phénotype	L'apparence physique d'un organisme vivant.	Mise en oeuvre / application d'une solution candidate.
Génotype	Constitution génétique d'un organisme vivant.	Codage d'une solution candidate.
Population	Définir des organismes vivants.	Multi-ensemble de chromosomes.
Génération	Population à un moment dans le temps.	Population à un moment dans le temps.
Reproduction	Créant les descendants d'un ou plusieurs (généralement deux) organismes (Parent).	Création (enfants) chromosomes provenant d'un ou plusieurs chromosomes (parent).
Fitness	Aptitude / conformité d'un organisme vivant.	Aptitude / qualité d'une solution candidate.
	Détermine les chances de survie et de reproduction.	

TABLE 2.2 – Notions évolutionnaires de base en biologie et en informatique [107].

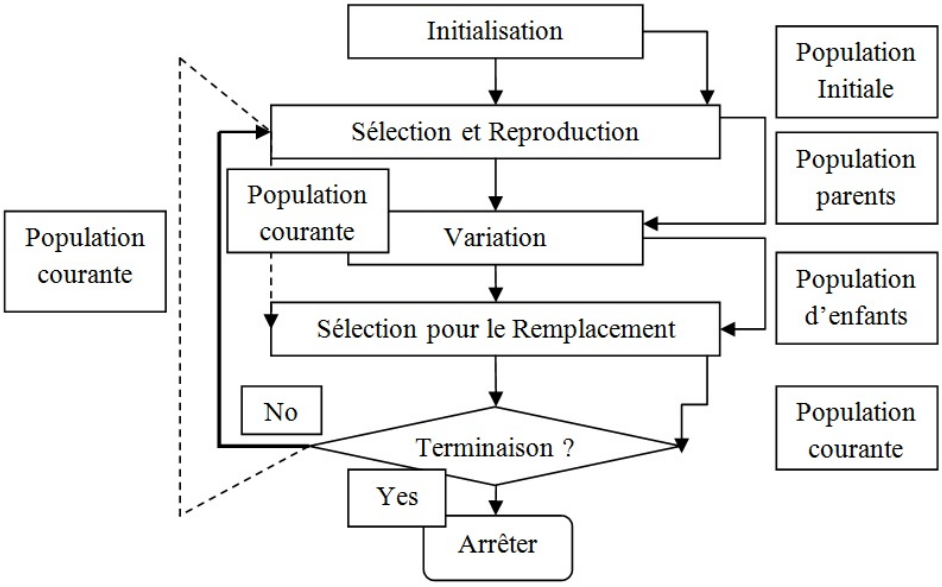


FIGURE 2.5 – Grandes lignes d'un algorithme évolutionnaire générique.

2.2.2 Domaines d'application des AEs

Les AEs permettent de résoudre, non seulement des problèmes purement théoriques en combinatoire, en économie, en apprentissage, dans la théorie des jeux, mais également des problèmes liés à des applications réelles complexes [28], [58], [68], [200], [204], [208], [179]. Ainsi, ils sont utilisés pour analyser des sondages de sous-sol et détecter des champs pétrolifères, établir des emplois du temps, prévoir les cours de la bourse (nombreuses applications financières), contrôler les pipe-lines de gaz, concevoir des automobiles, ils sont également utilisés en logistique (meilleure solution actuelle au problème du voyageur de commerce, avec une approche couplant AEs et recherche opérationnelle), pour optimiser les ailes d'avion, les empennages de missiles supersoniques, les aubes de turbines, les hélices, les tuyères de propulseurs, les manoeuvres des avions de combat, les allocations de routes aériennes, les allocations dynamiques de fréquences en téléphonie mobile (meilleur résultat actuel), le positionnement d'antennes, le routage dans les réseaux, l'analyse d'images médicales, le calcul des trajectoires de robots, la recherche de gènes responsables de maladies génétiques, l'approximation de formes 2D par des fractales (pour compression fractale d'image)...

2.3 Analyse de complexité des algorithmes évolutionnaires

2.3.1 Définition de l'analyse algorithmique

Analyser un algorithme consiste à prévoir les ressources qui lui seront nécessaires, en temps et en espace, pour résoudre un problème. La taille de l'entrée est le nombre d'éléments constituant l'entrée. Elle dépend du problème, par exemple, la somme des tailles des listes dans le cas d'un problème de fusion de deux listes. Dans la plupart des

cas, le temps d'exécution et l'espace mémoire de stockage d'un algorithme augmentent en fonction de la taille de son entrée [36].

Lorsqu'un problème de calcul peut être résolu en utilisant un algorithme donné, l'issue suivante, après la solvabilité, est la complexité du temps et de l'espace nécessaires pour la méthode de la solution. Par complexité de l'espace, nous noterons la mémoire requise pour la solution algorithmique, une fois un modèle de calcul adopté. La complexité par rapport au temps se réfère au nombre d'étapes algorithmiques exécutées par le modèle de calcul. Notre intérêt est lié au comportement de l'algorithme lorsqu'il est appliqué à des problèmes de plus en plus grands. Généralement, la complexité de l'espace n'est pas considérée comme une issue primordiale, puisque les modèles informatiques sont fournis avec une mémoire infiniment élevée. Ce qui est plus intéressant pour les applications consiste en la complexité du temps et donc, et par la suite, lorsqu'on parle de la complexité d'un algorithme, nous faisons référence effectivement à sa complexité par rapport au temps [160].

2.3.2 Analyse des algorithmes évolutionnaires

Les AEs ont été appliqués dans le domaine de la bioinformatique, la conception de circuits, l'exploration des données, la recherche d'informations, etc. Malgré le succès remarquable obtenu par les AEs sur les problèmes pratiques, les AEs sont souvent critiqués par l'absence d'une base théorique solide. Actuellement, un tel fondement théorique est très souhaité et recherché afin d'arriver à une compréhension approfondie de la force et la faiblesse des AEs actuels et par la suite à l'élaboration des AEs plus efficaces [206].

Une grande partie de la recherche théorique des AEs [212] consiste en l'analyse d'exécution rigoureuse qui provient de l'informatique théorique et l'analyse probabiliste des algorithmes aléatoires.

L'analyse de la complexité de calcul des AEs est devenue une branche reconnue et pas moins importante dans la théorie du calcul évolutif, surtout avec les grands avancées récentes. Ceci se fait habituellement par l'analyse de l'optimisation attendue du temps mesuré par le moyen d'un nombre d'évaluations et la description de sa croissance en tant qu'une fonction de mesure de la taille de l'espace de recherche. Les résultats asymptotiques décrivant seulement l'ordre de la croissance en sont les plus souvent dérivés [96].

Les théories consistant à expliquer pourquoi et comment fonctionne un AE, sont encore relativement peu nombreuses malgré les efforts récents. La complexité du temps de calcul des AEs est largement inconnue [79], sauf pour quelques cas simples.

Le premier temps de frappe attendu (The First Hiting Time) représente l'une des questions théoriques les plus importantes des AEs [206], car il implique la complexité moyenne du temps de calcul. Le premier temps de frappe attendu d'un AE est le temps en un cycle, qui permet de trouver la solution optimale pour la première fois.

L'analyse de la complexité des AEs ne se fait pas avec les méthodes classiques existantes pour de multiples raisons. On peut en citer les suivantes :

- Ils sont stochastiques ;
- Ils ne sont pas conçus pour être analysés ;
- Ils ne sont pas conçus pour des problèmes spécifiques ;
- Ils sont trop complexes, non linéaire, chaotiques ;
- Il sont mathématiquement infaisable...

2.3.3 Vue générale sur les travaux existants concernant l'analyse des algorithmes évolutionnaires

Les premières études ont été soucieuses d'expliquer les comportements des AEs plutôt que d'analyser leurs performances. La théorie du schéma est probablement l'outil le plus populaire utilisé dans ces premières tentatives. En particulier, elle a été d'abord proposée dans le but d'analyser le comportement d'un AG simple [144].

Comme *Eiben et Rudolph* [45] ont déclaré, lors de l'analyse initiale de ces outils, que bien que la théorie du schéma ait été considérée comme fondamentale pour la compréhension des AGs [144] jusqu'au début des années quatre vingt dix, elle ne peut pas expliquer le comportement dynamique ou limité des AEs.

Dans les années quatre vingt dix, avec l'avènement de la théorie des chaines de Markov dans l'analyse des AEs, les premiers résultats de convergence sont apparus liés à leur délai de comportement pour les problèmes d'optimisation. Avec les transitions d'états d'un AE et leur nature probabiliste, la définition de la convergence déterministe n'était évidemment pas le cas. Ainsi, la définition de la convergence stochastique a dû être utilisée. Idéalement, un AE devrait être capable de trouver la solution au problème, il s'attaque avec une probabilité 1 après un nombre fini d'étapes, indépendamment de son initialisation [144], [193].

Dans un tel cas, l'algorithme est sensé visiter l'optimum global dans un temps fini. Si l'algorithme influence la solution dans la population, alors il est considéré converger vers l'optimum. Utilisant les chaines de Markov, *Rudolph* [161] a prouvé que les algorithmes génétiques canoniques qui utilisent la mutation, le croisement et la sélection proportionnelle ne convergent pas vers l'optimum global alors que les variantes élitistes le font. Il a alors étendu son analyse en définissant des conditions générales selon lesquelles, s'ils sont satisfaits par un AE, ils garantissent sa convergence. Ce travail a été motivé par le postulat suivant : "il n'est pas nécessaire de construire exactement un modèle de Markov quantitatif pour chaque AE afin d'enquêter sur la limite du comportement".

Les Conditions de non-convergence ont été également données. Ainsi, c'est uniquement lorsqu'un AE ne satisfait pas les conditions données qu'une analyse spécifique s'avère nécessaire. Une autre nouveauté réside dans le fait que le travail intensif concernant la convergence des AEs avait finalement conduit à des démonstrations simples qui ne nécessitent plus la théorie des chaines de Markov. Cependant, si un algorithme converge, l'analyse du temps, qui limite le comportement, ne donne aucune indication concernant le temps prévu pour trouver la solution. *Aytug et Koehler* [4] ont proposé une analyse sur le nombre de générations suffisant pour garantir la convergence avec un niveau fixe de confiance indépendamment de la fonction d'optimisation. Les résultats obtenus ont été améliorés par *Greenhalgh et Marshall* [70]. Néanmoins, la meilleure borne supérieure de l'analyse qui peut être garantie est la même que celle d'un algorithme aléatoire en choisissant au hasard des individus de façon indépendante dans chaque génération. L'échec de ces approches pour obtenir des bornes utiles, a confirmé l'idée que lors de l'analyse de la complexité du temps pour les problèmes de recherche heuristiques, la fonction d'optimisation doit être considérée. Une autre confirmation d'une telle idée est venue quand *Droste* [41] a prouvé l'existence de fonctions pour lesquelles la stratégie d'évolution (1+1)-AE trouve l'optimum global dans un temps égal à O (n^n). Ainsi, la nécessité de mesurer la performance des AEs pour des problèmes spécifiques est devenue évidente dans les années quatre-vingt-dix. En particulier, il s'est avéré nécessaire d'utiliser des mesures standard

telles que la relation entre la taille du problème abordé et le temps prévu pour que la solution soit trouvée. *Beyer* [8] a confirmé qu'il n'y avait presqu'aucun résultat, avant le milieu des années quatre vingt dix qui estime le temps d'optimisation dans le pire des cas d'un AE qui travaille sur un problème quelconque, ou qui estime la probabilité de l'obtention d'une solution remplissant une demande minimale avec t(n) étapes.

Par conséquence, une tentative a finalement été réalisée en vue d'une analyse systématique de la complexité du temps qui transforme la théorie des AEs en une partie légale de la théorie des algorithmes efficaces.

En raison de leur nature stochastique, l'analyse de la complexité du temps des AEs est une tâche délicate. Les premières tentatives sont liées aux AEs de base (tels que le (1+1)-AE) sur des fonctions simples avec quelques propriétés structurelles. Choisir des problèmes pseudo-booléenne simples avec « une belle structure » sert à :

- Décrire le comportement d'un AE sur les problèmes typiques des fonctions ;
- Montrer quelques comportements extrêmes des AEs ;
- Réfuter largement les suppositions acceptées ;
- Comparer les différentes variantes des AEs.

Après les motivations ci-dessus, en 1998, *Droste* [40], *Jansen et Wegener* [94] ont analysé les (1+1) AE sur les fonctions pseudo booléenne telle qu'ONEMAX et bin. Ils ont étendu leurs résultats à des fonctions linéaires en prouvant une borne supérieure de O (n log n). Dans la même année, *Droste* [39] a également analysé, de nouveau, le (1+1) AE sur les fonctions uni-modales résultant du refus de la supposition, largement répandue, que les AEs sont efficaces sur toutes les fonctions en produisant un exemple pour lequel l'algorithme prend un temps exponentiel par rapport au temps prévu. Dans l'année 2000, *Wegener et Witt* [193] ont prouvé qu'on peut également trouver des fonctions quadratiques dont le temps de l'optimisation attendue pour le (1+1) AE est exponentiel pour un problème de taille n. En outre, ils ont montré comment un (1+1) AE n'accepte pas des individus avec les mêmes valeurs de fitness. Un autre objectif de ces études a été celui de l'obtention de méthodes mathématiques et d'outils qui peuvent s'avérer très utiles dans l'analyse des AEs sur des problèmes plus sophistiqués et réalistes, en général. Ceci était nécessaire, car il s'est avéré que l'utilisation des chaines de Markov n'a pas été suffisante pour l'analyse. Certains outils ont dû être importés à partir du domaine général de l'analyse des algorithmes aléatoires et d'autres ont été spécialement définis [193].

En conséquence de ces préliminaires, il est désormais possible d'analyser le (1+1) AE sur les problèmes d'optimisations combinatoires avec des applications pratiques. De plus, quelques résultats réalistes concernant les AEs ayant des populations et exploitant les croisement sont également été obtenus. De tels travaux sont devenus possibles en utilisant des outils obtenus dans les précédentes analyses, directement ou en les étendant aux nouveaux problèmes abordés. Toutefois, dans certains cas, de nouveaux outils semblent nécessaires

Les résultats les plus importants sont donnés en collaboration avec les idées véhiculées par les outils mathématiques les plus utilisés dans les analyses. Sans prétendre d'être complète, l'étude déjà faite a été limitée aux problèmes d'optimisation combinatoire avec des fonctions objectives simples. Bien que cette restriction se croise avec un grand nombre de travaux théoriques liés, elle sert à mettre l'accent sur un thème général sans provoquer une très grande dispersion [144].

2.3.4 Éléments d'analyse des algorithmes évolutionnaires

Afin de faire une telle analyse, nous devons connaître les éléments d'analyse des méta-heuristiques et spécialement celle des AEs.

2.3.4.1 Exploitation et Exploration

Les initiateurs des AEs ont introduit les notions d'*exploitation* et d'*exploration*. L'exploitation insiste sur la capacité, que doivent posséder ces algorithmes, d'examiner en profondeur, par une méthode des zones de recherche particulières alors que l'exploration met en avant la capacité de découvrir des zones de recherche prometteuses. Il est donc pertinent d'analyser l'ensemble des méta-heuristiques en fonction de ces deux notions [73].

Les méthodes de voisinage entendent exploiter les bonnes propriétés de la fonction de voisinage pour découvrir rapidement des configurations de bonne qualité. Elles reposent sur l'hypothèse que les zones les plus prometteuses sont situées à proximité des configurations (déjà visitées) qui sont les plus performantes. Le principe d'exploitation consiste à examiner en priorité ces zones [31].

Dans les AEs, la sélection a pour effet de concentrer la recherche autour des configurations de meilleure performance. Plusieurs méthodes introduisent des mécanismes d'exploitation spécifiques supplémentaires.

Cependant, l'application systématique du seul principe d'exploitation ne permet pas une recherche efficace. En effet, l'exploitation conduit à confiner la recherche dans une zone limitée qui finit par s'épuiser. Le cas de l'amélioration itérative rapidement piégée dans un optimum local illustre du façon caractérisée ce phénomène [73]. Une autre illustration, souvent évoquée, est fournie par le problème de convergence prématurée des AEs : du fait de la sélection, la population finit par n'être constituée que d'individus similaires. L'une des préoccupations majeures dans les AEs consiste d'ailleurs à préserver le plus longtemps possible une diversité suffisante dans la population. En termes de palliatif à ce type de difficulté, la solution consiste à diriger la poursuite de la recherche vers de nouvelles zones, c.-à-d., à recourir à l'exploration [74].

En plus de la relance, les heuristiques emploient principalement deux autres stratégies dans le but d'explorer : la première consiste à perturber aléatoirement et la seconde à caractériser les régions visitées pour pouvoir ensuite s'en éloigner. La première stratégie qui est aussi la plus simple consiste à introduire des perturbations aléatoires : c'est le cas pour les mutations aléatoires dans les AEs ainsi que pour la génération aléatoire d'un voisin dans le recuit simulé. Dans les deux cas, la configuration courante est altérée de manière aléatoire et un mécanisme d'acceptation est appliqué a posteriori. Ces deux méthodes admettent donc en permanence des perturbations aléatoires. La deuxième stratégie, permettant l'exploration, consiste à mémoriser au cours de la recherche, des caractéristiques des régions visitées et à introduire un mécanisme permettant de s'éloigner de ces zones. Citons également la pondération qui utilise une variante de cette idée, nous pouvons remarquer que cette seconde stratégie nécessite l'utilisation de la notion de mémoire. Notons enfin que la recombinaison est parfois présentée comme une manière supplémentaire de contribuer à l'exploration [73].

Nous venons de voir qu'il existe différents modes d'exploration. Les méta heuristiques se différencient également selon la manière dont elles font varier l'intensité de l'exploration au cours de la recherche.

La recombinaison constitue un autre principe général qui complète l'exploitation et

l'exploration. Elle consiste à construire de nouvelles configurations en combinant la structure de deux ou plusieurs bonnes configurations déjà trouvées [74].

2.3.4.2 Méthodes générales et méthodes spécifiques

Contrairement aux algorithmes traditionnels dédiés à un problème spécifique, les AEs constituent des mécanismes très généraux qui peuvent être adaptés pour traiter de nombreux problèmes différents. Pour expliquer l'efficacité d'un algorithme spécifique, on invoque souvent le fait qu'il utilise des connaissances spécifiques du problème. Pour expliquer l'efficacité d'une méta-heuristique, deux types d'arguments opposés sont généralement avancés.

Selon le premier point de vue, des mécanismes généraux suffisamment puissants ont par eux-mêmes la faculté de mener efficacement la recherche sans disposer d'information spécifique du problème considéré (contexte de boîte noire) : c'était notamment le point de vue classique porté sur les AEs. Malheureusement, de même qu'il ne puisse exister de stratégies avantageuses dans un jeu de hasard comme la roulette, il existe également des limitations théoriques fondamentales qui amenuisent les espoirs d'une méthode aveugle dans le cas le plus général. En fait, le théorème «No Free Lunch (NFL)» montre que pour les problèmes de type boîte noire, toutes les méthodes sont équivalentes et font aussi bien, ou plutôt aussi mal, que l'énumération aléatoire [199].

Selon le point de vue opposé, la puissance d'une méta-heuristique est d'abord liée à son aptitude à intégrer des connaissances spécifiques du problème. La connaissance du problème la plus fondamentale réside dans le codage du problème et dans le choix de la fonction de voisinage. Plusieurs auteurs insistent sur l'importance du codage du problème. Dans le cas du voyageur de commerce, plusieurs types de codages différents ont été proposés et conduisent à des performances très variées. En général, il n'existe pas de codage universellement efficace. Un «bon» codage doit permettre de restreindre l'espace de recherche et intégrer des connaissances du problème.

Les AEs tentent également d'améliorer leur efficacité en incorporant des connaissances supplémentaires dans leurs opérateurs. Plus les opérateurs d'une méthode utilisent des connaissances spécifiques, plus la méthode dispose de moyens potentiels pour conduire efficacement la recherche. En contrepartie, l'intégration de ces connaissances spécifiques nécessite un effort pour spécialiser ou adapter la méthode. En général, une méthode offrant des possibilités d'intégrer des connaissances du problème a plus de chance de produire de bons résultats, mais demande un effort d'adaptation et de spécialisation. Au contraire, une méthode très générale qui prétend n'intégrer aucune connaissance propre ne peut pas être compétitive [73].

2.3.5 Méthodes d'analyse des algorithmes évolutionnaires

Depuis 1990, l'approche systématique de l'analyse des heuristiques de recherches aléatoires a élaboré d'une façon complète un nouveau domaine de recherche [46]. Maintes analyse de calcul du temps de complexité des algorithmes évolutionnaire ont été réalisées depuis le milieu des années quatre vingt dix. Les premiers résultats étaient liés à des algorithmes très simples, tels que l'AE de type (1+1). Ces efforts ont produit une meilleure compréhension de la façon dont un AE effectue ses différents traitements sur divers types de fitness et de remise en forme générale des outils mathématiques qui peuvent être étendus à l'analyse des AEs les plus complexes sur des problèmes plus réalistes. En effet,

durant ces dernières années, il a été possible d'analyser l'algorithme (1+1) AE sur des problèmes d'optimisation combinatoire avec des applications pratiques et plus réalistes basées sur des AEs avec population [144].

En générale, si X_f est la variable aléatoire mesurant le temps durant lequel la solution est trouvée par un AE pour une certaine fonction f, alors l'analyse de l'exécution d'un algorithme aléatoire consiste à :

1. Estimer $E(X_f)$ dans le meilleur, le moyen et le pire des cas ;
2. Calculer de la probabilité de réussite $Pr(X_f <= t)$ dans le meilleur, le moyen et le pire des cas.

Comme les AEs constituent des processus stochastiques dont chaque état ne dépend en général que de la valeur de l'état précédent, la méthode la plus simple et la plus utilisée pour modéliser un algorithme évolutionnaire est les chaines de Markov.

Cependant, il n'est pas toujours facile de dériver des expressions explicites, ou les limites de temps, pour l'estimation de la variable aléatoire X_f directement à partir de la matrice de transition d'une chaine de Markov. Pour d'autres détails, des outils mathématiques sont mis à la disposition des chercheurs dans le but d'analyse les algorithmes évolutionnaires, le lecteur peut se référer au travail de Jun He et coll. [144].

2.4 Calcul de la complexité

Cette section discute un état de l'art concernant le calcul de complexité des AEs tout en considérant les améliorations faites sur l'analyse des AEs à base population et en décrivant les raisons derrière les travaux récents dans ce domaine.

2.4.1 De l'individu à la population

D'une manière similaire à l'analyse du type (1+1) AE, les premiers résultats théoriques pour les AEs basés population ont été obtenus avec des fonctions ayant des structures intéressantes. L'idée étant de comprendre quand est-ce qu'une population ou l'opérateur de croisement pourraient être bénéfiques. Les premières analyses ont été motivées par la conjecture que les algorithmes génétiques ont été supposés comme une forme surperformé des AEs de type (1+1), sans que des preuves théoriques aient été présentées. En particulier, il y avait une nécessité de prouver l'existence d'une des fonctions pour laquelle les algorithmes génétiques présentent de meilleures chances de réussite. Une autre direction dans l'analyse des algorithmes évolutionnaire à base population a été de considérer des stratégies d'évolution différentes plutôt que de d'exploiter directement les AGs.

2.4.1.1 Quand est-ce que les populations sont bénéfiques

Lorsque l'on compare les AEs d'un seul individu contre celle à base population, l'analyse doit prendre en compte le fait que les anciens algorithmes peuvent utiliser des stratégies de redémarrage. En dehors de l'équité dans la comparaison, ceci est également nécessaire du fait que les AEs nécessitent, dans leur pratique, un redémarrage probable après un certain temps. En 1998, *Rudolph* [162] a déclaré "qu'il est facile de voir qu'un algorithme évolutionnaire avec une large population ne peut pas être pire qu'un (1+1) AE en ce qui concerne le nombre d'itérations". Toutefois, le nombre de générations n'est

pas une mesure de performance équitable pour la comparaison, alors que le nombre d'évaluations de fitness est plus approprié.

En 1999, Jansen et Wegner ont présenté, pour la première fois, une analyse d'une fonction (en l'occurrence JUMP) qui peut être optimisée de manière plus efficace avec l'utilisation du croisement uniforme, contrairement à un AE qui ne l'utilise pas, à la condition que les réplications des individus dans la population soient évitées. Les investigations d'exécution typiques et les résultats du problème du collecteur coupon ont été d'une grande utilité pour cette analyse. Le problème est de savoir si les populations elles-mêmes pourraient être bénéfiques est resté ouvert [93].

En 2001, *Jansen et Wegner* ont étendu leurs résultats par la production d'une classe de fonctions pour lesquelles un algorithme évolutionnaire basé population, sans utilisation de croisement, nécessite un temps polynomial pour son optimisation. D'autre part, un AE de type (1+1) prend un temps d'optimisation super polynomial [95]. Afin d'obtenir leurs résultats, ils doivent éviter une population qui ne converge pas trop rapidement. L'algorithme satisfait la condition dans la mesure où la mesure de diversité est appliquée. Ainsi, deux possibilités émergent : l'évitement des duplications et des répétitions. Le dernier mécanisme de diversité n'assure que la condition que chaque enfant possède au moins un bit muté à chaque génération, et c'est le mécanisme utilisé par l'algorithme. Une fois encore, les exécutions typiques sont souvent utiles dans ces analyses. Les classes de fonctions de Royal Road associées à un AE (Steady state GA), et utilisant du croisement uniforme ainsi que le croisement en un seul point sont plus performantes que toute stratégie d'évolution. *Witt* a introduit des fonctions pour lesquelles, en augmentant la taille de la population, le temps d'exécution s'améliore de l'exponentiel vers le polynomial sans l'utilisation des mécanismes de diversité ou de croisement (le meilleur fossé prouvé précédemment était le super-polynomial vs le polynomial). Toutefois, des exemples avec des temps d'exécution inverses sont également donnés [197].

En 2003, *Stroch et Wegner* ont introduit de nouvelles fonctions pour lesquelles les mêmes résultats précédents peuvent être obtenus, mais avec une population d'une taille minimale de 2 individus [178].

En 2002, *He et Yao* [80] ont présenté un autre document qui prouve la façon dont une population peut être bénéfique pour des individus uniques sur toutes sortes de problèmes pseudo-booléen. Cette fois, les résultats sont atteints grâce à des stratégies de sélection soigneuses, bien qu'un seul bit flips soit utilisé. Les résultats décrits ci-dessus prouvent que parfois les populations et/ou croisements peuvent être utiles en montrant les classes de fonctions pour lesquelles la combinaison d'une certaine taille de population et un opérateur de croisement bien choisit de fournir une meilleure performance. Cependant, il n'est pas encore très clair quand et comment on peut opérer au choix de la taille de la population ou de l'opérateur de croisement. Aussi, il y a de nombreux exemples où ils ne s'avèrent pas utiles à tous les cas.

2.4.1.2 Stratégies d'évolution basées populations

Une autre direction dans l'analyse des AEs à basse population a été celle de considérer les différents types de stratégies d'évolution plutôt que de s'intéresser directement aux AGs. Jansen a analysé les AEs de type (1+λ) sur les principaux ESs, et les ONEMAX et a étendu leurs résultats de façon empirique à d'autres fonctions de référence plus complexes. Malheureusement, les analyses présentées ici ne sont pas capables d'offrir une forte prédiction sur la valeur appropriée de λ pour un cas de figure arbitraire. Toujours en

considérant les résultats de la stratégie d'évolution basée population, Stroch et Jäagerskuppe, au contraire, ont comparé les stratégies (1+λ) et (1,λ) et ont prouvé qu'il existe certaines valeurs de λ pour lesquelles, non seulement la stratégie des virgules s'exécute aussi bien que toute autre stratégie, mais elle peut également les supplanter sur certaines fonctions. Une extension des niveaux artificiels de la fitness est introduite à l'analyse [92].

D'autre part, *Witt* a analysé les AEs de type (μ + 1) sur les fonctions pseudo-booléennes les plus connues et a produit un exemple pour lequel l'algorithme est plus efficace que le AE de type (1+1) [198]. Cependant, sur les fonctions pseudo-booléennes classiques, aucun bénéfice n'a été observé ni pour la population des descendants ni pour la population parente. Dans le meilleur des cas, la fonction pour laquelle les populations donnent une amélioration considérable est plus compliquée que les autres analyses, confirmant partiellement la conjecture empirique. Alors que les preuves de la limite supérieure utilisent généralement la technique des fonctions de potentiel, l'approche des arbres est exploitée pour l'obtention de la borne inférieure. Les résultats précédents font apparaître la nécessité d'analyser les AEs basés population qui peuvent être plus proches des applications réelles. Des exemples simples sont les problèmes classiques d'optimisation combinatoire présentant des applications pratiques. Si un AE basé population est plus performant qu'un AE à base d'un seul individu sur des problèmes d'optimisation combinatoire difficiles, alors il peut y avoir une bonne chance qu'ils obtiennent de meilleurs résultats sur les applications difficiles de la vie réelle [144].

2.4.2 Récapitulation

Les heuristiques de recherches aléatoires sont des algorithmes de recherche robustes pour des problèmes indépendants. Elles ont été conçues pour les raisons pratiques suivantes :

1. Certains problèmes concernant des applications réalistes doivent être résolus rapidement, et souvent il n'y a pas assez de ressources financiéres, de temps ou de connaissances pour construire un algorithme de problème dépendant.
2. La fonction qui doit être optimisée peut ne pas être connue, et uniquement par échantillonnage de l'espace de recherche, quelques connaissances sur le problème peuvent être acquises.

Les objectifs précédents font apparaître la nécessité de concevoir des algorithmes de problèmes indépendants, même si un algorithme est construit spécialement pour résoudre un problème devrait mieux fonctionner. D'où l'accent est mis sur la conception d'heuristiques qui devraient bien fonctionner sur une large classe de problèmes. Concernant leur analyse théorique, il n'est pas possible d'étudier les performances des algorithmes sur des fonctions inconnues. Ainsi, leur comportement doit être analysé sur de larges classes de fonctions et sur des problèmes d'optimisation combinatoire connus avec des applications pratiques. Qu'il s'agisse des résultats positifs ou négatifs, tous les deux peuvent aider à mieux comprendre sur quel genre de problèmes un algorithme peut être le meilleur ou le pire, le cas échéant. Ainsi, ceci peut aider les praticiens dans le choix de l'heuristique et l'ajustement de ses paramètres. Par ailleurs, les algorithmes évolutionnaires sont souvent utilisés pour résoudre des problèmes d'optimisation combinatoire, car ils sont faciles à utiliser et les résultats empiriques qui suggèrent leur performance sont souvent couronnés de succès. Il est généralement admis qu'un algorithme soit efficace si son temps d'exécution

attendu est majoré par une fonction polynomiale de la taille du problème. En résolvant un problème avec un algorithme, la première question qui nécessite une réponse est de savoir si l'algorithme peut trouver la solution efficacement [57].

2.5 Mesures de performance

Les temps d'exécutions prévus et la probabilité de réussite sont les mesures de performance globale, ces mesures de performance étant typiquement les plus utilisées pour les algorithmes aléatoires. Dans la théorie des algorithmes évolutionnaires, plusieurs autres aspects sont considérés. Les autres mesures de performances ont une certaine valeur, mais nous pensons que finalement leur analyse est seulement un outil pour obtenir des résultats sur le comportement global. Afin de comprendre le comportement global, il est utile de comprendre le comportement local. La qualité de bénéfice et le taux du progrès sont comme des mesures de performances locales qui décrivent le comportement d'une seule étape. La théorie du schéma est aussi un résultat qui garantit un certain comportement pour une seule étape. Pour les chaines de Markov (comme les algorithmes évolutionnaires) la transition des probabilités pour une seule étape détermine le comportement global. Cependant, les mesures de performances locales sont dites « statistiques insuffisantes », ce qui implique qu'en général, il est impossible de déduire les états d'un comportement global. Nos mesures de performance globales décrivent le comportement avec un temps limité et raisonnable [193].

En définitive, les résultats les plus intéressants sont obtenus avec la modélisation des algorithmes évolutionnaires comme des systèmes dynamiques. Toutefois, ce modèle travaille implicitement avec une population infinie, mais il faut qu'on différencie entre les populations infinies, les populations finies, et les populations avec une taille raisonnable.

2.6 Conclusion

Les algorithmes évolutionnaires constituent une classe de méthodes approchées adaptables à un très grand nombre de problèmes combinatoires et de problèmes d'affectation sous contraintes. Ils ont révélé leur grande efficacité pour fournir des solutions approchées de bonne qualité pour un grand nombre de problèmes d'optimisation classiques et d'applications réelles de grande taille. C'est pourquoi l'étude de ces méthodes est actuellement en plein développement.

Si on peut constater la grande efficacité des algorithmes évolutionnaires pour de nombreuses classes de problèmes, il existe en revanche très peu de résultats permettant de comprendre la raison derrière cette efficacité. Cette question constitue sans doute un défi important pour les chercheurs.

Afin de résoudre des instances de taille et de difficulté croissantes, il faut mettre au point des méthodes toujours plus puissantes. Pour atteindre cet objectif, au moins deux voies privilégiées se développent : *l'hybridation de méthodes* et *la parallélisation* [89].

Le chapitre suivant sera consacré pour les études existantes dans le domaine de la parallélisation des algorithmes évolutionnaires sous les GPU ainsi que les techniques utilisées pour les optimiser.

Chapitre 3

Évolution des architectures parallèles classiques et modernes dans la parallélisation des algorithmes évolutionnaires

3.1 Introduction

L'informatique a connu une évolution importante ces dernières années avec la généralisation du concept de parallélisme. Le parallélisme s'est imposé au niveau de nombreuses architectures, langages et modèles de programmation parallèle qui ont été proposés au cours des deux à trois dernières décennies, et ce, pour aider à surmonter les limites avérées des processeurs standards. L'intérêt porté à ces architectures avait basculé récemment compte tenu des limites de performances des processeurs séquentiels. L'évolution est telle que, les processeurs séquentiels sont devenus de plus en plus efficaces faisant perdre momentanément aux chercheurs leur intérêt pour des machines parallèles complexes. Cependant, la stagnation récente de la fréquence d'horloge pour ces mêmes processeurs a fait regagner de plus en plus leur intérêt à l'utilisation des architectures parallèles [188].

Les constructeurs de cartes graphiques ont également eu recours à offrir la possibilité d'exploiter les performances de ces cartes à des fins de calcul généraliste : on parle alors de « General Purpose Graphic Processing Unit » ou GPGPU. Ainsi, de nouvelles plateformes comme CUDA « Compute Unified Device Architecture » ont été conçus pour utiliser le potentiel de ces cartes graphiques à des fins de calcul généraliste, et ce, du fait que ces mêmes cartes incarnent une puissance de calcul parallèle sans commune mesure ayant des coûts nettement inférieurs.

L'enjeu principal de la modélisation et de la simulation de l'évolution continue se mesure en termes de contraintes et d'objectifs. Cependant, l'exécution de cette évolution ne cesse d'exiger un temps de calcul exorbitant, allant jusqu'à plusieurs jours de calcul. Ceci est renforcé par le fait que ces modèles complexes peuvent également avoir plusieurs paramètres supplémentaires, qui doivent être considérés pour leur rôle important dans le comportement du modèle. Pour ces raisons, des efforts considérables ont été consentis dans la théorie ainsi que dans les technologies et les méthodologies avancées à l'effet d'accélérer l'exécution sans perdre la précision du modèle [20]. Néanmoins, le constat fait qu'il y avait peu d'études qui se sont intéressées à accélérer la simulation des applications faisant partie de l'intelligence artificielle en utilisant les GPUs.

La parallélisation des AEs n'est pas une nouvelle idée car les problèmes complexes requièrent de larges populations dont le traitement qu'on leur réserve est du même type. En effet, les AEs qui optimisent ces types de problèmes cherchent dans l'espace de solutions d'une manière efficace si un grand nombre d'échantillons est disponible (individus dans la population). On se voit donc confronté à un problème ayant deux volets : on ne dispose pas de puissance de calcul infinie, d'une part et d'autre part l'utilisation et le traitement de grandes populations conduit à des temps de calcul prohibitifs [124].

Alors que les unités de traitement informatique (CPUs) aient été optimisées pour exécuter des tâches séquentielles, les GPUs sont destinés à calculer et à exécuter des tâches massivement parallèles. De plus, les progrès récents ont contribué à l'apparition d'une nouvelle approche de programmation basée GPUs. Ainsi, de nombreux algorithmes ont été réécrits et repensés pour les GPUs modernes, qui sont caractérisés par le paradigme de programmation massivement parallèle SIMD « Single Instruction Multiple Data ». Par ailleurs, les GPUs ont été utilisés dernièrement pour accélérer des calculs dans divers sujets (en Neuroscience [117], dans les algorithmes évolutifs [97], dans le traitement d'images médicales [47], dans la compression de données [146], etc.)

Nous pouvons également noter que la disponibilité de nouveaux environnements de développement appropriés tels que CUDA et OpenCL tend à simplifier davantage le développement d'applications parallèles pour ce type de processeurs, et d'ouvrir, par la même, la possibilité de développer des modèles bio-inspirés prédictifs plus intégratifs et détaillés tout en diminuant en même temps le coût de calcul nécessaire pour simuler ces modèles.

Toutes ces considérations ont orienté de nombreux chercheurs vers les accélérateurs graphiques en détournant leur rôle vers le calcul généraliste afin d'optimiser des problèmes complexes. L'étude présentée dans la suite du document s'intéresse à l'utilisation des GPUs pour l'accélération des techniques évolutionnaires.

Dans ce chapitre, nous tentons d'élaborer un aperçu sur les architectures parallèles modernes et avancées ainsi que les modèles parallèles des AEs pour accélérer le processus de recherche. Nous introduisons, dans la suite de notre logique, le *GPU Computing* comme outil pour l'accélération des AEs. En outre, nous présentons les différents avantages et les défis associés à cette technologie émergente avant d'établir un examen et une analyse approfondis des différents travaux de la littérature qui se sont intéressés à l'implémentation des AEs sur des architectures GPUs.

Partie 1 :
Parallélisme et évolution des architectures parallèles classiques et modernes

3.2 Parallélisme : phases et définitions

Le calcul parallèle/distribué signifie que plusieurs processus fonctionnent simultanément et en parallèle sur plusieurs processeurs en résolvant une instance d'un problème donné. Le parallélisme suit une décomposition de la charge de calcul total et une distribution des tâches résultantes sur les processeurs disponibles. La décomposition peut avoir une liaison avec *l'algorithme*, *les données de l'instance du problème*, ou *la structure du problème*.

Dans le premier cas, dit parallélisme fonctionnel, des tâches différentes travaillant sur les mêmes données, s'exécutent en parallèle et sont allouées aux différents processeurs et exécutées en parallèle, avec la possibilité d'échange d'information.

Le deuxième type de parallélisme désigne le parallélisme des données ou le domaine de décomposition, et se réfère au cas où le domaine du problème, ou l'espace de recherche associé seraient décomposés et une méthodologie de solution particulière soit utilisée pour traiter le problème sur chacun des composants résultant de l'espace de recherche.

Pour ce qui du troisième cas, qui est en outre le plus récent, il génère des tâches en décomposant le problème en ensembles d'attributs. La décomposition peut être réalisée grâce à des techniques de programmation mathématiques ou grâce à des heuristiques. Consécutivement, certaines tâches travaillent sur des sous-problèmes correspondant à des ensembles particuliers d'attributs, alors que d'autres combinent des solutions des sous-problèmes aux solutions entières du problème original. Selon la façon dont «petit» et «grand» constituent des tâches en termes de travail de l'algorithme ou de l'espace de recherche, la parallélisation est dite 'fine' ou 'à gros grains', respectivement [27].

3.3 Architectures parallèles modernes et avancés : classification, modèles et langages de programmation

3.3.1 Classification des architectures classiques

Il existe plusieurs architectures parallèles différentes. En fonction du nombre d'instructions qui peuvent être exécutées simultanément et du nombre de flux de données sur lesquelles ces instructions peuvent agir, une taxonomie très connue proposée par Flynn en 1966, permet de regrouper ces architectures en quatre groupes : *Single Instruction / Single Data (SISD), Single Instruction / Multiple Data (SIMD), Multiple Instruction / Single Data (MISD), Multiple Instruction / Multiple Data (MIMD)* [124].

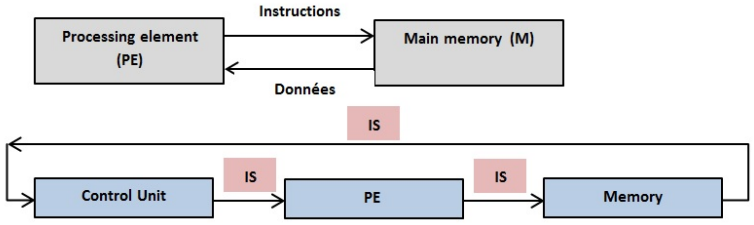


FIGURE 3.1 – SISD : Single Instruction Single Data.

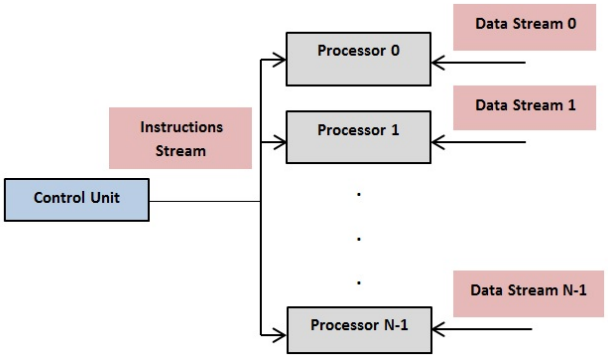


FIGURE 3.2 – SIMD : Single Instruction Multiple Data.

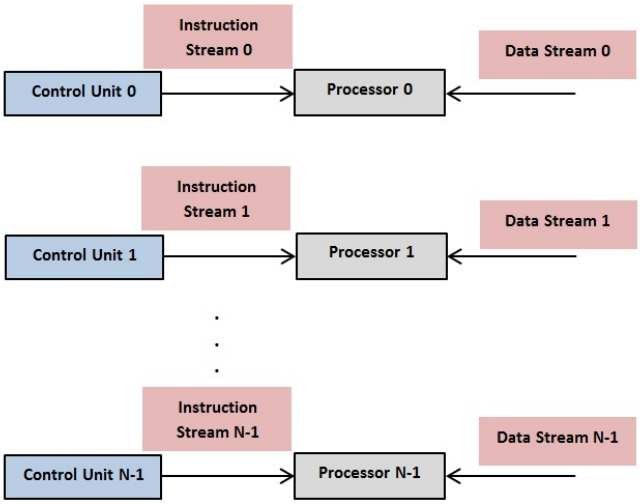


FIGURE 3.3 – MISD : Multiple Instruction Single Data.

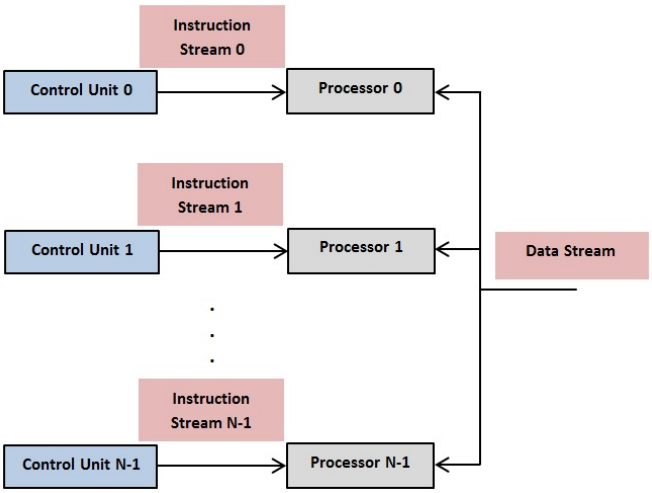


FIGURE 3.4 – MIMD : Multiple Instuction Multiple Data.

D'un côté, cette classification est universellement acceptée, utilise des notations brèves, et permet une classification facile d'un système. D'un autre côté, avec cette classification, une comparaison des différents systèmes est limitée, les interconnections, les entrées/sorties, les mémoires ne sont pas considérées dans le schéma. Par conséquent, il est rare de classifier des architectures complexes modernes comme l'une des catégories de Flynn [167].

En effet, la plupart des architectures modernes mettent en œuvre plusieurs de ces paradigmes, comme les processeurs multi-coeurs modernes. Ceux-ci peuvent être de type MIMD, en raison de leur implémentation multi-coeurs, mais peuvent être également de type SIMD en raison de leurs pipelines d'instructions et leur jeu d'instructions de vecteur [124], [167].

Handler a proposé une autre classification basée sur une notation élaborée pour exprimer le pipelining et le parallélisme, en adressant l'ordinateur à trois niveaux différents : Unité de contrôle du processeur (CPU), Unité arithmétique et logique (ALU), Circuit au Niveau du Bit (BCL). Cette classification est fortement orientée vers la description des pipelines et des chaines. Bien qu'elle soit en mesure de décrire le parallélisme dans un seul processeur, la variété du parallélisme dans les ordinateurs multiprocesseurs demeura mal traitée [84].

Comme un ordinateur parallèle MIMD peut être caractérisé par un ensemble de processeurs et de mémoire partagée ou des modules de mémoires communicants via un réseau d'interconnexion, une autre manière de classer les architectures parallèles fut proposée en considérant l'organisation, la mémoire et la communication inter-processus. Les plus répandues de nos jours sont l'accès uniforme à la mémoire (UMA), l'accès non uniforme à la mémoire (NUMA) et aucun accès à distance à la mémoire (NORMA). Dans l'accès de type UMA, la mémoire principale est uniformément partagée par tous les processeurs dans le système multiprocesseur et tous les processeurs disposent d'un temps égal pour l'accès à la mémoire partagée. Ce modèle est utilisé pour les applications multi-utilisateurs. Pour ce qui est du modèle NUMA, chaque processeur ou groupe de processeurs dispose de sa propre mémoire locale. L'ensemble de toutes les mémoires locales forme la mémoire globale qui est en fait partagée. De cette manière, la mémoire globale est distribuée sur tous les processeurs. Dans ce cas, l'accès à une mémoire locale est uniforme pour le processeur correspondant. Fournir des mémoires séparées pour chaque processeur réduit l'impact sur les performances qu'on peut observer pour les architectures de l'UMA lorsque plusieurs processeurs tentent d'accéder à la mémoire en même temps. Ce gain dépend de la localisation des données et est proportionnel au nombre de processeurs lorsque toutes les données se trouvent dans la mémoire locale. Cette architecture est plus évolutive que celle de l'UMA et permet d'associer un plus grand nombre de processeurs [167], [84], [81].

D'autres classifications sont possibles telles que celle basée sur l'arrangement des mémoires, la communication, et le type de parallélisme.

3.3.2 Environnements et langages de programmation

Dans un environnement parallèle, le programmeur doit gérer les problèmes liés aux ressources disponibles importantes. Les bibliothèques de programmation parallèle existantes offrent une interface de haut niveau pour l'architecture sous-jacente. Cette couche d'abstraction peut également inclure d'autres fonctionnalités inter-logiciel, comme le partitionnement des tâches, l'équilibrage de charge, et la communication des données ; qui

peut différer d'un environnement parallèle à l'autre selon le modèle cible de programmation et le matériel utilisé. Par conséquent, les modèles de programmation parallèle peuvent être classés, selon la façon dont les données sont gérées, en : *passage de message*, *parallélisme de thread*, et *parallélisme de données* [102].

Les applications de passage de message contiennent un ensemble de processus communicants indépendants qui sont nommés uniquement et interagissent en recevant et envoyant des messages les uns aux autres. Les spécifications les plus populaires pour le passage de messages sont la spécification PVN (Parallel Virtuel Machine) et la spécification MPI (Message Passing Interface). Les deux types de systèmes ont été conçus pour être portables et supporter les architectures hétérogènes. Le modèle de parallélisme de thread, également appelé le parallélisme de tâche de la mémoire partagée, est un modèle où différentes tâches sont exécutées par un même processus, en partageant le même espace d'adressage. Selon le point de vue des programmeurs, le principal avantage de ce modèle sur le passage de messages est de permettre l'accès aux données partagées de la même manière que le langage de programmation séquentielle. OpenMP est la norme la plus populaire pour la programmation de la mémoire partagée [102], [81].

3.4 Émergence des GPUs comme un outil de calcul général pour les méthodes parallèle bio-inspirés

3.4.1 Pourquoi les processeurs multi-coeurs et les GPUs ?

Les problèmes numériques les plus complexes sollicitent de plus en plus de puissance de calcul et les traitements à opérer sur les données sont de plus en plus exigeants. Les GPUs sont attractifs de par leur grande puissance de calcul théorique et leur coût modeste, ainsi que leur faible consommation liée à un système de refroidissement efficace [51].

En 2006, pour la première fois depuis leur invention, en raison des limites de dissipation de réchauffement, les processeurs ont atteint la limite des fréquences de fonctionnement possible. Afin de fournir des puces plus puissantes, les fabricants ont ensuite commencé à développer les processeurs multicoeurs, un chemin qui avait déjà été emprunté par les fabricants de cartes graphiques plus tôt. En 2012, NVIDIA a sorti les processeurs GK110 bénéficiant de 2880 coeurs avec une simple précision et 960 coeurs de double précision, pour une puissance de calcul de 6 Tflops en précision simple et 1.7 Tflops en précision double [188].

Actuellement, les supercalculateurs sont pourvus généralement avec de millions de coeurs dans les processeurs graphiques dédiés au calcul général, ce qui pose deux autres questions [18] :

- Quel genre d'applications peut exploiter la puissance de calcul des GPUs pour l'accélération ?
- Comment concevoir un GPU capable d'exécuter efficacement aussi bien les tâches graphiques que les tâches généralistes ?

FIGURE 3.5 – Les différents séries de cartes graphiques NVIDIA.

La partie suivante sera consacrée pour répondre à la deuxième question, en expliquant l'architecture des GPUs actuels (Hardware), ainsi que les modèles de programmation valable pour ces architectures (software).

3.4.2 Panorama matériel et logiciel des GPUs

Les architectures parallèles et hétérogènes sont de plus en plus parmi les courants domaines dominants et attractifs. Les accélérateurs graphiques disposent d'une performance de crête théorique de l'ordre du *TFlops*, exploitable dans le domaine du GPU Computing en utilisant les environnements de programmation associés, dans des domaines d'application très variés [140].

Actuellement, les GPUs peuvent contribuer efficacement dans un champ d'application plus large que le rendu graphique, dont l'exploitation de parallélisme de données est le point commun majeur entre ces applications. Un certain nombre de GPUs commerciaux sont disponibles sur le marché, telles que les gammes des constructeurs NVIDIA, AMD et Intel. Chacun d'entre eux possède différentes séries, telles que la GeForce, la Quadro, et la Tesla de NVIDIA, la Radeon, la FireGL et la FirePro/FireStream d'AMD et la GMA d'Intel. La connaissance approfondie de l'architecture matérielle des GPUs ainsi que leurs environnements de développement permet de mieux les exploiter, parmi ces environnements, on peut citer : CUDA pour les cartes NVIDIA ; ATI Stream SDK pour AMD/ATI et l'API "généraliste" OpenCL [81].

Naturellement, le nombre de coeurs du processeur continuera de croître proportionnellement à l'augmentation des transistors disponibles. Malgré leurs hautes performances démontrées sur les applications de données parallèles, les processeurs fondamentaux GPUs sont encore relativement de conception simple. Plus de techniques agressives seront introduites avec chaque architecture successive pour augmenter l'utilisation réelle des unités de calcul [188].

Du fait que le calcul parallèle évolutif sur les GPUs est encore un jeune domaine, on assiste à l'émergence rapide de nouvelles applications. En les étudiant, les concepteurs des GPUs pourront découvrir et mettre en oeuvre de nouvelles optimisations pour ces GPUs.

Le but de cette partie est la présentation des principales caractéristiques de l'écosystème GPU actuel, en introduisant l'historique, les architectures des cartes graphiques NVIDIA ainsi que les défis de programmation pour un tel hardware.

3.4.3 Développement du software GPU au cours de ces différentes générations : Historiques vs Actualités

Les techniques matérielles du pipeline graphique tridimensionnel (3D) a évolué comparativement aux grands systèmes coûteux du début des années 1980 et aux petites stations de travail et PC accélérateurs du milieu et la fin des années 1990 [102].

Depuis l'apparition des accélérateurs graphiques dans les années 80, leur évolution n'a cessé de s'accélérer [18]. Le terme GPU est introduit par NVIDIA, remplaçant le terme *VGA Controller* qui est insuffisant pour exprimer l'ensemble des fonctionnalités offertes par ces cartes.

La compréhension de l'héritage graphique de ces processeurs éclaire les forces et les faiblesses de ces processeurs à l'égard des principaux modèles de calcul. En particulier, l'histoire permet de clarifier le raisonnement derrière les grandes décisions de conception architecturale des programmables GPUs modernes : MultiThreading massif, les mémoires caches relativement petites par rapport aux unités centrales de traitement (CPUs), et la conception d'interface de mémoire à bande passante centrée. Un aperçu sur les développements historiques sera également susceptible de donner au lecteur le contexte nécessaire pour projeter l'évolution future des GPUs en tant que dispositifs de calcul [29]. Par la suite, nous allons présenter un bref historique de l'évolution de ces dispositifs ainsi que le développement des architectures actuelles.

Le calcul sous GPU a démarré comme un domaine de recherche pour les ordinateurs graphiques au début des années 2000 et a bénéficié d'une haute importance comme un processeur destiné vers le calcul généraliste. En 2001, pour la première fois, les unités de traitement graphiques ont été construites sur une architecture programmable, permettant à une multitude de programmes ([22], [23], [165]) d'être calculés et rendus en temps réel. Ces effets sont achevés en utilisant des langages à base de Shaders de haut niveau tel que GLSL (OpenGL Shading Language) [130], HLSL (Highlevel Shading Language) [130] et CG (C for Graphics) [37]. Durant la même période, l'idée du parallélisme massif a commencé à évoluer dans la société des chercheurs du domaine graphique (les chercheurs CG), qui ont conçu les algorithmes per-vertex et per-fragment destinés à un ensemble de millions de primitives. Durant les années passées, la communauté scientifique s'est intéressée à la capacité des GPUs et à leurs faibles coûts par rapport aux autres solutions (clusters, supercomputers). Tandis que, l'adaptation d'un problème scientifique à un environnement graphique était une tâche délicate et considérée d'abord comme un détournement du rôle du GPU, elle est devenue un grand challenge du point de vue technique.

En 2002, McCool et coll. ont publié un travail qui détaille un langage de méta-programmation dédiée au GPGPU nommé Sh [131]. En 2004, Buck et coll. ont proposé un autre environnement dit Brook-GPU [13]. Ceci était une extension du langage C qui permet la programmation générale sous les GPUs programmables. Ces deux langages (Sh et Brouk-GPU) ont joué un rôle très important dans l'élargissement de l'idée de calcul sur GPU (GPU Computing) en cachant le contexte graphique des langages shading.

En 2006, une autre API dédiée au calcul généraliste sous GPU a été réalisée. Mais cette fois-ci par NVIDIA, appelé CUDA (Compute Unified Device Architecture). Techniquement, L'API CUDA est une extension du langage C en compilant des utilitaires d'ordre général, qui sont exécuté sur GPU (en se basant sur le modèle de programmation à mémoire partagée). L'avènement de CUDA était un vrai important repère dans l'histoire de ce type de programmation puisqu'elle était la première API qui a délivré une docu-

mentation complète pour le démarrage dans ce paradigme de programmation. L'acronyme CUDA réfère au calcul généraliste des GPUs NVIDIA [26]. Actuellement, seule NVIDIA supporte CUDA.

En 2008, un standard a été créé avec OpenCL (Open Computing Language), permettant la création de codes massivement parallèles, pour des plateformes multiples. Son modèle de programmation est similaire de celui de CUDA, avec différents noms pour les mêmes structures. Le modèle de programmation derrière CUDA et OpenCL est un aspect clé pour la programmation GPU, tant ils définissent la majorité des composants essentiels pour l'implémentation d'un algorithme massivement parallèle [72].

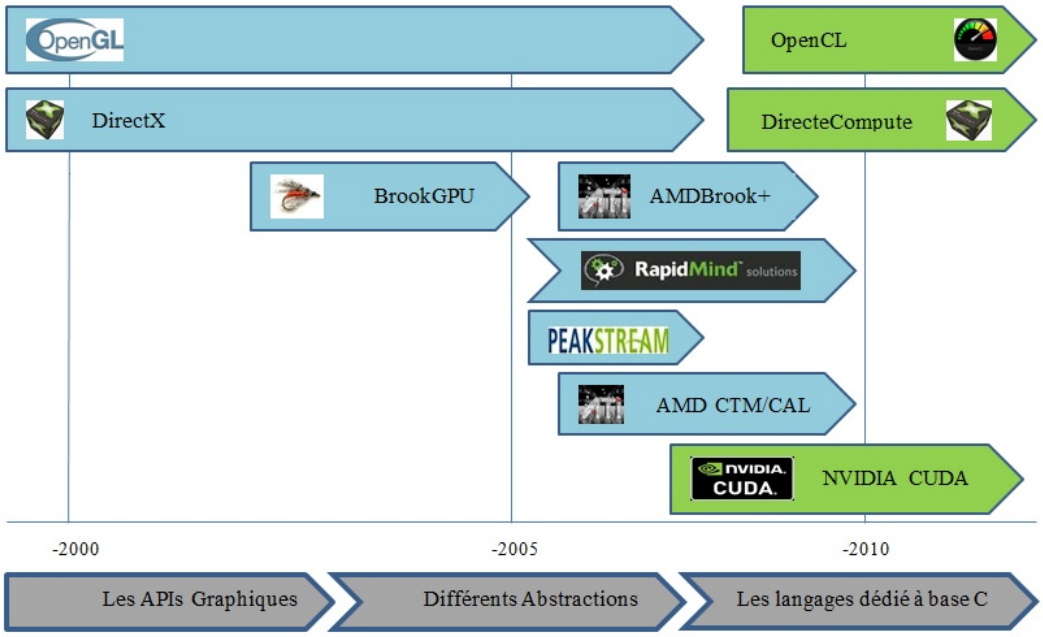


FIGURE 3.6 – La chronologie des languages au cours des années.

3.4.4 Futur du calcul GPU

Le calcul sous GPU a pris une évolution extraordinaire depuis l'introduction de CUDA. Le matériel (Hardaware) a évolué en un temps incroyable, offrant de nouvelles caractéristiques, améliorant la performance et le prix pour chaque génération.

Le logiciel a progressé également pour refléter ces changements, à partir de la version initiale de CUDA 1.0 à la version actuelle CUDA 7.0 rendant le calcul parallèle plus facile. Ces progrès ont créé une forte position du calcul sous GPU dans le monde du calcul haute performance (High Performance Computing - HPC), représenté par l'un des supercalculateurs les plus puissants du monde nommé Titan, qui est composé de 18 688 processeurs avec le même nombre de GPUs Tesla K20s, ce qui permet d'atteindre une performance totale de 27 PFLOPS. Récemment, Le Titan a été dépassé par le supercalculateur Tianhe-2, qui est composé de 16 000 noeuds informatiques (ordinateurs), chacun avec deux processeurs Intel lvy Bridge Xeon et trois puces Xeon Phi, ajoutant jusqu'à un total de 3.120.000 coeurs et 33.86 PFLOPS.

NVIDIA a travaillé également à étendre le calcul haute performance pour les appareils mobiles. Cet effort a donné lieu à une nouvelle gamme de puces codées nommées Tegra,

ayant servi dans de nombreuses différentes tablettes et téléphones mobiles. Une puce Tegra comprend un processeur ARM ainsi qu'un nombre de GPUs coeurs, qui pourraient être non programmable de la même manière que les GPUs standards tels que les cartes de GTX ou Tesla. Cependant, cela a changé en 2014, lorsque NVIDIA a annoncé une puce complètement redessinée nommée Tegra K1, comportant Kepler. L'introduction de Tegra K1 a redéfini le marché du mobile et a permis de nombreuses nouvelles fonctionnalités sur les appareils mobiles tels que DirectX 11.2, OpenGL 4.4 et CUDA 6.0.

L'avenir du calcul sous GPU semble également brillant avec la prochaine génération Maxwell, qui s'appuie sur le succès de Kepler.

La puce Maxwell GM117 comprendra un processeur Denver ARM 64 bits dual core intégré, qui décharge une partie du travail de GPU telle que les calculs PhysX. Un autre ajout inclut la mémoire virtuelle unifiée permettant au CPU d'accéder à la mémoire du GPU et vice versa. Cette étape sera suivie par une prochaine génération nommée Pascal, qui introduit deux technologies clés qui mettent l'accent sur l'apprentissage de la machine. La première est appelée « NVlink », qui renforcera la vitesse du bus PCIe de 5 à 12 fois par l'amélioration de la communication entre le CPU et le GPU. La deuxième technologie clé est une mémoire 3D, qui est fondamentalement une DRAM empilée. Cette nouvelle mémoire va avoir 4 fois plus efficace et une bande passante 5 fois plus élevée de l'ordre de 1 TB/sec.

3.4.5 Architecture matérielle et logiciel du GPU

Conduit par la demande pour des graphiques 3D à haute définition sur des ordinateurs personnels, les GPUs ont évolué en un environnement fortement parallèle, multithreads et multicoeurs [132]. Dans la suite du document, une vue générale de l'architecture matérielle et logicielle sera présentée.

3.4.5.1 Vue générale de l'architecture matérielle

Le processeur GPU (Graphics Processing Unit) orienté débit représente une tendance majeure dans l'avancement récent des architectures pour l'accélération du calcul parallèle. Effectivement, cette architecture offre une puissance de calcul énorme et une bande passante de mémoire très élevée par rapport aux CPUs traditionnels. Puisque plus de transistors sont dévoués au traitement des données plutôt qu'aux caches de données et aux flots de contrôle, le GPU est spécialisé pour des applications fortement parallèles et intenses en termes de calcul. Une revue complète sur l'architecture du GPU peut être consultée dans [140], [102], [188].

La plus évidente caractéristique du GPU est la disponibilité d'un très grand nombre de coeurs assez simples au lieu de quelques coeurs complexes comme les processeurs classiques multicoeurs de bureau à usage général.

Du point de vue physique, les GPUs NVIDIA sont organisés comme des Streaming Multiprocessors (SM : multiprocesseurs de traitement de flux) embarquant des processeurs scalaires élémentaires (SP : Scalar Processors). Les SMs utilisent la mémoire du périphérique comme une ressource partagée. Chaque SM contient par ailleurs un nombre important de registres, utilisés pour stocker les opérandes des instructions [140].

Par exemple, pour chaque type de GPU envisagé, il convient donc de tenir compte de ces spécificités en termes de répartition des unités de calcul et de modèle d'exécution, ainsi que de la hiérarchie mémoire.

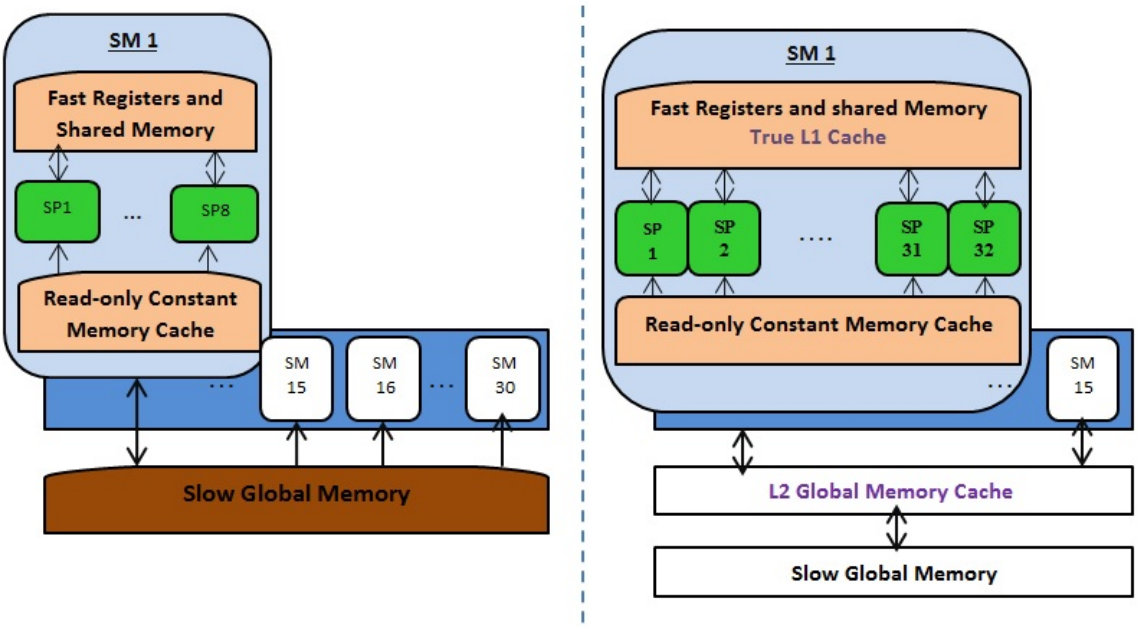


FIGURE 3.7 – Une représentation simplifié de l'architecture GPU de NVIDIA GTX 280 (à gauche) et GTX 480 (à droite).

3.4.5.2 Vue générale de l'architecture logicielle

Comme nous l'avons indiqué dans la partie matérielle, l'architecture du GPU varie considérablement à chaque génération. En outre, même dans une génération, la considération de la puce (en particulier, le nombre de coeurs) peut être modifiée. Ceci peut entrainer des complications lors du portage des programmes à différentes architectures ciblées. Pour pallier ce problème, NVIDIA a développé un logiciel appelé CUDA qui fournit une abstraction du matériel sous-jacent, ce qui facilite la portabilité, même à travers les différentes architectures et la compréhension du processeur sous-jacent.

Dans le paradigme GPGPU, le CPU est considéré comme l'hôte et le GPU est utilisé comme un coprocesseur périphérique. De cette manière, le GPU a sa propre mémoire et ses éléments à traiter qui sont séparés de l'ordinateur hôte. Les données doivent être transférées entre l'espace mémoire de l'hôte et la mémoire du GPU pendant l'exécution des programmes. Le transfert de la mémoire du CPU vers la mémoire du GPU est une opération synchrone qui est coûteuse en termes de temps de calcul. Le bus de la bande passante et la latence entre le CPU et le GPU peuvent diminuer significativement les performances de la recherche. Ainsi, ces transferts de données doivent être minimisés [132].

Les opérations séquentielles doivent être programmées comme des fonctions d'hôte qui s'exécutent sur le CPU. Par conséquent, les opérations parallélisables devraient être programmées comme des Kernels ou des fonctions du dispositif qui s'exécutent sur le GPU. Les deux fonctions d'hôte et de dispositif seront appelées via une fonction principale de l'hôte.

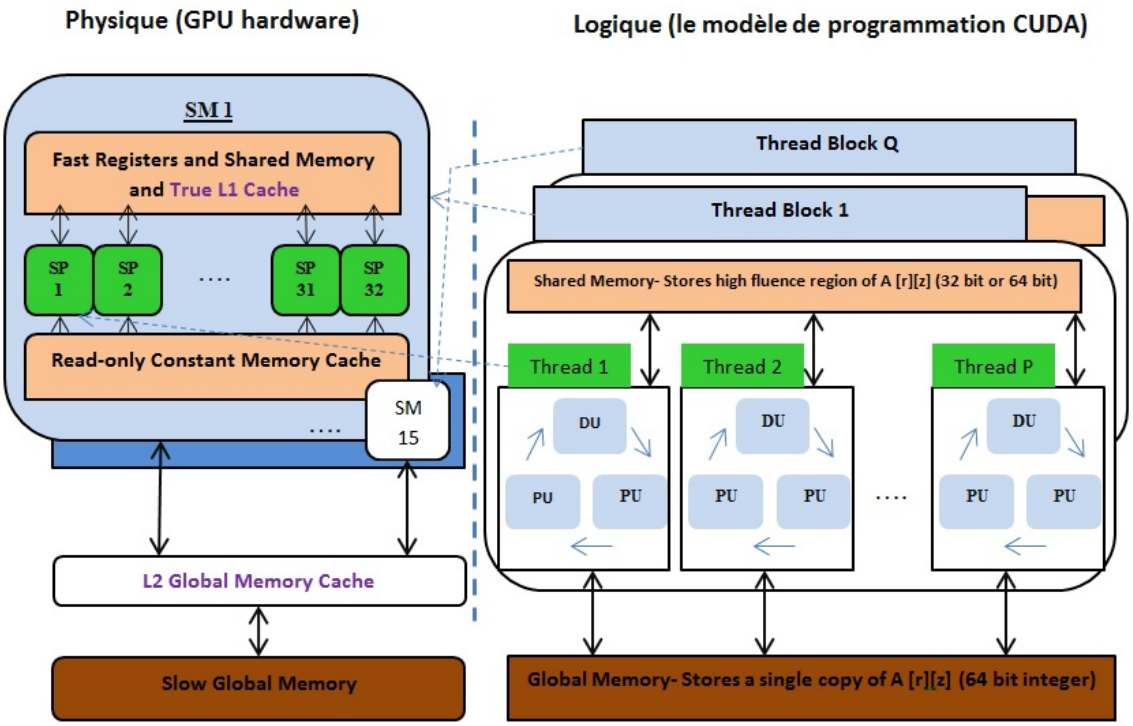


FIGURE 3.8 – La correspondance entre la physique et la logique de l'architecture GPU Nvidia avec son environnement de programmation CUDA, où DU représente *Direction Update*, PU représente *Position Update*, et FU représente *Fluence Update*.

Dans l'environnement CUDA, l'unité de base d'un code parallèle est le Thread, des milliers de Threads peuvent s'exécuter simultanément avec le même ensemble d'instructions appelé Kernel. Un Thread sur GPU peut être vu comme un élément de donnée à être traité. Comparés aux Threads sur CPU, les Threads sur GPU sont légers. Ce qui signifie que changer le contexte entre deux Threads n'est pas une opération coûteuse. Un Kernel peut employer les registres comme une mémoire à accès rapide. La communication entre les Threads peut être effectuée avec la mémoire partagée, qui est une sorte de mémoire très rapide surtout pour les opérations d'accès de lecture et d'écriture.

Concernant l'organisation spatiale de Threads, les Threads sont organisés à l'intérieur de blocs de Threads. Un Kernel est exécuté par de multiples blocs de Threads de même taille. Les blocs peuvent être organisés en des grilles à une, deux, ou trois dimensions de blocs de Threads, et les Threads à l'intérieur d'un bloc peuvent être regroupés de manière similaire. Tous les Threads appartenant à un même bloc peuvent être affectés à différents multiprocesseurs.

La communication entre le CPU et le GPU peut se faire par la mémoire globale, la mémoire constante, ou la mémoire de texture du GPU. Par ailleurs, il y'a lieu de noter que les dernières générations de GPUs contiennent une mémoire accessible par les deux périphériques.

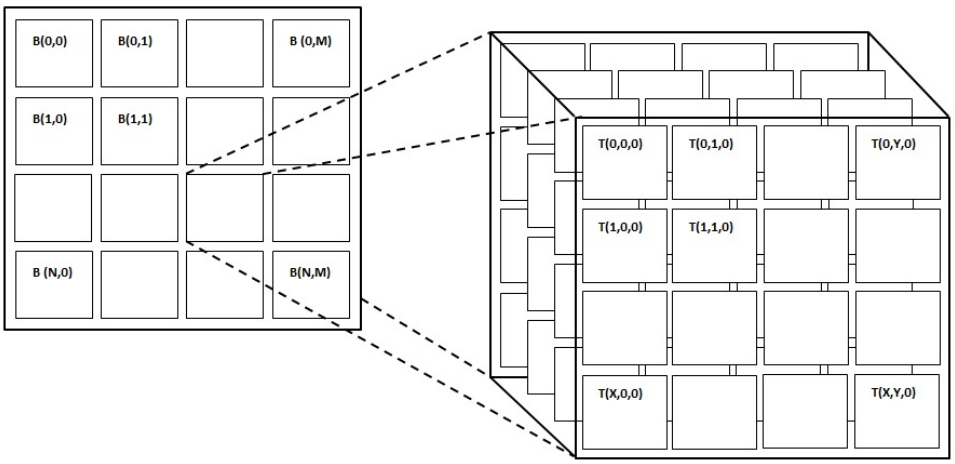


FIGURE 3.9 – L'organisation spatiale de threads.

Après la compilation par l'environnement CUDA, un programme s'exécute comme un Kernel sur le GPU. Un Kernel prend les paramètres d'entrée, effectue les calculs, et renvoi les résultats vers la mémoire du GPU où les résultats peuvent être lus par le CPU. Chaque Thread doit effectuer la même opération dans le Kernel, mais les données d'entrée peuvent être différentes. Avec des milliers de Threads qui font des tâches similaires simultanément, la vitesse de calcul peut s'avérer significativement améliorée. Le CPU est le propriétaire du code hôte qui prépare des données d'entrée et accepte des valeurs de sortie provenant du GPU. La tâche de calcul intensif est gérée par les Kernels GPU. Les données de sortie sont écrites dans la mémoire globale du périphérique (GPU) afin d'être récupérées par le programme du CPU. Pour une description très complète, le lecteur est renvoyé à consulter le guide de programmation CUDA C [26] et des ressources similaires.

Nous pouvons résumer les aspects clés de la haute performance d'un code GPGPU comme ils l'ont été dans l'étude réalisée dans le travail de (yang et al,[205]) :

1. Les accès mémoire doivent être coalescés et chaque élément de données peut avoir besoin d'être un type de vecteur ;
2. L'utilisation commune d'une mémoire partagée se fait par un logiciel qui gère la réutilisation de la mémoire ;
3. Les balances du parallélisme et les optimisations de la mémoire doivent être faites avec soin, comme plusieurs Threads dans le même bloc ainsi que de multiples blocs sont en concurrence pour les ressources limitées dans un SM, y compris le fichier de registre, la mémoire partagée, et le nombre de Threads étant pris en charge dans le matériel ;
4. Des milliers de Threads doivent être exécutées simultanément sur le GPU pour dissimuler la latence de la mémoire ;
5. La divergence de Threads doit être réduite au minimum par l'exécution de tous les Threads au sein d'un Warp en suivant le chemin d'exécution.

3.5 Applications GPGPU pour la vie artificielle.

Cette section fournit une vue générale concernant l'application des GPUs dans quelques domaines de recherche dans la vie artificielle, tels que *la bio-informatique*, *les réseaux de neurones artificiels*, *les L-Systèmes*, *le développement cellulaire*, *les algorithmes évolutionnaires*, *le traitement de la vision*, qui vont constituer la partie présentée par la suite.

3.5.1 Bioinformatique

Comme on pouvait s'y attendre, les GPUs ont été suggérés pour des applications médicales de traitement d'image depuis quelques années maintenant. Les chercheurs prévoient qu'après le portage de quelques algorithmes clés avec succès au GPU, dans quelques années, la bio-informatique adoptera le GPU pour l'accélération de plusieurs routines utilisées dans ces applications.

Charalambous et coll. [17] ont utilisé avec succès un GPU alimenté bas de gamme pour démontrer l'inférence d'arbres de succession évolutifs (par portage RAxML sur une carte graphique nVidia). La comparaison des séquences est la source vitale de la bio-informatique. Liu et coll. [118] ont exécuté l'algorithme clé de Smith-Waterman sur un GPU haut de gamme. Ils ont démontré une réduction d'un facteur allant jusqu'à taille (16) à l'époque et pour la plupart des protéines. Schatz et coll. [168] ont également utilisé CUDA pour porter une autre séquence d'outil de recherche (MUMmer) sur un autre GPU de la gamme G80 où ils ont obtenu une multiplication de la vitesse par un facteur allant de 3 à 10 lors de l'appariement des brins d'ADN courts contre des séquences plus longues. En brisant les requêtes des GPUs en fragments dimensionnés, ils étaient en mesure d'exécuter de courtes séquences (par exemple 50 bases) contre un chromosome humain complet. Gobron et coll. [60] ont utilisé OpenGL sur un GPU haut de gamme pour conduire une simulation d'automates cellulaires de l'œil humain et ont obtenu un traitement en temps réel d'une caméra d'entrée. Les GPUs ont également été utilisés dans l'ingénierie médicale. Par exemple, une GeForce 8800 fournit un ordre d'accélération de 15 à 20, améliorant la réponse haptique d'un outil de simulation de chirurgie interactive en temps réel [119].

3.5.2 Réseaux de neurones artificiels et les algorithmes évolutionnaires

Les applications d'intelligence computationnelle des GPUs ont inclus les réseaux de neurones artificiels (par exemple. les perceptrons multicouches et l'auto-organisation des réseaux [120]), les algorithmes génétiques [52], les stratégies d'évolution, et la programmation génétique [114], [158], [76], [75], [77], [110], [111], [19], [112], [191].

3.5.3 L-Systèmes

Les systèmes de Lindenmayer populairement connu sous le nom de L-systèmes sont des systèmes de chaîne de réécriture qui consistent en un initiateur appelé axiome, et un ensemble de règles appelées générateur [154]. Afin d'accélérer la génération d'images avec les L-systèmes, des travaux de recherche ont été réalisés en direction de la dérivation des L-systèmes directement sur les unités de traitement graphique, qui permet d'examiner

si l'interprétation géométrique des L-systèmes peut être divisée en milliers de Kernels qui peuvent être implémentés en parallèle avec les Threads CUDA. Certains travaux ont analysé le parallélisme de la mise en oeuvre des L-Systèmes sur GPU [108]. Lipp et coll. [115] ont été les premiers à analyser la possibilité de la mise en oeuvre des L-système en utilisant CUDA.

3.5.4 Traitement de la vision artificielle

Généralement, les systèmes standards de vision nécessitent plusieurs traitements à faire en un temps minimum. Comme l'augmentation de la résolution des caméras ainsi que la complexité des algorithmes de traitement d'images, il s'avère que le CPU est en retard par rapport au GPU en termes de performances (p. ex. Pulli et coll. 2012 [155]) et pourrait être lent pour aborder le contrôle des tâches en temps réel d'un robot.

Avec l'introduction des outils de programmation généraliste tels que CUDA et OpenCL, plusieurs API de traitement de vision ont été élaborées, telles que OpenCV (Pulli et coll. 2012, [155]), OpenVIDIA(Fung and Mann 2005, [54]), gpuCV (Patil and Shahapure 2013, [147]) ou NVIDIA Performance Primitives (NPP) library [36]. Cela active le développement rapide de nouvelles applications en traitement d'image dans la vision artificielle des robots (Folkers and Ertel 2007, [53]), telle que la vision stéréo (Lyes and Hawick 2011, [122]), le flux optique (Kim et coll. 2009 [101]), la détection de saillance (Thota et al. 2013, [184]), la reconstruction 3D (Gallup 2011, [55]), la fonctionnalité de suivi (Sinha et coll. 2006, [174]), la détection d'objets (Coates et coll. 2009, [21]), la détection de visage (Devrari and Kumar 2011, [34]) et la détection pose corps (Brown et coll. 2011, [12]).

La tendance générale est que ces algorithmes sont bien adaptés pour l'architecture GPU SIMD et s'exécutent significativement plus rapidement que l'implémentation CPU correspondante.

Partie 2 :
Modèles et paradigmes parallèles des algorithmes évolutionnaires

3.6 Modèles parallèles des algorithmes évolutionnaires

3.6.1 Sources de parallélisation des algorithmes évolutionnaires

D'une part, les problèmes d'optimisation sont de plus en plus complexes et leurs besoins en ressources augmentent sans cesse. Les problèmes d'optimisation de la vie réelle sont souvent NP-hard en matière de consommation de ressources CPU et/ou mémoire. Bien que l'utilisation des méta-heuristiques permette de réduire considérablement la complexité de calcul du processus de recherche, ce dernier reste consommateur de temps pour de nombreux problèmes dans divers domaines d'application, où la fonction objective et les contraintes associées au problème sont des ressources (p. ex., CPU, mémoire) intensives et la taille de l'espace de recherche est très élevée [180].

D'un point de vue algorithmique, la principale source de parallélisme pour les AEs est l'exécution concurrente de leurs itérations de boucles intérieures. Malheureusement, c'est souvent la seule source facilement accessible dans les AEs, la plupart des autres étapes étant dépendantes du temps et nécessitant le calcul des étapes précédentes pour être achevées. Même lorsque le parallélisme est disponible, la synchronisation dépendant du temps des étapes de l'AE donne des latences importantes, ce qui rend le calcul parallèle non pertinent [27].

Un AE contient plusieurs étapes qui peuvent être indépendantes ou non. Pour commencer, l'initialisation de la population est intrinsèquement parallèle, car tous les individus sont créés de façon indépendante (généralement avec des valeurs aléatoires).

Ensuite, tous les individus nouvellement créés doivent être évalués. Mais, comme ils sont tous évalués de manière indépendante en utilisant la fonction de fitness, l'évaluation peut également être faite en parallèle. Il est intéressant de noter que dans les AEs, l'évaluation des individus est généralement l'étape la plus consommatrice de l'AE.

Une fois la population parente obtenue (par évaluation de tous les individus de la population initiale), il est nécessaire de créer une nouvelle population de descendants. Afin de créer un descendant, il est nécessaire de sélectionner quelques parents sur lesquels seront appliqués les opérateurs de variation. Dans les AEs, la sélection des parents est aussi parallélisée parce qu'un parent peut être sélectionné plusieurs fois, ce qui signifie que les sélecteurs indépendants peuvent choisir ce qu'ils veulent, sans aucune restriction.

La création d'un descendant hors des parents sélectionnés est également une étape totalement indépendante : un opérateur de croisement doit être appliqué sur les parents, suivi par un opérateur de mutation sur le descendant créé.

Ainsi, on peut constater que jusque-là, toutes les étapes de la boucle évolutionnaire sont intrinsèquement parallèles, sauf pour mais pour la dernière, le remplacement. Afin de préserver la diversité dans les générations successives, la génération $(N+1)^{eme}$ est créée en sélectionnant l'un des meilleurs individus de la population des enfants des parents + la population des enfants de la N^{eme} génération. Cependant, si un individu est autorisé à apparaître plusieurs fois dans la nouvelle génération, il pourrait rapidement devenir prééminent dans la population, induisant ainsi une perte de diversité qui aura pour conséquence de réduire la puissance d'exploration de l'algorithme [127].

Par conséquent, les AEs imposent que tous les individus de la nouvelle génération soient différents. C'est une véritable restriction sur le parallélisme, puisque cela signifie que la sélection des N survivants ne peut être faite de façon indépendante, sinon un même individu peut être sélectionné à plusieurs reprises par plusieurs sélecteurs indépendants.

D'autre part, une quantité significative du parallélisme peut se trouver dans le domaine

du problème à résoudre ou dans l'espace de recherche correspondant. En effet, il n'y a pas de dépendances de données entre le coût ou les fonctions d'évaluation de solutions différentes et, par conséquent, elles peuvent être calculées en parallèle. De plus, théoriquement, le parallélisme dans la solution ou l'espace de recherche est aussi grand que l'espace lui-même. Il y a des limites considérables à une exploitation efficace de ce parallélisme, cependant. Pour des raisons évidentes, attribuer un processeur pour chaque évaluation de la solution ne présente aucun intérêt. L'espace de solutions ou de recherche doit donc être partagé entre les processeurs, donc, une sérialisation de l'évaluation de solutions affectées au même processeur doit prendre place. Les partitions résultantes sont généralement encore trop grandes pour une énumération explicite et, par conséquent, une méthode exacte ou heuristique de recherche est toujours nécessaire pour l'explorer implicitement. Le partitionnement pose alors deux questions à l'égard de la stratégie globale de la recherche méta-heuristique [27] : le contrôle d'une recherche globale menée séparément sur plusieurs partitions de l'espace original et l'exhaustivité de la solution finalement atteinte. L'allocation des ressources informatiques doit mener à une exploration efficace évitant, par exemple, la recherche des régions avec des solutions de mauvaise qualité. Finalement, on pourrait se demander si plusieurs générations pourraient évoluer en parallèle, le fait que la génération (N+1) est basée sur la génération N infirme cette idée.

Le calcul parallèle et distribué peut être utilisé dans la conception et la mise en oeuvre de méta-heuristiques et par la suite les AEs pour les raisons suivantes [121] :

- Accélération de la recherche : Ceci est un aspect crucial pour la classe de problèmes où il y a des exigences précises sur le temps de recherche. Un des principaux objectifs de la parallélisation d'une méta-heuristique est de réduire le temps de recherche. Ceci aide à une conception interactive en temps réel des méthodes d'optimisation.

- Amélioration de la qualité des solutions obtenues : certains modèles parallèles pour les méta-heuristiques permettent d'améliorer la qualité de la recherche. En effet, l'échange d'informations entre les méta-heuristiques coopératives va modifier leur comportement en termes de recherche dans l'espace de recherche associé au problème. L'objectif principal d'une coopération parallèle entre les méta-heuristiques est d'améliorer la qualité des solutions.

- Amélioration de la robustesse : Une méta-heuristique parallèle peut être plus robuste en termes de résolution de différents problèmes d'optimisation et différentes instances d'un problème donné avec une manière efficace.

- Résolution de problèmes à grande échelle : les méta-heuristiques parallèles permettent de résoudre des instances à grande échelle pour des problèmes d'optimisation complexes. Un enjeu est de résoudre de très grandes instances qui ne peuvent être résolues par une machine séquentielle. Un autre défi similaire est de résoudre des modèles mathématiques plus précis associés aux différents problèmes d'optimisation.

3.6.2 Classification des Modèles Parallèles des AEs

Plusieurs recherches ont été élaborées en ce qui concerne la classification des méta-heuristiques, les plus fameuses sont celle de Crainic et Toulouse [24], qui ont considéré *les sources de parallélisation* dans ces techniques, et celle de Cung et coll.[27] qui se veut *indépendante de l'architecture*.

Le processus d'évolution artificielle peut être mis en oeuvre sur le matériel parallèle de diverses manières, où l'idée de base est à propos des individus de la population, qui sont indépendants les uns des autres, et seulement quelques étapes de l'algorithme qui nécessitent une interaction entre elles, principalement le croisement et la réduction de la population [124].

Selon Cantu-Paz [14] et Alba et Tomassini [1], Grefenstette [71] est le premier à avoir introduit plusieurs types de parallélisme des AEs, où il a mis en oeuvre un algorithme Maître-Esclave synchrone et un autre asynchrone (coarse-grained model). Ces deux catégories sont fondées principalement sur la structuration de la population.

Dans le modèle global, une population est maintenue et tous les processeurs accèdent au même groupe d'individus. Par contre, dans la deuxième catégorie, la population est structurée en sous-ensembles, cette dernière concerne les AEs parallèles distribués.

Les AEs cellulaires représentent une autre catégorie, qui pousse encore la structuration de la population, en considérant un individu et son voisinage immédiat comme une sous-population. Il est notable ici que ces catégories ne sont pas strictes, puisqu'il existe également des modèles mixtes, où plusieurs niveaux de parallélisme sont présents

3.6.2.1 AEs parallèles globales ou le modèle Maitre/Esclave

Approches standards

L'évaluation d'un individu particulier est en général, le cas le plus commun, et indépendamment de l'évaluation des autres individus. Sur une architecture parallèle, il est possible de répartir la population (parents ou descendants) en sous-ensembles et attribuer chaque sous-ensemble à un processeur pour être évalué. Cette parallélisation ne modifie pas le comportement de l'algorithme. Seul le temps d'évaluation est réduit en utilisant plusieurs processeurs pour évaluer la population.

La parallélisation en utilisant le modèle maître/esclave présente l'avantage d'être comparable à un algorithme séquentiel équivalent (l'évaluation des populations et des individus se fait d'une façon parallèle). Ce qui simplifie la comparaison entre les algorithmes séquentiels et parallèles, et le calcul de l'accélération en particulier peut être fait à la base d'une seule exécution. En outre, ce modèle n'impose pas une architecture de matériel spécifique, tant il existe des variantes sur de nombreux types de machines, qui sont soit partagées ou à mémoire distribuée (répartie), pour des processeurs MIMD ou SIMD.

Cependant, l'accélération réalisée avec un tel modèle est en fonction de la difficulté du problème et de l'architecture matérielle, puisqu'elle est limitée à la partie parallélisable de l'algorithme.

Cette méthode offre plusieurs avantages. Le premier est évidemment, comme on l'a noté ci-dessus, d'être tout à fait comparable avec les algorithmes séquentiels. Le second est la portabilité de ce modèle, qui peut être implémenté sur un grand nombre d'architectures parallèles. Comme dernier avantage, on peut citer l'absence de paramètres supplémentaires

par rapport aux approches séquentielles. En effet, seul le nombre de noeuds doit être spécifié, mais ça ne modifie pas le comportement de l'algorithme.

Toutefois, ce modèle a aussi des désavantages : c'est un modèle synchrone, où les noeuds travaillent *pas à pas*. Le noeud maître envoie séquentiellement les sous-populations aux noeuds esclaves faisant de son interface avec les noeuds d'esclaves *un goulot d'étranglement*. Ensuite, le noeud maitre fonctionne au ralenti, car il attend les noeuds esclaves pour calculer les valeurs de le fitness, même s'il est tout à fait possible de lui attribuer un sous-ensemble à évaluer. Encore une fois, l'interface de communication du noeud maître devient un goulot d'étranglement. Enfin, le noeud maître remonte à l'exécution de l'algorithme séquentiel standard, ce qui signifie que les noeuds esclaves sont au ralenti. Ainsi, il y a plusieurs synchronisations, opérations d'échange et d'attente dans ce modèle.

Les points critiques sont la vitesse de l'interface de communication du noeud maître, et le rapport de temps entre l'évaluation de la population et le reste de l'algorithme (moteur d'évolution).

Enfin, la parallélisation utilisant ce modèle permet de partager les ressources de calculs, mais ne prend pas en compte l'avantage de la taille de la mémoire distribuée parce que la population est toujours stockée sur le noeud maître. Cependant, elle permet d'utiliser une architecture asymétrique, où le noeud principal est une machine puissante avec un grand espace mémoire, et les noeuds esclaves sont plus petits avec moins de possibilités d'accès mémoire.

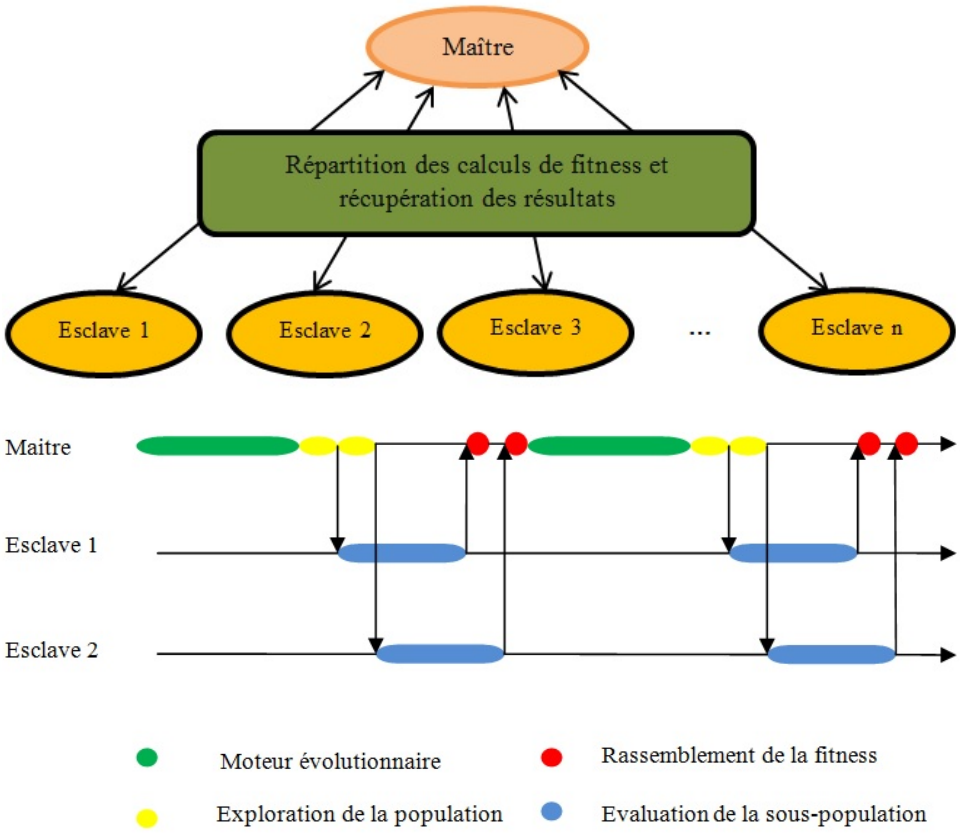
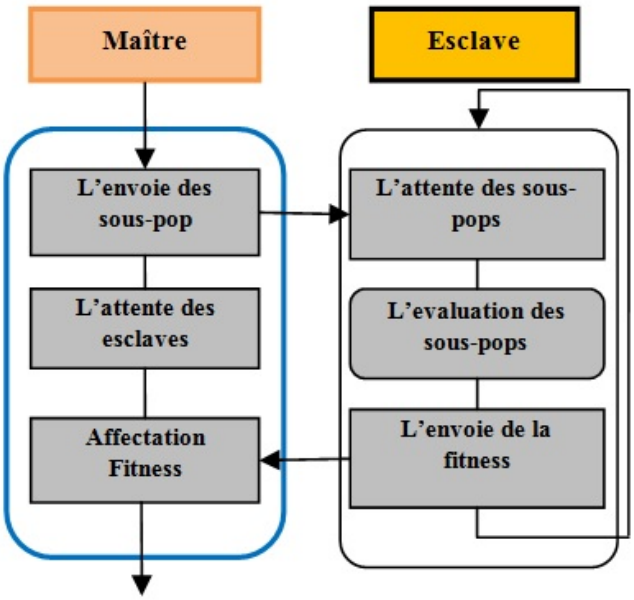


FIGURE 3.10 – Exécution d'un modèle Maitre/Esclave.



Les entrées : P : la population parent.
R : les parents sélectionnés.
C : les enfants de la population.
P. initialise ()
P. evaluate ()
Tantque le critère d'arrêt n'est pas atteint faire
Pour i=1 : λ faire
R: = P. selectNparents ()
C: = variations (R)
C: =add(c)
Fin pour
EnvoieAuxEsclave (C)
F: = attentePourEvaluationPop
AssignFitToPop (F, C)

P. ajouter ()
P. réduire (μ)

Fin Tant que

Algorithmel : un AE en utilisant le modèle M-E

FIGURE 3.11 – Algorithme de l'évaluation au sein d'un modèle Maitre/Esclave.

Approches non standard :

Il existe des implémentations dites *panmictiques*. Dans ce cas, elles perdent leur comparabilité avec les algorithmes séquentiels, mais le principe sous-jacent reste toujours le même [82]. On peut distinguer :

- **Les AEs panmictiques distribués :** cet algorithme effectue un grand nombre d'échanges individuels entre les noeuds, avant les étapes globales de l'algorithme ;

- **Les AEs panmictiques asynchrones :** ce type d'algorithmes est effectivement très approprié pour une architecture à mémoire partagée, avec un nombre restreint de processeurs (threads). L'absence de synchronisation et de communication entre les cores permet une accélération linéaire par rapport au nombre de coeurs, au moins en termes de nombre d'évaluations.

Les AEs parallèles sont développés en tenant compte des architectures matérielles disponibles. Partant de cette constatation, Golub et coll. ont proposé [65], [67], [66] un algorithme qui se concentre sur l'utilisation de la mémoire partagée des machines parallèles des années 2000. Durant ces années, un PC parallèle fusionne 2 ou 4 coeurs. En utilisant un modèle maître/esclave, ont obtient une vitesse utile dans certains cas, mais présentant des restrictions pour certains types d'algorithmes.

3.6.2.2 AEs parallèles distribués (En îlots)

L'utilisation d'une seule population, où tous les individus peuvent potentiellement s'accoupler pour produire un descendant est évidemment loin d'un modèle de la nature. En outre, nous avons vu que cela nécessite soit l'utilisation du modèle classique maître-

esclave centralisé, ou des échanges très fréquents entre les sous-populations. Ceci impose des contraintes sur le matériel, qui ne peuvent être remplies par l'architecture utilisée.

Ce modèle qui est plus proche de la notion naturelle de DEME, est adapté pour les architectures à mémoire distribuée, qui sont largement utilisées dans le calcul à haute performance. Ce modèle de parallélisation est dit 'Algorihmes évolutionnaires distribués' (distributed evolutionary algorithm, dEA).

Chaque noeud exécute un AE classique, avec un échange de temps en temps des meilleurs individus avec les autres noeuds du réseau.

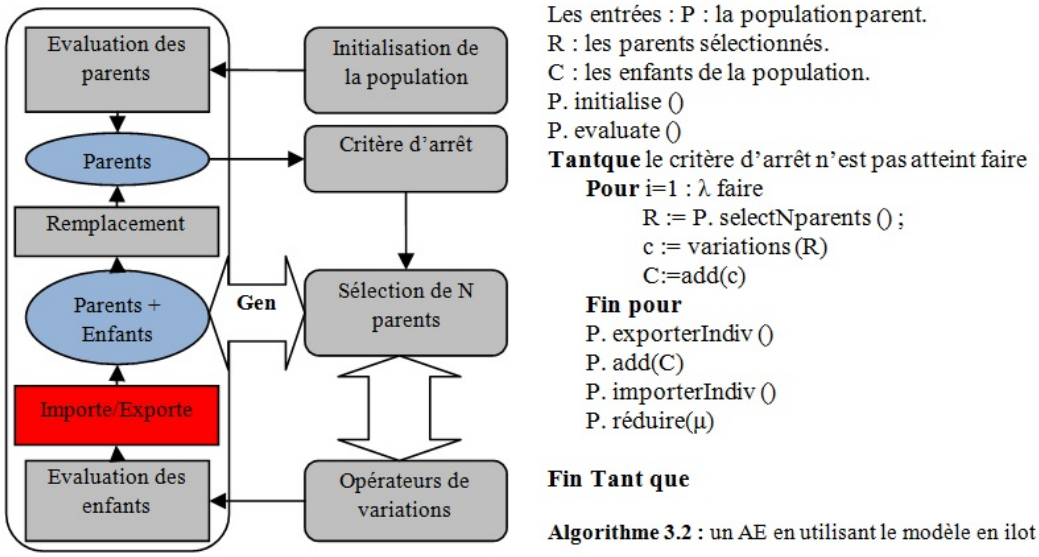


FIGURE 3.12 – Algorithme de l'évaluation au sein d'un modèle en ilôt.

Ces modifications permettent un parallélisme avec peu ou pas de synchronisation entre les noeuds et quelques transferts entre eux. En outre, la sous-population peut être considérée comme une population unique répartie entre plusieurs noeuds, donc présentant une structure. Cette distribution peut avoir comme effet d'augmenter la mémoire disponible pour stocker la population.

Il y a beaucoup de choix de conception qui influent sur le comportement d'un modèle en Îlot, parmi les plus importants : *la politique d'émigration, la politique d'immigration, l'intervalle de migration, le nombre de migrants et la topologie de la migration.*

3.6.2.3 Algorithmes évolutionnaires parallèles hybrides

Il est également possible de combiner plusieurs des approches précitées. Par exemple, on peut imaginer un modèle en îlots où chaque île exécute un AE cellulaire afin de promouvoir davantage la diversité. On peut penser à des modèles hiérarchiques de l'îlot où les îles sont eux-mêmes des modèles insulaires. Les modèles parallèles insulaires (en îlots) et cellulaires peuvent également être mis en oeuvre en tant que modèles maître-esclave pour obtenir une meilleure accélération [124].

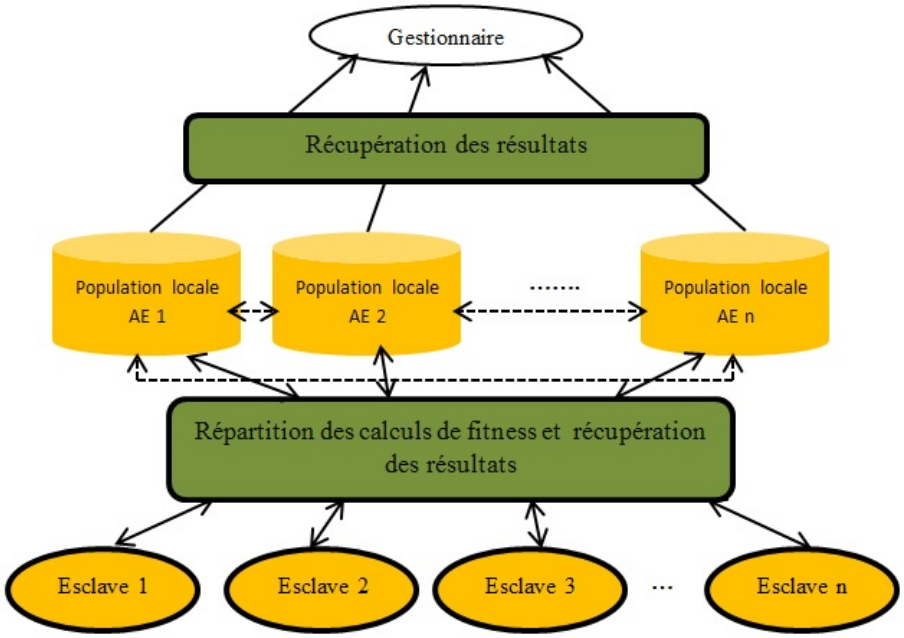


FIGURE 3.13 – Algorithme génétique hiérarchique.

3.7 Paradigmes informatiques considérés comme outils de parallélisation pour les AEs

Divers auteurs, chercheurs, scientifiques ont parallélisé des AEs sur différentes architectures de calcul parallèle, à savoir : les Clusters, MPP (Massively Parallel Processing), GPGPU (General Purpose Graphics Processing Units), les Grilles de calcul, le Cloud Computing, les processeurs Multi-coeurs/HPC, pour assurer une solution optimale à chaque fois avec efficacité et efficience.

La performance d'un AE est optimisée par la parallélisation. La façon dont les AEs Parallèles sont parallélisés dépend des paramètres suivants :

1. L'initialisation de la population.
2. La disponibilité d'une population unique ou de multiples sous-populations.
3. La politique de migration, taux de migration.
4. La méthode de sélection.
5. La fonction objective.
6. Le croisement.
7. La mutation.
8. La méthode de sélection de survie.

Selon la façon dont chacun de ces éléments est mis en oeuvre, différentes méthodes d'algorithmes parallèles sont plus adaptées, comme présenté dans la section précédente. Dans notre thèse, nous proposons d'élaborer une brève étude concernant les paradigmes informatiques considérés pour la parallélisation des AEs avant d'entamer notre étude concernant la contribution des GPUs dans cette zone.

3.7.1 Cluster et MPP

Un Cluster est généralement déployé pour améliorer la performance et la disponibilité existante sur un seul ordinateur. La plupart des implémentations existantes des AEs parallèles utilisent soit le Cluster ou le MPP pour l'exécution (large nombre de processeurs (ou ordinateurs) pour effectuer un ensemble de calculs coordonnés en parallèle.). Le retard de communication devrait être pris en considération lors de la mise en oeuvre sur des clusters et des MPPs. La plupart des supercalculateurs existants sont soit des Clusters ou bien des MPPs [138]. Les Clusters présentent divers inconvénients sur grille tels que le contrôle centralisé, moins de sécurité, pas d'accès direct aux ressources hétérogames distribuées et l'accès à distance complexe aux sources de données. L'avantage des clusters et des MPPs réside dans le fait qu'ils se composent de noeuds homogènes, facilitant ainsi le développement d'algorithmes parallèles.

Le modèle le plus célèbre des ALGORITHME EVOLUTIONNAIRE PARALLÈLEs s'exécutant sur des clusters est le modèle d'îlot. Les travaux [27], [124] et [14] rapportent des mises en oeuvre sur les clusters.

3.7.2 Grille de calcul

La Grille de calcul (composée de plusieurs ordinateurs interconnectés en réseau conjointement pour effectuer de grandes tâches) est un concept important, il a rapidement fait évoluer des paradigmes de programmation parallèles. Il est plus faiblement couplé, hétérogène avec une distribution géographique. La grille de calcul présente de nombreux avantages par rapport aux Clusters tels que : l'absence de contrôle centralisé, la sécurité, l'accès aux ressources hétérogènes distribuées, l'accès facile et fiable à des sources de données à distance et un service à toute les applications disponibles.

Un Algorithme Génétique sur une grille de calcul est dit Algorithme Génétique Orienté Grille (GOGA). Ce terme a été introduit par Herrera et coll. [82]. Les algorithmes génétiques coarse grained, medium grained et fine grained GAs ont été implémentés sur grilles de calcul.

3.7.3 Les architectures multicoeurs et les systèmes HPC

De nombreux Algorithmes Génétiques ont été mis en oeuvre sur des systèmes multicoeurs, mais ne soucient pas de leur utilisation CPU/Coeur. Aujourd'hui, les processeurs multi-coeurs sont moins en moins chers et sont facilement disponibles. Les Systèmes multicoeurs comportent plusieurs coeurs de traitement sur une même puce tandis que les systèmes multiprocesseurs sont construits par assemblage de plusieurs processeurs (puces) à l'intérieur d'un même système. Les défis des AEs sur des systèmes HPC sont liés à la prise en charge de la tolérance aux pannes, de l'assurance d'évolutivité, de l'équilibrage de charge, du stockage de données ainsi que de l'exploitation optimale des architectures multi-coeurs.

Zheng et coll. [211] se sont intéressés à l'élaboration des Algorithmes Génétiques Parallèles avec la perspective des architectures de type Multi-cores et Many-core. De son côté, Cristea a proposé la conception et le design des Algorithmes Génétiques Parallèles sur des modèles d'analyse et de performance HPC [25]. Par ailleurs, Zhuang et coll. ont proposé un Algorithme Génétique Parallèle pour la planification sur la puce IC dans un système

HPC [214]. Dunlop et coll. [42] utilisèrent les AGs sur l'outil de réglage automatique de référence.

Les architectures des Algorithmes Génétiques Parallèles sont mises en oeuvre sur des Clusters, des MPPs, des grilles de calcul, des GPUs, le Cloud Computing, les architectures Multi-coeurs et les HPCs. Pour la conception d'un AGP sur une architecture Multi-coeurs, HPC et GPGPU, le nombre de coeurs peut être pris en considération afin d'optimiser la coopération et l'utilisation CPU/GPU.

3.7.4 Le Cloud Computing

Nous vivons aujourd'hui l'ère de la croissance rapide de l'informatique et des logiciels de parallélisation. Acheter la dernière configuration du PC ou un système de haute performance représente un élément de jugement d'actualité. Cependant, il n'est pas possible d'investir dans toutes les dernières infrastructures informatiques matérielles et logicielles. La disponibilité de l'infrastructure informatique et de logiciels sur un Cloud est assez facilement disponible.

Du fait de l'exigence de potentialités de calcul intensif et parallèle d'un Algorithme évolutionnaire Parallèle, il s'avère difficile de mettre en oeuvre et d'optimiser ses performances sur un Cloud Computing. En outre, développer un AE Parallèle sur un Cloud requiert impérativement des connaissances approfondies sur l'AE ainsi que de sur le Cloud Computing. Beaucoup de questions liées au Cloud, comme la sécurité, la bande passante, etc. sont encore en discussion.

Zhao et coll. ont mis en oeuvre un AE parallèle, sur un prototype de Cloud Computing appelé Hadoop [210].

Par la suite, les travaux qui seront mentionnés dans chaque catégorie sont liés principalement aux architectures des GPUs.

Partie 3 :
Contribution des accélérateurs graphiques (GPUs) dans la parallélisation des algorithmes évolutionnaires

3.8 Travaux de recherche sur les AEs à base GPU

Les AEs sont des algorithmes de recherche stochastiques permettant de renforcer la zone de recherche avec des solutions alternatives pour trouver la solution optimale. Ils ont réussi dans la résolution des problèmes d'optimisation difficiles dans divers domaines tels que l'optimisation, l'apprentissage, l'adaptation, et autres [180]. La qualité des résultats prévus dépend de nombreux facteurs, y compris la puissance de calcul disponible [2].

Récemment, en raison de leur nature parallèle, il y a eu plusieurs tentatives pour accélérer les algorithmes évolutionnaires sur les architectures massivement parallèles à base de GPUs. La plupart des travaux sur les GPUs sont mis en oeuvre avec le Toolkit de développement CUDA, ce qui permet une programmation sur GPU dans un langage plus accessible de C [190].

Cette partie permet de passer en revue divers approches parallèles des AEs utilisant les GPUs. Nous proposons une classification des AEs parallèles avec implémentation sur les GPUs en nous basant sur le nombre de Threads affectés pour chaque chromosome (solution). En effet, les travaux relatifs à l'implémentation des AEs basée GPU peuvent être distingués selon deux catégories :

1. Affectation d'un Thread à chaque chromosome ;
2. Affectation d'un Thread à chaque gène du chromosome.

3.8.1 GPU-AEs basés chromosome

Dans cette catégorie, Arora et coll. [2] peuvent être cités. Leur étude a présenté une mise en œuvre pour les AGs exploitant le toolkit CUDA. Dans cette étude les auteurs ont tenté d'exploiter un ensemble de paramètres (nombre de Thread, taille du problème, taille de la population, etc.). En appliquant le paradigme de programmation parallèle GPGPU, ils ont pu obtenir des accélérations avec un facteur de l'ordre de 40 relativement à la version séquentielle de leur algorithme.

Ogier et coll. [126] ont proposé différentes stratégies de portage des AEs sur GPU dans le cadre du framework EASEA (Easy Specification of Evolutionary Algorithm) qui est dédié à des non-spécialistes pour les aider à optimiser leurs problèmes en exploitant un AE. Les algorithmes présentés, et les accélérations réalisées dans leur travail sont classés selon différentes cartes graphiques NVIDIA pour différentes familles d'algorithmes d'optimisation [126].

Pospichal et coll. [152] ont adopté un AG parallèle avec le modèle en ilot de Skolicki [175], pour une mise en oeuvre exécutée sur le GPU avec 256 individus pour chaque ile. Pour maintenir la population, les auteurs ont proposé d'utiliser une architecture basée sur le plaquage des Threads sur les individus, en utilisant le "on-chip hardware scheduler" afin d'échanger les iles entre les multiprocesseurs rapidement, et ce dans le but de masquer la latence de la mémoire. Les auteurs ont rapporté une vitesse jusqu'à 7000 fois plus élevée sur le GPU par rapport à la version séquentielle de l'algorithme.

Jaros et Pospichal [98] ont comparé une mise en oeuvre optimisée sur un processeur multicoeurs avec une version GPU fortement optimisé d'un algorithme génétique pour le problème du sac à dos . L'objectif principal de cette étude était de présenter la relation entre la performance réelle entre les processeurs modernes (CPUs) ainsi que les accélérateurs graphiques, afin d'éliminer certains mythes de la performance des environnements GPUs. Ils ont clairement montré qu'une version CPU soigneusement mis en oeuvre peut

être plus rapide qu'un seul thread GAlib compilé par défaut. Ils ont également montré que d'une manière réaliste, qu'un seul NVIDIA GTX580 peut surpasser un processeur Intel Xeon X5650 avec un facteur d'environ 12 tout en atteignant une efficacité d'exécution de 26 % et une performance de 405 GFLOPS.

Zhu a suggéré dans [213] un algorithme de stratégie d'évolution pour la résolution d'un banc de problèmes continu en utilisant le Toolkit CUDA. Dans son implémentation, plusieurs Kernels sont conçus pour quelques opérateurs évolutionnaires tels que la sélection, le croisement, l'évaluation et la mutation. Le reste du processus de recherche est géré par le CPU. Dans leur implémentation, la performance de crête est prévue lorsque le nombre de threads est suffisant pour maintenir une activité simultanée pour tous les multiprocesseurs. Pour cette raison, l'analyse effectuée est principalement basée sur 10240 threads, avec une accélération représentant un facteur 100.

3.8.2 GPU-AEs basés Gènes

Le travail effectué par Shah et coll. [171] est un exemple de cette catégorie. Les auteurs se sont proposés d'exploiter le parallélisme à l'intérieur d'un chromosome (intra-chromosome), en plus du parallélisme effectué pour évaluer de multiples chromosomes (inter-chromosome). Leur implémentation traite un AG simple avec un chromosome 1D et deux méthodes différentes de sélection. Le cadre proposé peut être étendu à une accélération de la bibliothèque d'algorithmes génétiques générique sur GPU en intégrant de plus en plus de fonctionnalités. Avec une accélération réalisés au cours d'un facteur de 1000 et une bibliothèque-programmable comme interface, le GPU GA accélérée peut trouver des applications dans de nombreux domaines.

Kromer et coll. [106] ont proposé une technique utilisant la même analogie avec le précédent travail, en étudiant l'évolution différentielle . Avec l'aide du toolkit CUDA, la fonction objective a été accélérée d'un facteur allant de 2.2 à 12.5. De plus, celle-ci est devenue plus rapide que la version CPU avec un facteur allant de 25.2 à 216.5 et ceci en exploitant un code orienté objet c++.

Oiso et coll. [143] ont suggéré une technique exploitant le parallélisme entre les individus et les gènes simultanément. La méthode de mise en oeuvre proposée a donné des résultats environ 18 fois plus rapides que la mise en oeuvre sur CPU de leurs tests de référence.

Il convient de signaler que toutes ces études ont exploité l'aspect parallèle sur GPU et ont montré l'efficacité de ces dispositifs pour améliorer l'évolution des comportements complexes. Par conséquent, et pour faire face à l'accélération de l'évolution des comportements complexes de robots humanoïdes, nous proposons d'utiliser une stratégie d'évolution orientée vers le parallélisme GPU, afin de faire face à la phase d'apprentissage du RNN, ce dernier étant responsable du contrôle du mouvement du robot humanoïde (Notre proposition).

3.9 Considérations pour une implémentation parallèle d'un AE sur la GPU

Pour réaliser une implémentation parallèle, la population doit être affectée aux multiprocesseurs (MPs) présents sur la carte. Cette étape est importante, car elle doit être

en mesure d'assurer un bon équilibrage de charge sur chaque processeur de l'architecture multi-GPU ainsi qu'un ordonnancement efficace.

Toutefois, cette répartition doit être compatible avec les contraintes matérielles. Ces contraintes matérielles consistent à vérifier que chaque bloc ne puisse utiliser que les ressources effectivement disponibles sur un MP. Certains principes relatifs à l'occupation de la carte et la distribution automatique efficace des individus sur la carte pour contrôler le parallélisme de Threads seront présentés.

Un SM (Streaming Multi-processor) peut exécuter deux demi-Warps en même temps. Les Warps de plusieurs blocs de Threads peuvent être mis en attente sur le même SM. Lorsque les Threads d'un Warp établirent une demande à la mémoire globale, ces Threads sont bloqués jusqu'à ce que les données arrivent de la mémoire. Pendant ce temps de latence élevé, d'autres Warps dans la file d'attente peuvent être organisés et exécutés. Ainsi, il est important d'avoir suffisamment de chaînes en attente dans le SM pour masquer les latences de la mémoire globale en les chevauchant avec le calcul, ou avec d'autres accès à la mémoire. La première considération pour maximiser l'occupation est de choisir une taille de bloc appropriée. Par exemple, dans l'architecture Fermi [186], le nombre de Threads par bloc doit être un diviseur entier du nombre maximal de Threads par SM. Ce nombre de Threads doit être supérieur ou égal à 192, afin de permettre de remplir le nombre maximal de Threads par SM avec pas plus de 8 blocs. En outre, concernant l'effet d'occupation, la forme choisie a également un impact significatif sur la coalescence (une demande de mémoire effectuée par des threads consécutifs dans un demi-Warp est strictement associée à un segment), le partition-camping (causé par des accès mémoire qui sont biaisé vers un sous ensemble de partitions de mémoire disponibles, ce qui peut dégrader les performances des Kernels GPU jusqu'à sept fois), et les goulets d'étranglement de mémoire. Une chaîne devrait demander 32 éléments contigus pour réduire la bande passante de la mémoire totale.

Une machine parallèle pourrait avoir plus de ressources comme les mémoires ou le cache ; et c'est le cas des accélérateurs graphiques. Lors du portage d'un code de la machine CPU à la machine GPU, l'algorithme pourrait utiliser ces ressources. Ces ressources représentent la hiérarchie de mémoire, où l'application des optimisations aux accès mémoires peut améliorer significativement la performance.

3.10 Récapitulation

Plusieurs tentatives ont eu lieu, à l'effet d'accélérer des AEs sur des architectures massivement parallèle orientées GPU. Cependant, de nombreux chercheurs ont exploité dans leurs entreprises des benchmarks numériquement simples sans données globales ou avec seulement un ensemble de données très limité [51]. Ceci pourrait être considéré comme une contradiction aux problèmes du monde réel, où des simulations à des échelles plus larges doivent être effectuées et où l'évaluation des conditions de ces simulations est souvent l'opération la plus délicate et la plus sensible, mais souvent négligée.

Néanmoins, nous sommes en mesure de tirer un certains nombre de conclusions :

1. Dans les cas où l'évaluation de la fonction de fitness prend plus de temps qu'une phase de manipulation génétique, un AE maître/esclave peut être utilisé en raison de la nature parallèle de la création individuelle et l'évaluation de la fonction de fitness [190], [126].

Classification des AEs basés GPU : à base de gène et à base de chromosome

	Étude	Organisation des données	Approches à base d'un ou plusieurs threads	Gestion de la mémoire de données
À base de chromosome	Arora et coll. (2010)	Des entiers 1D et des tableaux flottants.	Différents threads d'un bloc pour différents individus de la population.	Les variables d'un même type et qui appartient à des individus différents sont stockés de manière adjacente.
	Ogier et coll. (2012)	Collection d'objets représentant les individus.	Un thread par individu pour la phase de l'évaluation.	Les individus sont regroupés dans des tampons contigus.
	Pospical et coll. (2010)	Des tableaux séparés 1-D, où chaque tableau représente la sous-population.	Chaque individu est contrôlé par un seul thread CUDA.	Les populations de l'île locale sont stockées dans la mémoire partagée de la puce GPU.
	Zhu (2009)	Des tablaux 1-D séparés, où chaque tableau représente la sous-population.	Chaque individu est contrôlé par un seul thread CUDA.	Les mémoires : Globales, partagée, et de texture.
À base de gène	Kromer et coll. (2011)	Toute la population peut être considérée comme une matrice réelle.	Chaque vecteur (solution) est traité par un bloc de threads.	Toute la population réside dans la mémoire principale.
	Osio et coll. (2011)	Collection d'objets représentant la population.	Chaque individu est traité par une SM, et chaque gène par un SP.	Toute la population réside dans la mémoire principale.
	Jaros et coll. (2012)	C structure constituée de deux tableaux à une dimension.	Chaque vecteur (solution) est traité par un bloc de threads.	Les chromosomes sont stockés dans le cache L1 du GPU.
	Shah et coll. (2010)	Toute la population peut être considérée comme une matrice réelle.	Chaque vecteur (solution) est traité par un bloc de threads.	La matrice de la population réside dans la mémoire principale du processeur graphique.

TABLE 3.1 – Synthèse des travaux des algorithmes évolutionnaires à base de GPU.

2. D'un autre côté, si l'évaluation de la condition prend un temps similaire au processus évolutif, il est généralement préférable d'exécuter l'ensemble de l'AE sur le GPU [2].

Ces alternatives sont basées sur l'attribution d'un individu à un Thread, ou l'attribution d'un individu à un bloc de Threads.

Avec de petites populations, l'inconvénient de cette approche est la limitation dite *par bloc* introduite par CUDA [26]. L'effet de cette restriction réside dans le fait qu'il est très difficile de mettre en oeuvre des fonctions d'évaluation et d'exploiter de larges chromosomes. D'autre part, l'attribution d'un individu par bloc de Threads nécessite de larges chromosomes pour aboutir à l'occupation maximale de tous les Threads, où les gènes doivent être lus à plusieurs reprises pour effectuer une simulation complexe [106], [143].

Peu de travaux vérifiant le modèle en Ilots ont été développés, où chaque île est stockée dans la mémoire partagée comme une autre approche [152], [133]. Parmi les limites de cette approche est d'exploiter la même configuration des paramètres entre les îles, car les processeurs présents sur le GPU sont utilisés selon le modèle SPMD (Single Program Multiple Data).

3.11 Conclusion

Dans ce chapitre, nous avons décrit tous les concepts nécessaires à la compréhension générale du document. Dans ce but, nous avons introduit les notions liées au parallélisme, les sources de parallélisme dans les AEs, les modèles de parallélisation dans les AEs et l'émergence des accélérateurs graphiques comme outil de parallélisation pour des fins de calcul généraliste y compris la parallélisation de l'évolution artificielle. Par ailleurs, nous nous sommes surtout concentrés sur les AEs parallèles sur les GPUs. La compréhension de l'organisation hiérarchique de l'architecture du GPU est utile à l'effet de prévoir une implémentation parallèle efficace d'un processus évolutionnaire.

L'objectif de cette thèse est de revoir les différents modèles parallèles de l'évolution artificielle sur des architectures GPU. Dans ce but nous avons proposé une vue globale des différents défis déjà levés ou à relever dans la conception et la mise en oeuvre des AEs, en exploitant les accélérateurs GPUs.

Chapitre 4

PEvoRNN : GPU Computing pour la robotique évolutionnaire (Framework GPGPU)

4.1 Introduction

Définies comme un sous-domaine de l'intelligence artificielle (IA), les techniques évolutionnaires ont été utilisées pour développer des contrôleurs pour des robots autonomes et forment un sous-domaine appelé "robotique évolutionnaire". Plus spécifiquement, un des meilleurs investissement des systèmes informatiques avancés réside dans l'opportunité de trouver de nouvelles voies qui offrent des solutions performantes aux problèmes de la robotique. Néanmoins, la plupart de ces solutions sont basées sur les paradigmes de l'IA classique en s'appuyant sur la conception humaine. Ces solutions étant, néanmoins, limitées en termes d'efficacité, si le contrôle du robot est fortement contraint et soumis à des fonctions objectives différentiables [146]. L'IA classique est également incapable de faire face à des comportements de robots qui soient entachés d'incertitude [3] en raison de la programmation explicite des comportements souhaités en utilisant un modèle mathématique exacte et complet pour concevoir le robot et son environnement [192]. Pour les robots à pattes, divers algorithmes d'apprentissage ont été proposés afin de fournir des opérations autonomes aux nombreux problèmes complexes avec le challenge de conception et de contrôle tel que présenté dans le travail de Tang en 2007 [181]. Un de ces problèmes concerne le contrôle de la marche d'un bipède.

Afin de traiter ce problème, des alternatives de contrôle biologiquement inspirées ont été proposées [69]. Dans ce cadre, l'apprentissage d'un réseau neuronal multicouche avec un algorithme évolutionnaire est l'une de ces méthodes efficaces proposées pour concevoir des contrôleurs pour ces robots.

Lors du contrôle, le réseau de neurones à base évolutive consomme souvent un temps de traitement important, surtout s'il y a une large gamme de données, même avec des entraineurs efficaces de réseaux de neurones. Pour faire face à cette restriction, les architectures parallèles ont été proposées pour améliorer les performances de calcul et de traitement [150], [177], [15], [139], [134], ceci vu le constat que les technologies associées aux processeurs actuels (architectures CPUs) ont atteint leurs limites en termes de puissance de calcul.

Les chercheurs en biomécanique, en robotique, et en informatique tentent de comprendre le mouvement naturel humain afin de trouver de nouvelles inspirations pour résoudre des problèmes de calcul en reproduisant ce mouvement humain.

L'un des domaines les plus importants ayant été inspiré par le mouvement humain est la robotique humanoïde. L'objectif principal de la recherche dans ce domaine est d'obtenir des robots qui peuvent reproduire les comportements humains afin de collaborer de la meilleure façon avec les humains. Un problème inhérent à la robotique humanoïde est la génération d'allures stables, réalistes et efficaces en un délai raisonnable. Pour prendre en charge ce problème, des alternatives bio-inspirées ont été proposées [145], elles n'incluent pas les spécifications des références de trajectoires aidant à réduire le temps de calcul.

La modélisation et l'intégration de modalités différentes et de comportements de la robotique humanoïde nécessitent des algorithmes complexes et des calculs intensifs qui pourraient s'exécuter en des temps raisonnables. Dans cette étude, nous mettons l'accent sur la proposition d'un contrôleur de robot amélioré basé sur deux techniques bio-inspirées nommément, les stratégies évolutionnaires (SEs) et les réseaux de neurones artificiels (RNAs).

D'un côté, les AEs sont considérés comme une des méthodes d'optimisation les plus importantes qui engendrent des solutions approchées, néanmoins avec des temps de calcul très élevés [156], [98]. Dus à leur nature intrinsèquement parallèle, les AEs sont très appropriés pour les systèmes distribués [150]. Plusieurs de ces techniques visent à fournir une plateforme de calcul généraliste, comme mentionné dans le chapitre précédent.

D'un autre côté, plusieurs études antérieures ont examiné des alternatives de parallélisation matérielles pour les réseaux de neurones artificiels (RNAs) en utilisant des dispositifs de calcul tels que Field Programmable Gate Array (FPGA) et les GPUs qui offrent une architecture parallèle fine-grained [123], [170].

La mise en œuvre de l'association SE/RNA, incluant une phase d'entrainement, sur la plateforme accélératrice graphique CUDA constitue un cadre idéal, et ce en raison de ses caractéristiques qui en font un environnement de programmation massivement parallèle pour le GPU. De plus, du fait que l'architecture du GPU qui contient des milliers d'unités indépendantes, opérant en virgule flottante, et connectées à une mémoire intégrée, et disposant d'une bande passante élevée, rendant ce dispositif parfait pour fournir des accélérations significatives exploitant le parallélisme [148].

Comme l'usage général d'unité graphique de calcul (GPGPU) est un domaine émergent dans de nombreuses disciplines, il y avait peu d'approches proposées pour faire face à l'application de cette technique dans la robotique évolutionnaire [69], [149], [172], [137].

Dans cette étude, nous proposons une idée originale basée sur une implémentation sur GPUs qui vise à accélérer l'évolution d'un robot évolutif afin de fournir des comportements efficaces nécessaires dans plusieurs situations complexes comme la marche vers une destination cachée. Cette proposition est fondée sur la combinaison d'une stratégie évolutive (SE) et un réseau de neurones récurrents (RNN) où la SE est responsable de l'optimisation de la génération des trajectoires du robot humanoïde, et le RNN constitue le cerveau contrôlant les comportements du robot. Pour accélérer le processus d'évolution de l'apprentissage du RNN, nous proposons l'utilisation d'un accélérateur graphique (GPU) sur plusieurs niveaux, en tenant compte de la version GPGPU intégrant le simulateur physique ODE (Open Dynamique Engine) proposé par Zamith [207]. Le processus de contrôle du robot exploitant le GPU s'exécute nettement plus rapidement que la solution séquentielle basée CPU.

Pour ce faire, il est nécessaire de garantir une utilisation efficace et optimale de la hiérarchie mémoire associée au GPU. La principale nouveauté ici est de réduire le temps de transfert des données entre les deux sous-systèmes de mémoires (c.-à-d. la mémoire du CPU et la mémoire GPU). En outre, pour rentabiliser l'utilisation du GPU, il est nécessaire d'assurer une gestion judicieuse et efficace du parallélisme du GPU, et ce, à tous les niveaux du processus d'évolution du contrôleur. De plus, nous proposons un ensemble d'instructions génériques pour la programmation du GPU, ces instructions devant être respectées afin de bénéficier des performances optimales du GPU. Les résultats obtenus sont comparés à ceux obtenus avec la mise en œuvre séquentielle.

Le rôle de ce chapitre est bien d'explorer deux volets, où *le premier* sert à présenter les possibilités existantes pour la parallélisation des composants du problème étudié, et *le deuxième* permet de présenter notre vision théorique et technique concernant le processus de parallélisation du problème étudié à base GPU.

Partie 1 :
Analyse du parallélisme au sein du problème étudié

4.2 Analyse du parallélisme au sein d'un problème lié à la robotique évolutionnaire

Sachant que le problème étudié est composé de trois éléments essentiels, il serait très intéressant d'analyser et de souligner quelles sont les parties potentiellement parallélisables.

4.2.1 Niveau stratégie évolutionnaire

Les AEs disposent d'une caractéristique intéressante qui consiste dans le fait qu'ils soient intrinsèquement parallèles. En effet, en nous basant sur un AE standard, il est possible d'analyser la parallélisation de toutes ses étapes [107], [124].

En outre, il est possible d'étudier le nombre de tâches pouvant être créées à chaque étape. Ce point est particulièrement intéressant lors de l'utilisation du principe de programmation GPGPU. En effet, comme nous l'avons vu, de nombreux cœurs réels et virtuels sont présents au sein d'une architecture GPU, et tous ces cœurs doivent être maintenus occupés pendant l'exécution de tout l'algorithme pour une efficacité maximale.

Dans cette section, μ se réfère au nombre de parents et λ fera référence au nombre d'enfants.

4.2.1.1 Initialisation

Il s'agit d'une étape parallélisable, dans le cas général. En effet, l'initialisation d'un individu ne dépend généralement pas de celle des autres individus, les gènes des individus sont habituellement initialisés de façon aléatoire. Par conséquent, l'initialisation des individus peut être effectuée par des tâches, de manière indépendante sur les N cœurs.

4.2.1.2 Évaluation

Il s'agit d'une étape également parallélisable, car dans le cas général, un individu est évalué indépendamment des autres. Après la phase d'initialisation, l'évaluation doit être effectuée une fois par parent (μ fois). Dans la boucle de l'évolution, l'évaluation doit être effectuée une fois pour chaque enfant (λ fois).

4.2.1.3 Critère d'arrêt

Il s'agit d'une étape également parallélisable :

1. Si le critère d'arrêt est le nombre de générations, chaque cœur peut vérifier indépendamment si ce critére est atteint.
2. Si le critère d'arrêt est l'atteinte d'une valeur spécifique de la fitness, chaque cœur peut vérifier indépendamment si l'individu, qui vient d'être évalué, a rencontré ce seuil ou non. Si cela est le cas, le cœur peut procéder à l'arrêt du moteur de l'évolution à la prochaine génération, chaque cœur sera interrogé et le meilleur individu de tous les cœurs est retourné.

4.2.1.4 Choix de sélection pour la création de descendants

Il est nécessaire de sélectionner un certain nombre de parents pour la recombinaison. Comme il est possible (et même souhaitable) que le même individu soit sélectionné pour créer plusieurs descendants, n cœurs peuvent sélectionner indépendamment α parents en parallèle pour la reproduction. Si α est l'arité de l'opérateur de recombinaison, la sélection des parents est exécutée $\lambda * \alpha$ fois indépendamment, afin de produire λ descendants.

4.2.1.5 Recombinaison

Après le choix indépendant des parents, chaque cœur peut procéder à la création d'un enfant, en effectuant une recombinaison entre les gènes d'un parent sélectionné, qui peut être consulté simultanément en lecture seule.

4.2.1.6 Mutation

Après la création d'un descendant dans la phase de recombinaison, on peut effectuer une mutation indépendamment de toutes les mutations des autres descendants. Ainsi, cette phase est également parallélisable.

4.2.1.7 Remplacement

Cette étape n'est pas parallélisable d'une manière simple, au moins une méthode de réduction simple est appliquée.

Si un schéma de remplacement est sélectionné dans une stratégie évolutionnaire, un remplacement efficace de la population implique une sélection de μ survivants sans remplacement. En effet, si le même individu a été sélectionné plusieurs fois pour remplir la nouvelle génération, cela provoquerait la génération de clones qui induiraient la réduction de la diversité et provoqueraient une convergence prématurée.

S'assurer que tous les individus sélectionnés sont différents exigerait la communication des différents cœurs dans le but de ne pas choisir le même individu. La sélection de μ différents individus hors de la population temporaire $\mu + \lambda$ implique de nombreuses communications et synchronisations, qui sont coûteuses dans un environnement parallèle.

Afin de résoudre ce problème, nous proposons dans le chapitre 5 (la section 8.4) l'opérateur de sélection, qui travaille sur des ensembles disjoints d'individus.

Comme mentionné au-dessus, toutes les étapes d'un AE, sauf une peuvent être parallélisées d'une manière simple sur une architecture parallèle à mémoire partagée. Le nombre de tâches est généralement élevé, soit respectivement, μ, λ, et $\lambda * \alpha$ (nombre de tâches pour créer les parents, nombre de tâches pour créer les enfants, nombre de taches pour sélectionner les enfants). Même si une phase de réduction séquentielle est mise en œuvre à la fin de chaque génération, il est possible de paralléliser le contenu de la boucle d'évolution qui est notamment la plus coûteuse.

D'un point de vue parallélisation, l'exécution de multiples itérations de la boucle évolutive n'est pas possible simultanément, car la population actuelle dépend de la précédente.

Un AE est réalisé par différentes étapes, comme décrit ci-dessus. La complexité des opérateurs génétiques dépend de *la taille du génome* et l'évaluation d'un individu dépend du *problème*. Enfin, toutes les différentes étapes dépendent de la taille des populations μ et λ.

4.2.2 Niveau réseau de neurones

Les réseaux de neurones naturels contiennent un parallélisme intrinsèque, il semble donc logique de retrouver cette propriété pour les réseaux de neurones artificiels.

L'un des problèmes liés à l'utilisation des réseaux de neurones est en effet le fort coût de ceux-ci en temps de calcul, et ce, essentiellement en phase d'apprentissage. Il est donc intéressant de pouvoir utiliser la puissance de calcul des machines parallèles modernes pour accélérer l'exécution de nos réseaux, pour les configurer comme pour les exécuter [10].

L'apport du parallélisme peut donc permettre, en premier lieu, de réduire de façon significative ce coût et faciliter ainsi la tâche des connexionnistes. Ce gain est conséquent tant du point de vue exploitation (apprentissage parallèle d'un réseau utilisé séquentiellement ultérieurement) que du point de vue expérimental (gain de temps pour tester la validité d'une méthode). Un autre attrait du parallélisme réside dans le fait qu'il permet une autre approche des réseaux de neurones artificiels. À la différence d'une approche séquentielle, la parallélisation permet de tenir compte du parallélisme intrinsèque des réseaux de neurones et de leur synchronisme. Les neurones d'un réseau effectuent leur calcul à partir des résultats des neurones les "précédant", à la réception de tous ceux-ci, d'où le synchronisme. Tous les neurones effectuent leurs calculs simultanément, d'où le parallélisme intrinsèque [11].

En phase de fonctionnement, pour un réseau multicouche, différentes possibilités de parallélisme peuvent être exploitées. Nous en citerons trois : parallélisme intra-neurone, parallélisme de données, et parallélisme inter-couche [49].

Dans le premier cas, il s'agit du parallélisme de plus bas niveau (on dit aussi « à grain fin »). Cette forme de parallélisme opère à l'intérieur du neurone. Le calcul du potentiel post-synaptique nécessite d'effectuer la somme pondérée des états de sortie des neurones connectés. Ce calcul peut être effectué en parallèle si on utilise simultanément plusieurs unités de calcul (multiplieurs et additionneurs).

Le principe du parallélisme de données repose sur le constat que les neurones d'une couche n'utilisent pour leur calcul de mise à jour que l'état des neurones de la couche antérieure. De ce fait, l'état de sortie de chaque neurone d'une même couche peut être évalué séparément (et donc en parallèle) à partir du moment où la mise à jour de l'état des neurones de la couche antérieure ait été effectuée.

Enfin, le parallélisme inter-couches consiste à exploiter le flot de données. Le temps de relaxation est généralement obtenu en mesurant le temps séparant la présentation d'un stimulus en entrée et l'évaluation en sortie du réseau, c'est-à-dire le temps de propagation dans le réseau. L'ensemble des calculs est alors découpé en différents modules de traitement mis bout à bout. Cette technique, appelée « pipeline », permet de diminuer la durée du cycle de calcul. Dans le cas d'un réseau de neurones, chaque module peut être composé d'une, voire plusieurs, couches du réseau, l'objectif étant de séparer la quantité de calcul en tâches autonomes à peu près équilibrées, de manière à ce que le temps de calcul de chaque tâche soit inférieur au temps de cycle global.

4.2.3 Niveau simulateur physique

Le GPGPU est devenu très important et a permis d'initier un nouveau domaine lié à la recherche graphique (celle de l'infographie). Cela a conduit à un nouveau paradigme, où le processeur n'a pas besoin de calculer toutes les issues non graphiques. Cependant,

il y a certains types d'algorithmes qui ne peuvent être attribués au GPU. En outre, même si un problème est approprié pour le traitement par des GPUs, il y a des cas où l'utilisation des GPUs pour résoudre des problèmes n'est pas utile, car la latence générée par la manipulation de la mémoire sur ce type de dispositifs peut être trop élevée, ce qui mène à une dégradation de la performance de l'application.

De nombreux problèmes mathématiques et de simulation physique peuvent être formulés comme des processus à base de flux, ce qui permet de les distribuer naturellement entre le CPU et le GPU. En bref, les principales tâches effectuées à chaque pas de simulation d'une scène composée d'un ensemble de corps rigides sont la détection de collision, la gestion des collisions, et la résolution d'équations différentielles. Pour la détection de collisions, les GPUs peuvent être utilisés en tant que coprocesseur pour accélérer les techniques de calcul d'intersections [48].

ODE (Open Dynamics Engine) est une bibliothèque Open-Source responsable de la simulation dynamique en temps réel des corps rigides et élastiques et qui correspond à l'un des outils essentiels disponibles pour la simulation de la physique. Le solveur ODE est un composant qui peut également être mis en œuvre efficacement sur les GPUs, étant donné que l'équation de l'intégration destinée aux corps rigides est faite indépendamment des autres. En outre, les données liées à l'état d'un corps rigide et sa dérivée peuvent facilement lire et écrire à partir des flux tel que le solveur Runge-Kutta sur GPU (Kernel arithmétique [207].

Dans la littérature, il y a beaucoup de travaux sur la résolution des équations différentielles sur GPU, en particulier celles liées au système de simulation (discret) de particules. Une des possibilités pour ce type de problème est d'utiliser le GPU comme un coprocesseur mathématique pour mettre en œuvre un certain nombre de techniques de calcul numérique, principalement celles pour les opérations matricielles et de résolution de systèmes d'équations linéaires.

4.3 Récapitulation

Il existe un grand nombre de possibilités de parallélisation pour les algorithmes évolutionnaires, les réseaux de neurones, et même la simulation physique. Diverses technologies peuvent être employées. Notre choix s'est naturellement tourné vers les architectures GPUs tant elle représente notre intérêt principal en plus du potentiel d'accélération très élevé qu'elles recèlent. Nous avons toutefois passé en revue les points parallélisables pour les différentes phases de chaque partie afin de quantifier les possibilités de parallélisation offertes.

Partie 2 :
Framework GPGPU PEvoRNN

4.4 Robotique évolutionnaire

La robotique évolutionnaire (RE) est une technique relativement nouvelle dans le domaine de l'intelligence artificielle dont l'idée principale est de créer des programmes de contrôle adaptatif portables sur des robots réels, et ce, en exploitant des principes évolutionnistes [141]. Cette méthodologie est inspirée de la vision darwinienne de la reproduction sélective du plus apte, qui est reprise par les AEs. Les robots sont considérés comme des organismes artificiels qui développent leurs propres systèmes de contrôle voire leur morphologie en étroite interaction avec l'environnement et sans intervention humaine. La RE s'inspire du concept d'auto-organisation présent en biologie et comprend les aspects de l'évolution, du développement, ainsi que les systèmes neuronaux et morphologiques.

L'avantage réside dans le fait qu'un grand espace de recherche avec de nombreuses dimensions peut être exploré d'une manière efficace et dont les solutions trouvées ne sont pas souvent intuitivement simples (selon la perspective humaine), mais sont néanmoins efficaces. Les inconvénients proviennent du fait que les fonctions des contrôleurs neuronaux apparaissent comme des boites noires et que leur entrainement est très coûteux en terme de temps, en particulier lorsque le matériel fait également l'objet d'un processus évolutif [32].

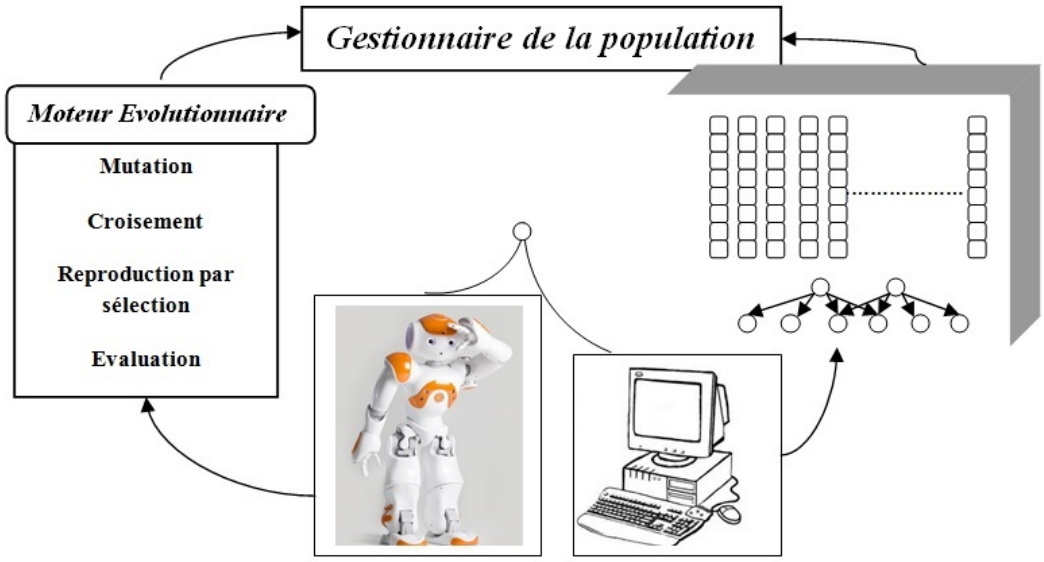


FIGURE 4.1 – Diagramme d'un exemple d'une expérience évolutive sur un seul robot humanoïde.

Les principales étapes méthodologies exploitées dans la robotique évolutive peuvent être définies comme suit :

(a) Création d'une population initiale de chromosomes aléatoires, où chacun représente le système de commande du robot et intégrant également si nécessaire la morphologie.

(b) Les robots disposant d'une liberté d'interaction avec l'environnement pour un nombre déterminé d'itérations tandis que leur performance est évaluée automati-

quement. La performance est mesurée grâce à une fonction de fitness, qui représente une mesure de la façon dont chaque individu évolue.

(c) Les robots ayant les meilleurs scores de la fonction de fitness sont autorisés à se reproduire et régénérer des chromosomes qui sont sujets à des mutations aléatoires et d'échange de matériel génétique en utilisant des opérateurs de croisement. Ce processus crée une nouvelle génération de chromosomes dont certains pourraient être plus efficaces que leurs parents.

(d) Les individus dans la population nouvellement créée sont évalués pour constituer le point de départ d'une nouvelle itération.

(e) Les quatre premières étapes sont répétées pour un certain nombre de générations jusqu'à ce qu'un robot représentant le meilleur individu soit obtenu (traduisant la fonction de fitness définie par l'utilisateur).

4.5 Contrôleurs évolutifs sur machines parallèles

Les techniques évolutionnaires appliquées au contrôle de robots peuvent générer des solutions efficaces, intelligentes, créatives qui correspondent aux contraintes imposées par l'environnement ainsi que par les critères de sélection. La puissance, la flexibilité, et la généralité de l'évolution artificielle ont souvent été exploitées à la fois pour trouver des solutions d'ingénierie pour différents problèmes de contrôle de robots autonomes ainsi que pour faire la lumière sur les mécanismes biologiques inhérents aux comportements adaptatifs.

Cependant, la robotique évolutionnaire, à l'image des autres méthodes d'adaptation, tels que l'apprentissage par renforcement et l'apprentissage des systèmes de classeur, peut nécessiter du temps et des ressources matérielles considérables, exigeant ainsi une évaluation minutieuse des outils matériels et des méthodes employées [116], [150], [15].

La recherche expérimentale, dans le domaine de la robotique évolutionnaire, nécessite des ressources CPU puissantes. Dans ce cadre, plusieurs tentatives ont expérimenté l'utilisation des architectures parallèles dans le but d'améliorer la convergence de l'évolution et réduire le temps de calcul.

Capi et Coll. [15] ont proposé une application d'un AE étendu, pour lequel ils ont implémenté un AG étendu multi-population (EMPGA) [15]. Les sous-populations concurrencent et coopèrent entre elles en appliquant différents stratégies évolutionnaires. Les résultats de cette approche (EMPGA) ont prouvé qu'elle était plus performante par rapport à un AG à une population unique (SPGA), et ce en distribuant d'une manière efficace les individus entre les sous-populations. De plus, la comparaison avec d'autre AGs multi-populations montre que la concurrence entre les sous-populations améliore la qualité de la solution. Les contrôleurs implémentés ont été évolués et testés en utilisant le matériel réel du Cyber Rodent robot.

Dans [150], l'expérimentation de Petrovic sert à réaliser l'évolution d'un AE pour un mécanisme d'arbitrage du contrôleur d'un robot représenté comme un ensemble d'automates d'états finis augmenté. L'expérience de la robotique évolutionnaire dans ce cas est basée sur le package GaLib, qui ne supporte pas la parallélisation, et donc le besoin d'un package de calcul distribué. Par conséquent, ils ont choisi de mettre en œuvre leur

propre technique pour des raisons de simplicité, de modificabilité, de contrôle, de maintenance et d'installation, et en raison d'autres exigences qui sont décrites dans leur papier. La solution distribuée implémentée et utilisée pour acquérir des résultats expérimentaux consistait en deux grandes technologies, les scripts Shell UNIX avec la copie sécurisée pour la soumission, la surveillance et la gestion des tâches, et de base de données SQL pour une application distribuée évolutive.

Parmi les études réalisées pour les accélérateurs graphiques, il y'a lieu de mettre en avant celle présentée dans [142], où les auteurs ont discutés si le comportement collectif dans un système évolutionnaire de robotique en essaim (ESR) peut être efficacement mis en œuvre par des techniques exploitant le GPU Computing. Comme étude de cas, ils ont implémenté le problème de recherche de nourriture. Dans l'environnement informatique expérimental utilisé, le GPU a joué un rôle non négligeable pour le traitement parallèle du ESR, en particulier lors du traitement de capteurs indépendants équipant les robots. Par rapport à la méthode classique, la méthode proposée a permis d'obtenir des temps de traitement dont les facteurs de réduction était de l'ordre de 2.5, 3.0, 2.9, et 2.8 pour des essaims de tailles 8, 16, 24 et 32 respectivement. La méthode proposée conférait son plus grand avantage dans l'opération de détection, qui a pu être accélérée avec un facteur de 11.3 par rapport à la méthode standard.

Un autre travail présenté par Jones et coll. [99], dont le but est d'accélérer la simulation des robots en allant vers la fourniture d'un robot autonome avec une capacité incorporée nommé what-if.

4.6 Facteurs limitatifs des modèles actuels

La section précédente a présenté brièvement plusieurs différents modèles de contrôleurs évolutifs sur des machines parallèles. Dans un premier temps, il y'a lieu de faire le constat qu'ils ne nous aient permis une meilleure compréhension scientifique, leur complexité et application n'ayant pas subi de changements au cours de ces dernières années.

Concrètement, les tâches expérimentales ainsi que l'échelle des contrôleurs évolutifs (Réseaux de neurones simples, ou Réseaux de neurones évolués avec une technique évolutionnaire) pour les mouvements (les actions) sont souvent réduites au minimum pour éviter des temps d'entrainement excessifs qui se développent de façon exponentielle avec les données d'apprentissage ainsi que les détails des paramètres associés à la technique utilisée (tel que : le nombre de neurones). Tani et Nolfi [182] ont utilisé un modèle de réseaux de neurones récurrents à deux niveaux. Pour le niveau inférieur, le réseau était composé de cinq modules RNN, chacun avec 18 neurones et 5 modules RNN avec 17 neurones chacun au niveau supérieur. Les auteurs ont utilisé un robot mobile simulé avec 20 capteurs de distance laser. Cependant, les auteurs déclarent que : "Seulement 6 des 20 valeurs de senseurs sont utilisées pour l'apprentissage du RNN dans le but de réduire le temps de calcul". Les expériences de la robotique évolutive menées par Massera ont introduit le module CTRNN. Plus complexe, il est constitué de 64 neurones et 23 moteurs de contrôle. Le temps d'entrainement des CTRNN de cette complexité est typiquement autour d'un jour (5000 générations). Cependant, si la tâche devient plus complexe avec un CTRNN plus complexe, le temps d'entrainement pourrait facilement atteindre des jours voire des semaines.

Ces études mettent en évidence quelques états de l'art des modèles de réseaux neuronaux pour l'acquisition de mouvements. On peut remarquer que les initiateurs sont

souvent obligés de simplifier leurs modèles de réseaux neuronaux ainsi que les tâches expérimentales afin de rendre l'entrainement possible. Cependant, de nombreuses études ont démontré que l'utilisation de réseaux de neurones à grande échelle présente un grand potentiel pour étendre la complexité des contrôleurs neuronaux [85], [169] et leur capacité d'apprendre [113]. Sans surprise, beaucoup ont tourné leur attention vers le traitement distribué en utilisant des clusters de CPU et plus récemment sur des processeurs GPU parallèles, qui ont abouti à de grandes cadences et ont permis l'exploration de nouveaux domaines.

À notre connaissance, cette thèse est parmi les premières expériences où l'application du GPU Computing pour la robotique évolutionnaire a produit des contrôleurs évolutifs efficaces, capables de contrôler chaque degré de liberté du robot humanoïde pendant l'acquisition de l'information, et la génération des mouvements désirés.

4.7 RNAs exploitant la programmation sur GPU pour les contrôleurs des robots humanoïdes

Les RNAs représentent l'un des outils ayant servi à imiter la phase d'apprentissage du système nerveux humain. Cependant, la principale différence vient du fait que le système nerveux est massivement parallèle [153], tandis que le processeur de l'ordinateur reste significativement séquentiel.

Étant donné que les RNAs nécessitent un nombre considérable d'opérations vectorielles et matricielles pour obtenir des résultats, ils sont très aptes à être mis en œuvre dans un modèle de programmation parallèle et à fortiori à envisager leur implémentation sur une architecture de type GPU.

CUDA a été employé dans une grande variété d'applications, néanmoins, peu d'entre elles examinant cette technologie pour la neuro-robotique évolutionnaire. Seules quelques tentatives ont mis en œuvre les réseaux de neurones. Ceci est dû au fait que ce domaine nécessite une étude plus approfondie dans l'application de la technologie CUDA pour le traitement neuronal artificiel évolutif et la recherche en robotique [149], [35].

Peniak a présenté un nouveau logiciel nommé « Aquila » [149], utile pour la recherche dans le domaine de la robotique cognitive et développementale. Aquila répond au besoin de la haute performance du contrôle de robots en utilisant le dernier paradigme de traitement parallèle, basé sur la technologie CUDA de NVIDIA.

González-Nalda a visé à travers l'évolution du réseau de neurone [69], qu'il couple avec un corps complexe d'un robot humanoïde, où le réseau neuronal utilisé a été plongé dans le modèle de programmation CUDA. Dans ce travail, les problèmes d'un environnement non structuré ainsi que ceux de la robotique évolutionnaire requièrent une approche connexionniste sous-symbolique basée sur l'intelligence artificielle, qui peut faire face à l'accouplement entre les sensorimoteurs, les neurones et les parties de l'environnement.

Divers types de réseaux de neurones sont utilisés pour générer des comportements de la marche ainsi que pour la conception des contrôleurs des robots humanoïdes, ou quelques-uns d'entre eux ont étudié le traitement parallèle avec les GPUs. Cependant, ces approches basées RNAs ne permettaient pas la construction d'un contrôleur qui s'adapte avec le corps du robot humanoïde complexe.

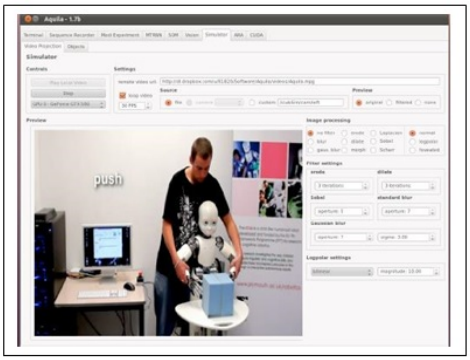


FIGURE 4.2 – Interface de Aquila Toolkit.

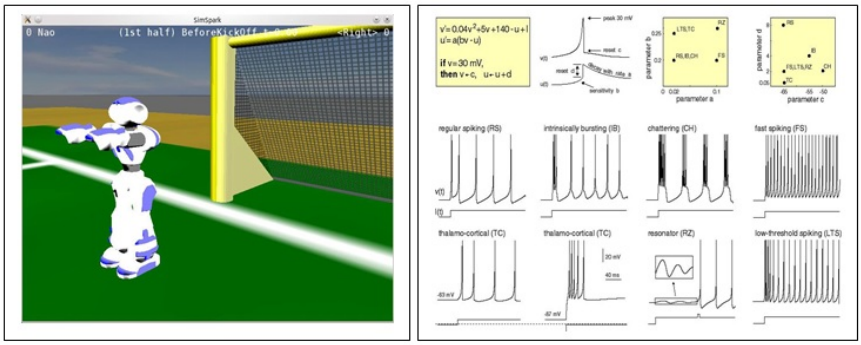


FIGURE 4.3 – A propos de Nalda proposition.

Pour surmonter cette lacune, nous proposons dans notre étude actuelle l'intégration d'une stratégie évolutionnaire (SE) basés CUDA, pour concevoir le contrôleur du robot. Les SEs soulèvent un grand intérêt dans leur capacité à un grand intérêt dans leur capacité de trouver des solutions qui ne sont pas nécessairement optimales, mais convenables pour des problèmes complexes.

4.8 Simulateurs physiques à base GPU

Au cours de la dernière décennie, la simulation physique est devenue une technologie clé pour différentes applications. Elle a pris un rôle important dans les jeux informatiques [83], l'animation de mondes virtuels [151] et la simulation robotique [203]. D'autre part, les logiciels de conception automatique de contrôleurs ne sont pas intégrés avec les moteurs physiques [185].

Les robots et les moteurs physiques sont composés d'une multitude de composants nécessitant une grande précision dans certaines parties, alors que dans d'autres domaines, le temps de calcul représente l'élément clé indispensable. Ainsi, la vitesse de simulation est l'élément directeur lors d'une étude sous GPU. Pour atteindre l'accélération exigée, certains moteurs physiques ont déjà recours aux accélérateurs graphiques [100], [207] à travers les fonctionnalités de CUDA ou d'OpenCL, telles que l'accélération de la détection

de collision entre les corps rigides et l'interaction des particules. Le fait d'affecter certains calculs physiques au GPU est basé sur deux constats principaux :

1. Il est possible de décharger le CPU de certains calculs typiques, afin qu'il puisse effectuer d'autres calculs.
2. Il est possible d'optimiser le moteur physique afin qu'il puisse supporter plus d'organismes rigides dans la simulation.

En 2009, Zamith et coll. [207] ont présenté une version modifiée du simulateur ODE (Open Dynamic Engine) qui tire profit du potentiel des moteurs graphique. Pour la réalisation de simulations de corps rigides avec certaines de ses méthodes implémentées sur le GPU, il a proposé l'implémentation de nouvelles méthodes de distribution automatique de processus entre le CPU et le GPU. En effet, le GPU a été utilisé dans ce travail comme en tant que coprocesseur mathématique pour assurer le critère temps réel pour des jeux spéciaux et dans des simulations physiques avec l'intention de supporter les lois mathématiques et physiques accélérées. L'architecture proposée est efficacement exploitable pour les langages Shaders ainsi que pour les langages des processeurs graphiques à usage général (GPGPU).

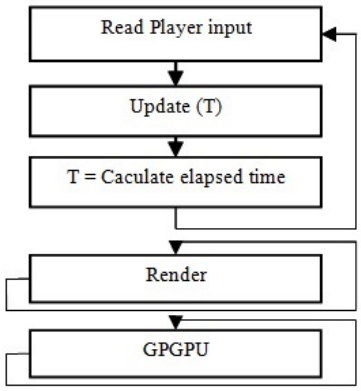


FIGURE 4.4 – Architecture de la multithread découplé avec le modèle de GPGPU [207].

Dans notre étude, nous adoptons l'architecture GPGPU proposé par Zamith et coll. (2009) afin de garantir l'utilisation du GPU comme un coprocesseur mathématique et physique qui accélère le calcul de lois mathématiques et physiques utilisées dans les mouvements devant être générés par le robot humanoïde.

4.9 Hybridation SE-RNN du contrôleur du robot exploitant le GPU : Conception et mise en œuvre

Cette section présente les détails de la proposition d'une hybridation SE-RNN du contrôleur exploitant le GPU, proposé dans [7]. Nous démarrons avec un aperçu de notre proposition, suivi de la présentation du modèle du robot humanoïde utilisé (c.-à-d. les primitives géométriques, les articulations, la hiérarchie de mémoire utilisée pour le stockage des primitives du robot). L'évolution des paramètres des Kernels et de stockage de données est introduite, et sera suivie par l'illustration de l'architecture proposée. Enfin,

nous concluons cette section en analysant la complexité du système proposé et la justification du processus de conception et des choix arrêtés au cours de la mise en œuvre de cette proposition comme la répartition des tâches et la gestion de la mémoire.

4.9.1 Aperçu général

Pour faire face à l'aspect critique du paradigme de programmation GPU, et dans le but d'obtenir un gain considérable ou une accélération de l'évolution de notre contrôleur, nous proposons une nouvelle architecture hybride basée sur la SE et le RNN conçu sur une plateforme GPU. Ce modèle adopte l'architecture GPGPU proposée dans [207] afin d'exploiter la grande précision de calcul de ce type de dispositifs, avec l'intention de soutenir l'accélération des lois mathématiques et physiques, dans la simulation.

Concernant la spécification du RNN, le robot humanoïde étudié utilise le réseau de neurones récurrent d'Elman pour sa plausibilité biologique et ses capacités de mémorisation puissantes. Bien que les RNAs biologiques ne se servent pas de la propagation arrière de l'apprentissage, nous proposons d'utiliser une stratégie évolutionnaire basée GPU pour évoluer et faire converger le comportement de locomotion. Pour cette tâche, nous devons exploiter un grand réseau neuronal récurrent pour connecter et simuler les muscles et un système moteur dans un robot humanoïde avec des dizaines de degrés de liberté. Le réseau de neurones doit posséder plusieurs milliers de neurones, car il est plus facile d'extraire et de mettre les informations correctes pour chaque articulation. Le nombre de nœuds dans les couches cachées du RNN est limité par la capacité de la mémoire associée au GPU. Cette hybridation (RNN + SE) exploite le parallélisme à un niveau supérieur, où des groupes de threads sont formés pour manipuler un seul chromosome qui correspond à chaque robot.

L'objectif de la stratégie d'évolution est d'optimiser les poids du RNN qui contrôle la marche du robot. Une relation synergique existe entre la SE et le RNN comme illustré dans la figure 4.5. La SE optimise le RNN et le RNN produit le comportement du robot qui est ensuite évalué et classé. Au démarrage, la simulation est lancée sur le CPU, les paramètres de l'évolution sont copiés dans la mémoire constante du GPU, les centres des positions des pieds des robots sont copiés dans la mémoire globale du GPU alors que les chromosomes de la population sont initialisés aléatoirement avec un (01) gène qui correspond à un poids du RNN. Le nombre de connexions représente le nombre de gènes dans le chromosome tandis qu'un nombre réel représente chaque gène. Les caractéristiques des valeurs pour toutes les instances d'évolution de l'apprentissage sont transférées vers la mémoire principale du dispositif (GPU) afin qu'ils soient prêt pour les calculs SE-RNN. À chaque pas de simulation, le dispositif exécute les étapes décrites dans la figure 4.5.

Après l'initialisation de la partie GPU et pour chaque génération, tous les senseurs du robot sont activés afin de recueillir des informations à partir de l'environnement de simulation. Ces valeurs représentent les entrées du RNN. Par conséquent, les sorties du RNN seront mises à jour en utilisant notre proposition d'évolution parallèle. Pour atteindre cet objectif, la répartition des tâches élémentaires (threads) est gérée de façon judicieuse en utilisant l'algorithme proposé ainsi qu'une gestion efficace de la mémoire disponible dans la hiérarchie du GPU (chapitre 5).

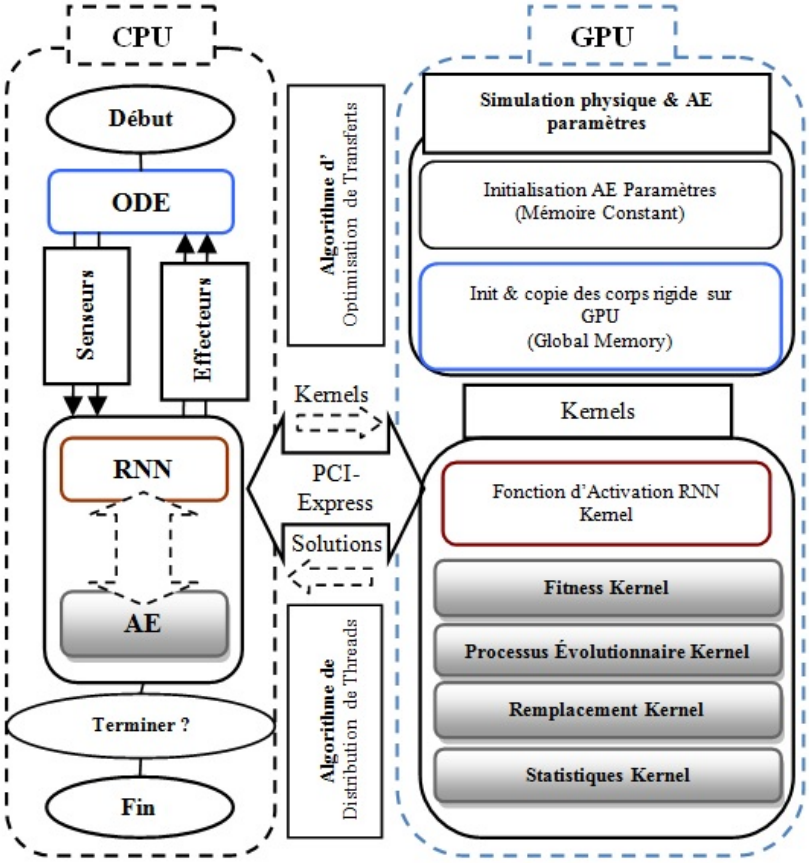


FIGURE 4.5 – Configuration du RNA et SE avec ODE sous le GPU.

En outre, la nouvelle population (récupérée à partir de la dernière génération) représente la nouvelle entrée du RNN. Nous pouvons remarquer ici que les senseurs représentent les parties en contact direct avec l'environnement de simulation. Ces senseurs fournissent des données provenant de l'environnement comme le contact avec le sol, tandis que les effecteurs ont la capacité d'interpréter les valeurs de sorties du RNN (les muscles qui sont représentés dans notre cas par quelques primitives géométriques). Les Kernels mentionnés dans la partie GPU représentent le calcul évolutif fonctionnant sur le périphérique.

Les détails techniques concernant la mise en œuvre du GPU-SE, seront présentés par la suite. Commençons avec l'initialisation de la stratégie évolutionnaire (SE) de la première population qui se fait de manière parallèle dans le but d'augmenter le taux d'utilisation de la carte graphique qui est un facteur influant directement sur la performance globale.

La simulation du processus évolutif basé GPU, est décrite par le pseudo-code (1). La partie suivante décrit la distribution de la population sur la carte ainsi que les détails de l'implémentation des Kernels.

4.9.2 Modèle du robot humanoïde et de stockage de données

4.9.2.1 Simulateur physique : ODE (Open Dynamique Engine)

ODE est une bibliothèque open source pour simuler la dynamique et les collisions des corps rigides d'une manière efficace et précise [59]. Il consiste d'une bibliothèque de haute

Algorithme 1 Processus de simulation sous GPU.

Télécharger les paramètres de simulation et d'initialisation des robots.
Initialiser les RNNs.
Générer la première population évolutionnaire des solutions.
Initialiser la SE.
Allocation des entrées du problème dans la mémoire du GPU.
Allocation de la population dans la mémoire du GPU.
Allocation des structures de fitness dans la mémoire du GPU.
Copie des entrées de problème sur la mémoire du GPU ;
Copie la population généré sur la mémoire du GPU ;
Copie des structures supplémentaires sur la mémoire du GPU ;
Évaluation les valeurs de fitness de tous les individus de la population en utilisant une mesure sur la base de la fonction objectif à optimiser ;
répéter
    pour tout les individus **faire**
        Évaluation de l'individu ;
        Insertion des fitness résultantes dans leurs structures.
    fin pour
    Copie les structures de la fonction de fitness sur la mémoire du CPU ;
    Appliquation des opérateurs de recombinaison ;
    Construction de la nouvelle population (Robots) ;
    Remplacement de la population parente par la nouvelle population descendante ;
    Copie la nouvelle population sur la mémoire du GPU ;
    Copie de la structure supplémentaire dans la mémoire du GPU ;
jusqu'à ce que le critère d'arrêt soit atteint

performance pour la simulation de la dynamique des corps rigides écrite une simple API C/C++.

Bien que le moteur ODE soit un moteur fiable dans la simulation physique, le calcul de toutes les interactions physiques d'un système complexe peut exiger énormément de puissance de traitement. Comme l'ODE utilise un moteur de rendu simple basée sur OpenGL, il a des restrictions pour le rendu des environnements complexes comprenant de nombreux objets et des corps. Ceci peut affecter de manière significative la vitesse de simulation des expériences complexes de la simulation robotique [9].

Algorithme 2 Structure algorithmique d'une application typique d'ODE, De [166].

1: Créer un environnement dynamique ;
2: Créer les organismes dans l'environnement dynamique ;
3: Définir le statut (position et vitesse) de tous les organismes ;
4: Créer les articulations (contraintes) qui relient les organismes ;
5: Créer un monde de collision et la géométrie de collision des les objets ;
6: **tantque** $temps < temps_{max}$ **faire**
7: Appliquer les forces nécessaire pour les organismes ;
8: Appeler la détection de collision ;
9: Créer un contact de jointure pour chaque point de collision, et le mettre dans le groupe de contact de jointures ;
10: Prendre une étape de simulation vers l'avant ;
11: Retirez tout le groupe de jointure ;
12: Avancer le temps : temps = temps + Δ_t ;
13: **fin tantque**

4.9.2.2 Modèle du robot humanoïde

Notre démonstration s'appuie sur un modèle robotique, basé sur des primitives disponibles dans la bibliothèque de simulation ODE, qui offrent un environnement contrôlé avec ou sans obstacle. Un modèle physique de locomotion décrit les relations non linéaires entre les forces, les moments agissant sur chaque jointure, les pieds, la position, la vitesse et l'accélération angulaire de chaque articulation. En plus des données géométriques, un modèle dynamique nécessite les masses des éléments cinématiques, les centres des masses et les matrices d'inertie pour chaque liaison, a besoin également de jointure, des valeurs min/max des couples moteurs et les vitesses angulaires qui sont difficiles à obtenir et qui représentent souvent une source d'imprécision négligée pour la simulation. Toutes ces données sont copiées dans la mémoire principale du GPU à l'effet d'effectuer un maximum de calculs sur celle-ci.

Pour simuler l'interaction avec l'environnement, la détection et le traitement des collisions ainsi que les modèles appropriés des contacts avec le sol pour les pieds sont nécessaires. Pour s'adapter au monde de la simulation et après avoir effectué plusieurs essais, nous avons fixé les paramètres expérimentaux conformément à ceux mentionnés sur la table (4.1), ces valeurs représentant les meilleures valeurs empiriques de notre robot.

Body part	Geometry primitive	Dimendion (m)
Head	Sphere	Radius :0.188
Arm	Caped cylinder	0.14x0.25x0.44
Torso	Rectangular box	0.9x0.25x1.0
Thigh	Caped cylinder	0.20x0.25x0.7
Shank	Caped cylinder	0.20x0.25x0.7
Foot	Rectangular box	0.4x0.5x0.1

TABLE 4.1 – Paramétres du corps du robot.

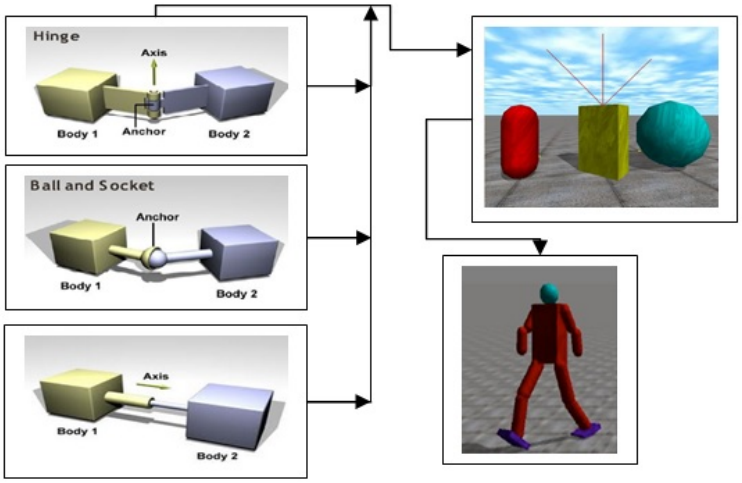


FIGURE 4.6 – Primitives géométriques, les jointures, et le modèle 3D du robot.

4.9.2.3 Invocation des Kernels, population de la SE et paramètres des données de stockage

Avec plusieurs types de mémoires, une utilisation incorrecte des capacités de la mémoire offerte et disponible au sein du GPU peut influer sur l'accélération désirée. Ainsi, et en général, la question posée concerne plutôt le type correct de mémoire devant être utilisée pour chaque phase de l'implémentation. Une fois la structure créée au niveau de la partie hôte, elle sera copiée dans la mémoire constante du GPU, car ce type de mémoire est utilisé spécifiquement pour les cas où les données ne changeront pas à l'exécution d'un Kernel évolutionnaire, y compris la taille de la population, la taille des chromosomes, le taux de mutation, le taux de croisement, le nombre de générations, les données statistiques (moyenne, maximum, et minimum des valeurs de la fonction de fitness), etc.

La population de la stratégie évolutionnaire est aménagée dans la mémoire principale du processeur graphique, comme une matrice à deux dimensions N*L, où les colonnes réfèrent aux chromosomes et les lignes correspondent aux gènes à l'intérieur des chromosomes. Dans notre cas, N est la taille de la population et L est la longueur des chromosomes, en tenant compte du fait que le stockage des variables de la même séquence individuelle dans un tableau ne permet pas un accès efficace à la mémoire.

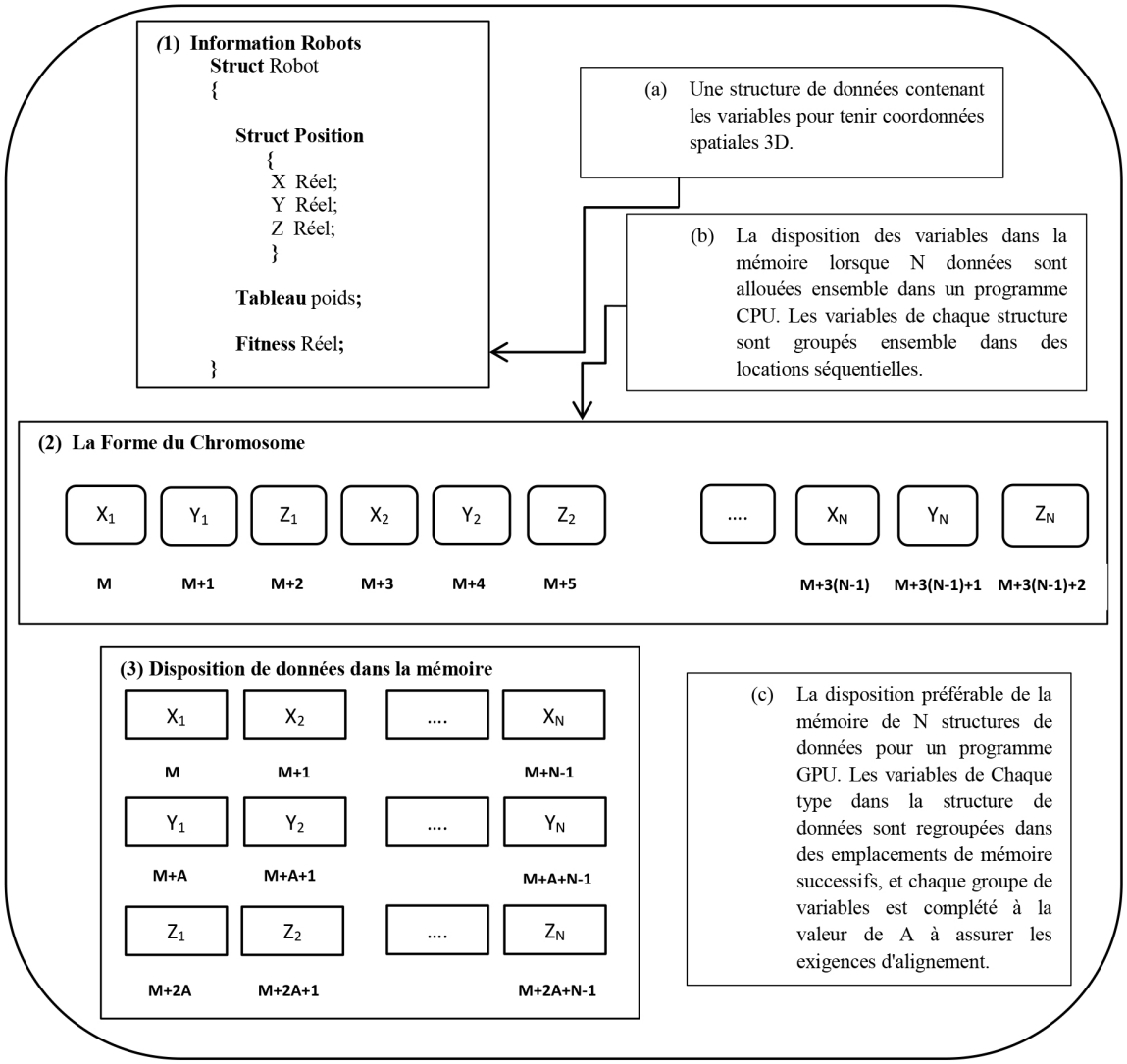


FIGURE 4.7 – Organisation de structure de données dans la mémoire.

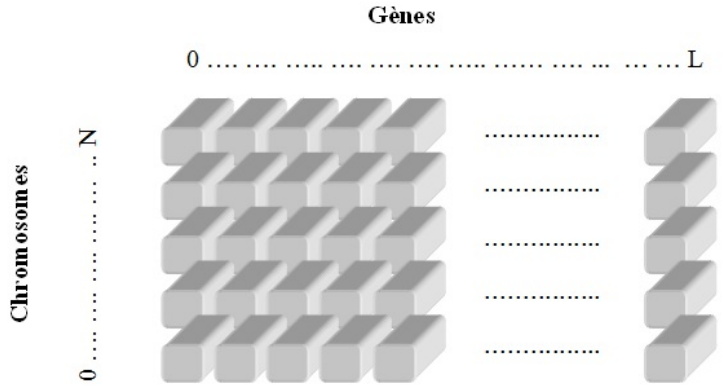


FIGURE 4.8 – Représentation de la population dans la mémoire GPU.

Ainsi, pour atteindre la coalescence de la mémoire, les variables du même type et appartenant à des individus différents de la population sont stockés dans des buffers adjacents.

4.9.3 Architecture hybride SE-RNN du Contrôleur du Robot

Comme mentionné ci-dessus, le GPU, dans de telles applications, est utilisé en tant que coprocesseur, ou les phases de calcul intense sont affectées au GPU telles que le processus évolutif du RNN avec quelques calculs de la simulation physique.

4.9.3.1 Configuration et paramétrage des Kernels

Pour trouver la meilleure configuration des paramètres (la forme des blocs, le nombre total de threads) des Kernels évolutionnaires, nous proposons un algorithme permettant de sélectionner à la fin de son exécution la configuration la plus appropriée, celle-ci devant maximiser en même temps le taux d'occupation de la carte utilisée et minimiser le temps pour trouver les solutions. Le principe de notre stratégie de distribution suppose que le nombre de blocs (correspondant au nombre de chromosomes) ne doit pas être supérieur au nombre des MPs existants. En outre, le nombre de threads est le nombre de poids dans chaque RNN, où la capacité des blocs de la carte utilisée doit être respectée (la taille max de bloc). Plus de détails sur l'algorithme utilisé, ainsi que sur la distribution sur la carte seront présentés dans le chapitre suivant.

4.9.3.2 Modules d'hybridation SE-RNN basée GPU.

Notre système est composé de trois modules, le premier étant le module de simulation qui est responsable de la tâche de la simulation dans l'environnement ODE en tenant compte de toutes les lois nécessaires de la physique. Le second est le module du contrôle (système neuronal artificiel), qui est responsable de créer le cerveau du robot à l'aide du RNN. Le troisième et dernier représente le module principal (module chargé de l'évolution du RNN) de notre système dans lequel toutes les phases de la stratégie évolutionnaire sont implémentées parallèlement sur le GPU comme des Kernels.

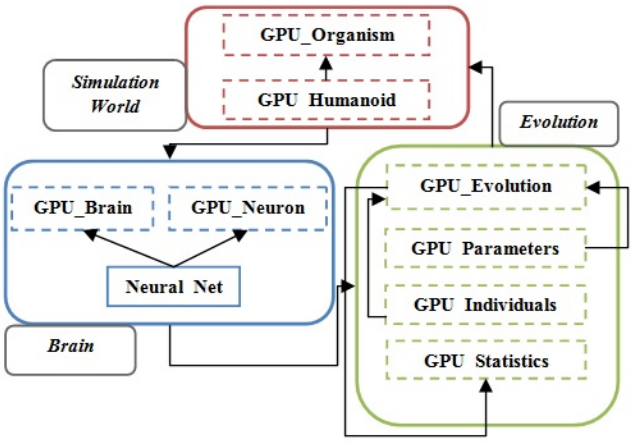


FIGURE 4.9 – Modules principaux du modèle proposé.

Dans les prochains paragraphes, les Kernels du module d'évolution seront détaillés.

4.9.3.3 Gestion de la mémoire

Il est indispensable de réduire au minimum les transferts des données entre le CPU et le GPU puisqu'il y a un volume important d'informations échangées entre les modules proposés, ceux-ci étant le module la simulation, le module de contrôle (système nerveux) et le module chargé de l'entrainement évolutionnaire, ces modules seront expliqués en détail dans les paragraphes suivants. Nous proposons la réduction du transfert de données par un sous-module inspiré des travaux de Becchi [6] et Yang [205] en prenant les avantages des règles d'optimisation d'accès à la mémoire globale par la coalescence, et la capacité d'ordonnancement pour le recouvrement des latences de la mémoire. Les programmeurs exploitant le GPGPU considèrent que la coalescence fait référence au besoin d'accéder à un demi-Warp (c.-à-d. 16 threads consécutifs peuvent être fusionnés en un seul accès mémoire).

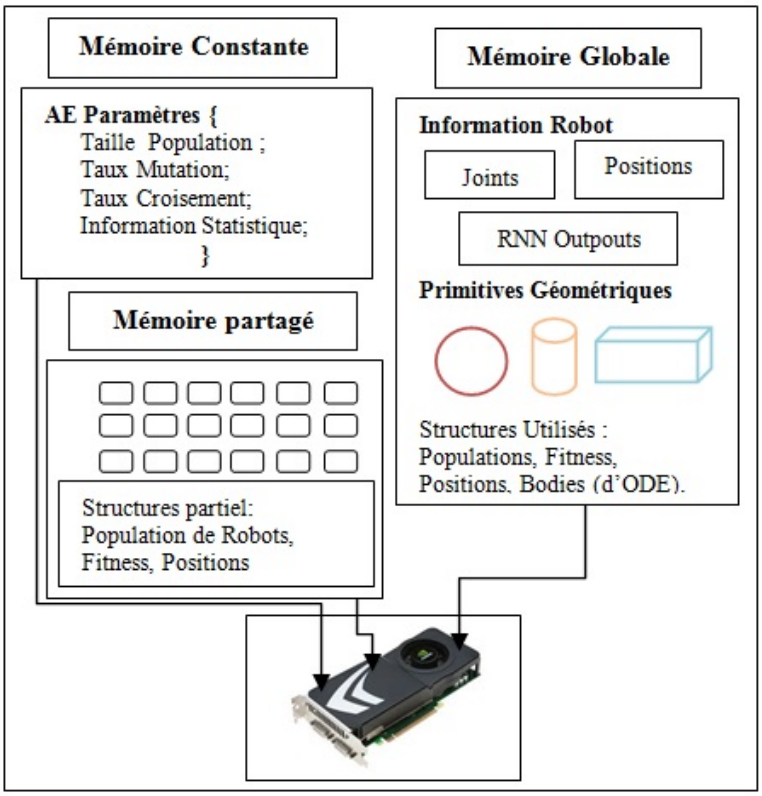


FIGURE 4.10 – Gestion de la mémoire du processus de simulation (le Moteur Physique, l'entrainement évolutive formation).

Notre proposition de réduire au minimum le taux de transfert entre les deux sous-systèmes (CPU-GPU) est basée sur la coalescence de la mémoire où l'idée de base du Mapping entre les mémoires CPU-GPU du processus évolutionnaire est la vectorisation des structures de données utilisées (la structure de la population, la celle de la fonction de fitness, etc.) aidant à sauvegarder toutes les données sur des vecteurs ou des matrices 2D pour faciliter le processus de coalescence. Par conséquent, le mappage des données sur la

mémoire du GPU est effectué en utilisant CudaMalloc, le transfert des données est assuré à l'aide de CudaMemcpy où nous vérifions avant tout transfert, si ces données résident dans la mémoire du GPU ou non.

Avant d'exécuter les Kernels évolutionnaires, la coalescence de données est vérifiée à l'aide des règles définies par Yang [205] (Seront présenté dans le chapitre suivant).

À la fin de ce processus, les données sont libérées à l'aide de CudaFree, soit à la fin de l'application ou lorsque la mémoire du GPU s'avère pleine.

Les détails de l'algorithme utilisé au niveau de cette étape auront lieu dans le chapitre suivant.

A. Génération des nombres aléatoires

Un des facteurs qui affectent la performance des algorithmes évolutionnaires consiste en la génération de nombres aléatoires, du fait que ces algorithmes sont des processus de recherche stochastique. Cependant, la bibliothèque CUDA ne comprend pas toutes les fonctionnalités d'un générateur de nombres aléatoires (RNG) à l'heure actuelle, alors que le RNG est naturellement nécessaire à l'exécution des processus évolutionnaire. Afin de pallier cette défaillance, et générer des nombres aléatoires dans notre application, nous utilisons la bibliothèque Random123 de « counter-based » (CBRNGs) comme décrite par Salmon et coll. (2011) [163]. La bibliothèque Random123 peut être utilisée avec les applications CPU (C et C++) et GPU (CUDA et OpenCL). La version utilisée dans cette implémentation dispose de plusieurs familles (Threefry, Philox, AESNI, ARS). Cette bibliothèque est choisie en raison de la rapidité de son générateur, ses potentialités de parallélisme et l'utilisation minimale des ressources de la mémoire traditionnelle relativement aux ressources de la mémoire cache (trois fois plus rapide que la fonction rand du C standard et 10 fois plus rapide que le générateur cuRand de CUDA).

B. Kernel d'initialisation de la population

Dans chaque processus évolutif, la première étape est l'initialisation de la population. Une des questions qui se posent pour cette étape est : où est-ce que les chromosomes seront générés ? Pour répondre à cette exigence, deux approches principales [190] sont proposées dans la littérature :

1. *Génération de la population sur CPU et Évaluation sur GPU :* À chaque itération du processus évolutionnaire, la population est générée sur le CPU. La structure qui lui est associée et qui mémorise les solutions est copiée sur le GPU. Ainsi, les transferts des données sont essentiellement l'ensemble des solutions de population copiées du CPU vers le GPU.

2. *Génération et évaluation de la population sur le GPU :* Dans cette approche, la population est générée sur GPU. Ce qui implique qu'aucune structure explicite ne doit être allouée. De ce fait, seule la représentation de la solution doit être copiée du CPU vers GPU. Ainsi, comme bénéfice de cette approche nous pouvons noter réduction considérable des transferts de données, puisque l'ensemble de la population ne doit pas être copié. Cependant, la mise en correspondance entre un thread et un individu eut s'avérer une tâche difficile.

Dans notre cas et comme mentionnés ci-dessus, les éléments de chaque chromosome représentent les poids du RNN, l'opération d'obtention de ces valeurs est entièrement parallélisable et tous les chromosomes sont initialisés en même temps. Cette phase est réalisée par un bloc de threads pour chaque chromosome, comme le montre figure (4. 11).

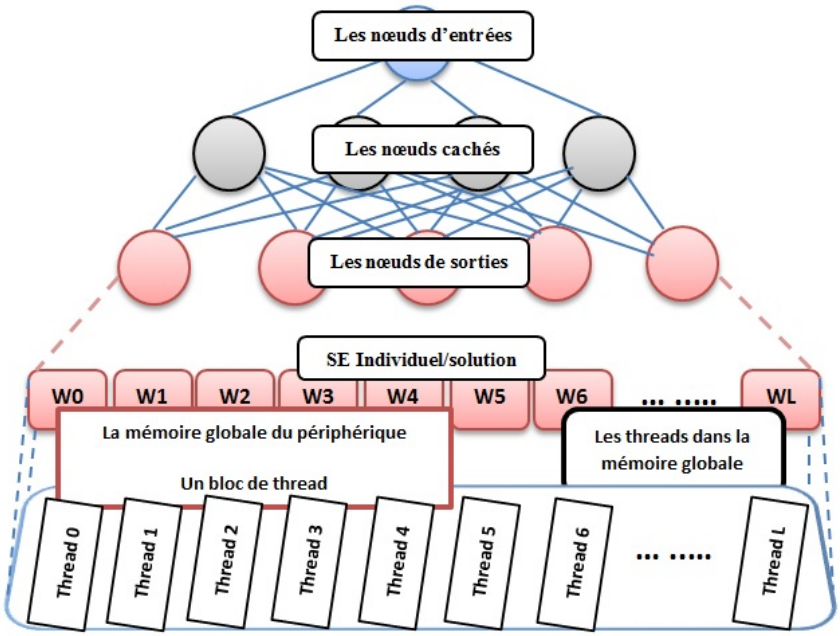


FIGURE 4.11 – Mappage de chromosome sur les blocks.

C. Kernel de la sélection

Comme la plupart des schémas de sélection qui travaillent avec des probabilités et prélèvent des échantillonnages aléatoires de la population, et même promouvoir l'échantillonnage multiple du même individu, les prélèvements peuvent être effectués en parallèle [86].

Notons, dans notre cas, que la forme modifiée de la sélection de la roulette pipée est utilisée dans la phase de sélection. Il existe peu de travaux ayant contribué à la mise en œuvre de la sélection de la roulette pipée sur GPU. Pour sa part, Dawson et Stewart ont prouvé dans [30], qu'un algorithme exploitant la sélection de roulettes hautement parallèle doit respecter le parallélisme au niveau du warp, ce qui réduit les dépendances de la mémoire partagée.

Pour simuler le Kernel de la sélection de la roulette pipée dans notre cas, la population des robots est triée (méthode GPU). La fonction de fitness est calculée en fonction de la distance parcourue, en utilisant le Radix-Sort basé GPU, représentant un utilitaire rapide fourni et décrit dans [166]. Ces scores sont normalisés pour calculer les probabilités de sélection. Nous contribuons dans cette phase avec l'utilisation d'un bloc de threads pour chaque chromosome dans le processus. Une somme de réductions est effectuée sur un tableau normalisé qui contient les valeurs collectées. Ce nouveau tableau est stocké dans la mémoire globale et utilisé comme un tableau de la roulette.

D. Kernel du processus évolutionnaire

Ce Kernel vise à actualiser l'ancienne population, en sélectionnant toutes les paires de chromosomes qui vont subir le processus de croisement pour produire les descendants (les enfants). Nous contribuons dans cette phase par l'application d'un croisement massivement parallèle en un seul point, en plus de la mutation. Ces opérateurs sont contrôlés respectivement par les probabilités de croisement et de mutation.

Pour concevoir une implémentation efficace du Kernel de manipulation génétique, une divergence faible d'accéder à la mémoire plus une quantité considérable de données pour utiliser tous les cœurs CUDA sont requises.

Afin d'implémenter le croisement en un point, nous devons avoir deux Kernels, le premier sert à trouver le point de croisement. Nous suggérons ici d'assigner un thread pour chaque chromosome pour surmonter le problème de synchronisation entre les blocs CUDA, produit dans le cas d'attribution d'un bloc de thread pour chaque chromosome, ce qui signifie que tous les threads doivent avoir l'information concernant le point de croisement. Le second sert à retourner les nouveaux descendants. Pour ce faire, chaque bloc CUDA est organisé en deux dimensions. La dimension x correspond aux gènes d'un seul chromosome, tandis que la dimension y correspond à des chromosomes différents. L'ensemble de la grille est organisé en deux dimensions avec une taille de x fixé à un (01), et la taille de y correspondant à la taille de la population des descendants (threads) divisée par la taille des blocs de y multipliée par deux (02).

Pour le Kernel de mutation, la distribution de threads est réalisée de sorte que chaque thread occupe un gène, l'objectif de ce processus est d'éloigner la stratégie évolutionnaire d'un minimum ou maximum locaux.

E. Kernel de la Fonction de Fitness

Dans cette proposition, une méthode totalement parallélisée est proposée pour le processus d'évaluation afin de réduire la complexité de calcul, tant cette étape est la plus importante et est généralement plus consommatrice en termes de temps. Dans notre cas, le meilleur individu est le robot capable de traverser la plus grande distance en un temps bien défini. Dans ce problème, la fonction de fitness est basée sur la position initiale et finale pour chaque robot (individuel).

La même décomposition du processus évolutionnaire est utilisée, où chaque warp est copié vers la mémoire partagée, puisque cette mémoire peut fournir une accélération considérable en conservant la bande passante vers la mémoire globale, en raison des caractéristiques de la mémoire cache (User-Managed) [26] lors du calcul de la distance dans le GPU.

F. Kernel de remplacement

Cette phase utilise le processus de sélection déjà décrit (la version modifiée de la roulette pipée) pour les parents ainsi que pour les enfants afin de créer une nouvelle population parente. Les paramètres utilisés pour le Kernel sont les mêmes que ceux utilisés dans les phases précédentes avec une modification dans les dimensions des Kernels qui sont dérivées de la taille de la population mère.

G. kernel des statistiques

Pour chaque génération, après le calcul des valeurs de la fonction de fitness, les statistiques doivent être mises à jour. Les statistiques de la population sont utilisées pour le processus de sélection et pour le processus de la terminaison de la décision. Les valeurs maximales et minimales de la fonction de fitness, la moyenne et la déviation constituent les éléments de la structure des données statistiques.

4.9.4 Analyse de complexité du problème

Trois parties sont considérées pour calculer la complexité de cette proposition : la partie simulation, la partie système nerveux et la partie évolution. Dans le cadre de la simulation, la complexité de calcul de l'ODE est d'ordre O (n^2), où n est le nombre d'objets (des primitives géométriques). En outre, la complexité du réseau de neurones est d'ordre O (2^{L-1}) où L est le nombre de couches cachées de réseau neuronal. En ce qui concerne la partie évolution, la complexité dépend des étapes et des opérateurs évolutionnaires. Comme les opérateurs utilisés sont la mutation ponctuelle avec un croisement en un point plus la sélection en utilisant la roulette pipée, la complexité de la partie évolutionnaire est de : O (g. (t. m + t. m + n) = O (g. t. m). Avec g est le nombre de génération, t est la taille de la population et m est la taille de l'individu.

Par conséquent, la complexité du calcul global de l'algorithme proposé est calculée en additionnant la complexité des trois parties est d'ordre : O (g. t. m).

4.10 Expérimentation, Résultats et Discussion

Nous rappelons que le premier but de cette étude est de démontrer le potentiel de l'utilisation de dispositif GPU pour les SEs utilisées pour faire évoluer un réseau de neurones récurrent. Pour ce faire, nous avons effectué diverses expériences en modifiant les différents paramètres de la SE, à savoir la longueur du chromosome, la taille de la population, le nombre de générations et les paramètres sensibles des Kernels exécutés (la forme du bloc, le nombre de threads). Le gain a été mesuré sur l'ensemble de la SE. Lors de nos tests, les temps de différentes phases de la SE sont mesurées sur le CPU ainsi que sur le GPU. Pour plus de précision, le temps de transfert de la population, de et à partir du GPU, est ajouté au temps d'exécution des Kernels.

4.10.1 Configuration de l'environnement d'exécution

Nous avons mis en œuvre le modèle proposé en C/C++ et le framework CUDA. Ces expériences sont exécutés sur un PC avec un processeur (Intel Core i7 870) et deux GPUs (NVIDIA GeForce GTX 480).

Pour la SE, nous avons utilisé un taux de croisement fixé à 0.7, le taux de mutation génomique égal à 0.01, le taux d'élitisme 0.2 , la méthode de sélection de la roulette pipé est utilisé et au moins de 100 individus pour chaque population. La simulation est effectuée jusqu'à ce qu'à l'obtention des mouvements adéquats pour les robots.

4.10.2 Convergence de la proposition évolutive

Tout d'abord et avant tout, tout effort de parallélisation doit garantir que la solution finale atteinte est similaire ou meilleure en qualité à celle obtenue à partir de l'algorithme séquentiel. Premièrement, nous allons démontrer que notre approche évolutive parallèle est en mesure de trouver une solution qui est très proche de la solution optimale. Spécifiquement, la solution optimale attendue est liée aux meilleures valeurs des poids du RNN qui conduisent le robot à développer des mouvements réels très semblables à ceux d'un humain réel pour effectuer des tâches complexes.

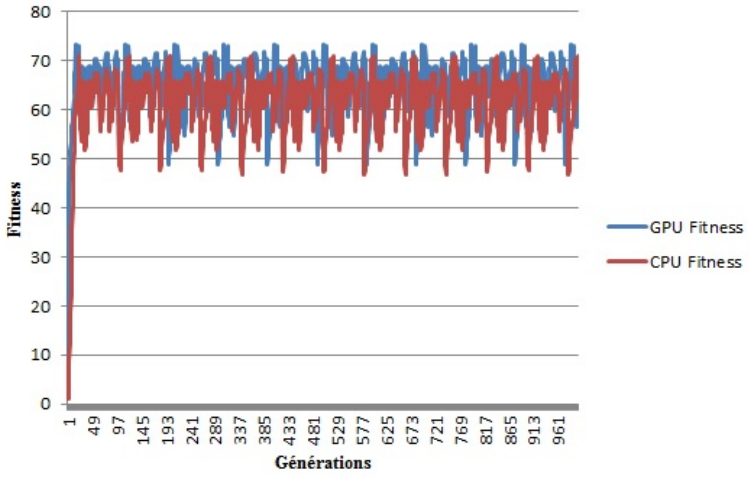


FIGURE 4.12 – Valeurs de fitness sur différentes générations de CPU et GPU.

La convergence de l'algorithme peut être étudiée en examinant les valeurs moyennes de la fonction objective de l'ensemble des générations de la population. Pour discuter la convergence de l'algorithme, nous considérons les valeurs moyennes de la fonction objective et les valeurs prises de la moyenne de la fonction objective pour les 900 séries de la SE.

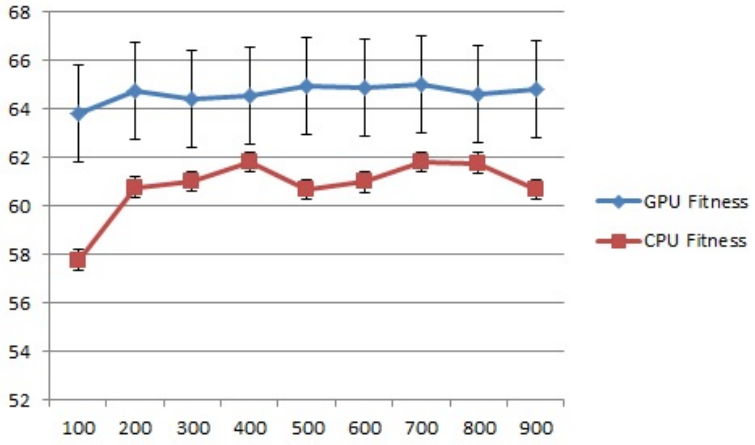


FIGURE 4.13 – Aptitude moyenne avec un intervalle de confiance de 95 % pour chaque algorithme.

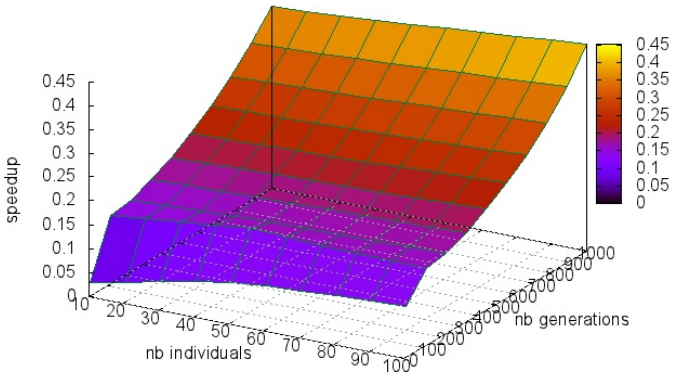


FIGURE 4.14 – Exécution globale du système accéléré.

4.10.3 Analyse de performance et de qualité des solutions, sous les limites de temps

La figure 4. 15 représente les résultats expérimentaux des implémentations CPU et GPU sur les différentes phases de la SE proposée. Ces résultats ont été atteints après 30 essais. Notons que le processus de parallélisation inter et intra chromosomes est plus efficace, puisque chaque gène est mappé à un thread dans le bloc qui maintient le chromosome. Ceci est dû au fait que la mise en œuvre basée GPU permet l'exécution de plusieurs threads. En outre, l'exécution de la plupart des processus de la SE peut réduire la fréquence de transfert des données entre le dispositif hôte et le périphérique, qui constitue le plus grand défi dans une telle application sur GPU.

Après avoir discuté les différences entre les résultats obtenus par les implémentations basées GPU et basée CPU, nous discutons dans ce paragraphe l'impact de certains paramètres sur l'ensemble des performances. Le temps de calcul et l'occupation mémoire (mémoire partagée ou registres) consacrés à la fonction objective sont plus élevés que ceux des autres kernels pour les deux tailles de chromosomes (n = 512) et (n = 1024). La raison derrière ce phénomène réside dans le fait que les éléments de calcul du Kernel proviennent de la simulation physique effectuée par l'ODE. La performance globale de notre système croît en fonction de l'augmentation du nombre d'individus dans la population.

4.10.4 Discussion de performance en termes d'efficience et d'efficacité

Pour chaque étape de la simulation, une mise en œuvre séquentielle utilisant un CPU mono-cœur ainsi qu'une mise en œuvre coopérative CPU-GPU sont considérées, en plus de l'échange des données entre le CPU et le GPU. Le délai moyen correspond à 30 essais. Les valeurs moyennes de la fonction d'évaluation ont été recueillies et le nombre d'essais réussis est également représenté. Étant donné que le temps de calcul peut concerner 1.000 générations et plus, le temps moyen attendu pour la mise en œuvre basée CPU, a été déduit sur la base de deux études d'exécution.

La génération et l'évaluation des chromosomes d'une manière parallèle sur GPU fournissent un moyen efficace pour accélérer l'évolution et le processus de recherche par rap-

port à une version basée CPU. Comme le montre les figures 4.14 et 4.16, la version basée GPU est plus rapide que celle basée CPU (c'est-à-dire. l'exécution parallèle basée GPU a permis de diviser le temps par un facteur 12 par rapport à une mise en œuvre séquentielle sur CPU). Dus aux accès élevés mal alignés à la mémoire globale, les accès non coalescés réduisent les performances de la mise en œuvre sur GPU. Pour surmonter le problème et atteindre la coalescence de la mémoire, les variables d'un même type et appartenant à des individus différents de la population sont stockées dans des tampons adjacents. Le GPU conserve l'accélération du processus évolutif hybride aussi longtemps avec l'augmentation de la taille de la population. Concernant la qualité de la solution, les résultats obtenus avec la SE proposée sont très compétitifs par rapport à ceux obtenus avec la SE séquentielle.

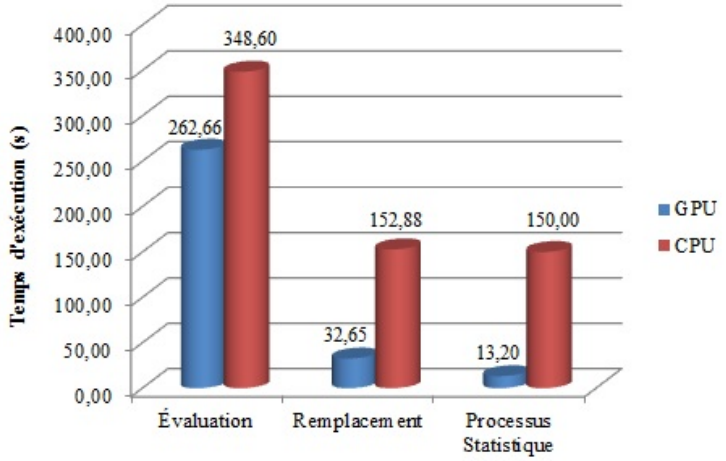


FIGURE 4.15 – Temps des phases évolutionnaire CPU vs GPU.

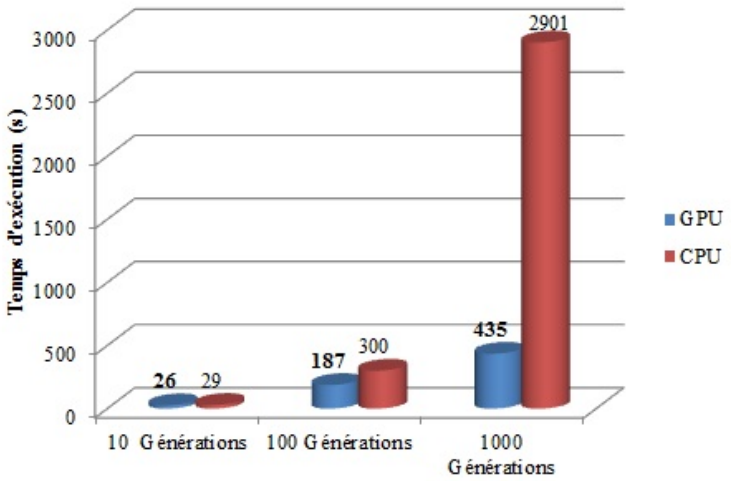


FIGURE 4.16 – Temps d'exécution du système sur CPU et GPU sur différente générations.

Les figures (4. 16) et (4. 17) représentent respectivement le temps moyen pendant quelques générations par rapport à la taille de la population et le temps Évolution CPU

vs GPU au cours des générations, selon le nombre d'individus, à travers une exécution alternée sur le CPU et sur le GPU afin de montrer plus d'informations sur l'exécution. Comme mentionnée ci-dessus, la fonction objective dans ce cas est la distance parcourue par chaque robot où tous les individus commencent et terminent en même temps (dans chaque étape de simulation). Dans chaque génération, un bon phénotype est un ensemble de paramètres qui incite l'individu à effectuer le mouvement souhaité.

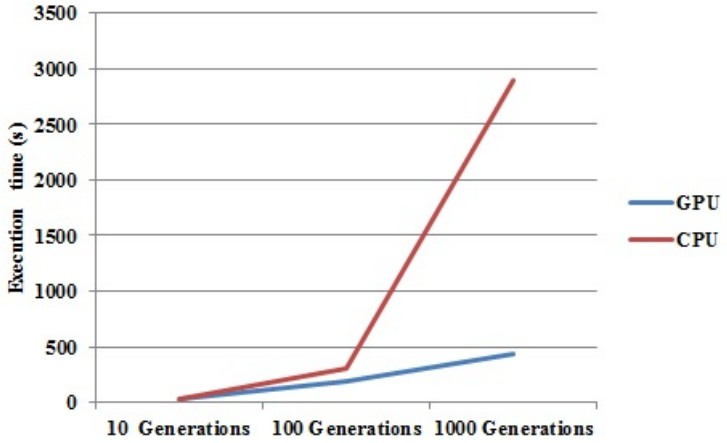


FIGURE 4.17 – Évolution de temps au cours des générations.

Nous concluons que l'utilisation des GPUs fournit un moyen efficace pour faire face à ce type d'application où plusieurs paramètres sont considérés (paramètres de simulation : par exemple, les paramètres RNN, et les paramètres SE). Par conséquent, l'implémentation de l'entrainement sur le GPU a permis d'exploiter le parallélisme dans une telle application et l'amélioration de la robustesse et de la qualité des solutions fournies. D'autre part, comme ce problème prend en compte un ensemble de paramètres, l'ordre de l'accélération obtenue est très prometteur pour aller vers d'autres méthodes d'optimisation, voire l'automatisation du réglage des paramètres de la SE sur le GPU, ou plus encore, en utilisant le multi-GPU et les nouvelles architectures telles que l'architecture Kepler.

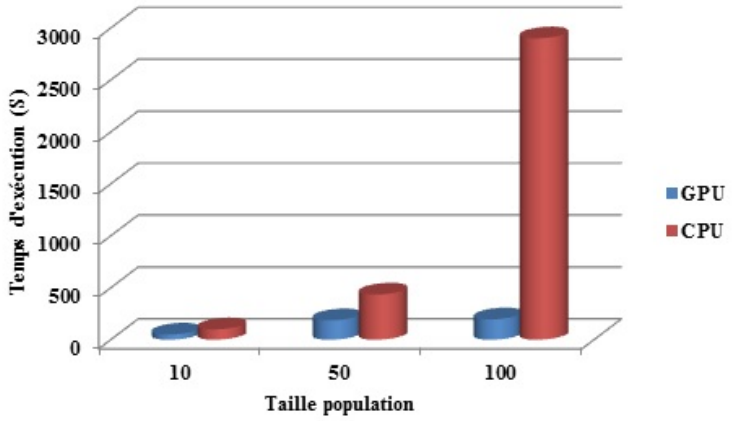


FIGURE 4.18 – Temps moyen pour chaque génération par rapport à la taille de la population.

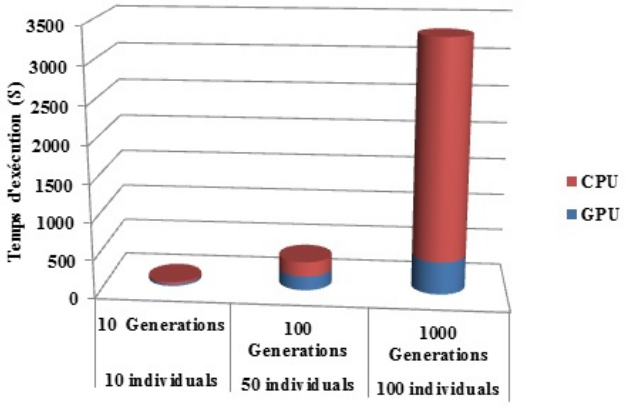


FIGURE 4.19 – Évolution du temps CPU vs GPU.

4.11 Conclusion

Le but de cette recherche consistait à étudier la faisabilité d'utilisation des capacités des technologies GPU pour simuler un modèle complexe en utilisant des approches artificielles bio-inspirées basées sur une stratégie d'évolution et un réseau de neurones récurrent. Pour atteindre cet objectif, une nouvelle approche parallèle hybride (SE-RNN basée GPU) a été proposée pour construire et simuler un modèle 3D de locomotion pour un robot humanoïde. D'une manière plus spécifique, le mouvement et l'évolution du contrôleur du robot humanoïde ont été mis en œuvre en utilisant le modèle proposé. Ce dernier est élaboré par le biais d'un simulateur physique dont le rôle est de reproduire des phénomènes physiques tels que la gravité, les collisions, etc. Nous avons ainsi mis en place notre modèle évolutif du RNN sur une plateforme à base de GPUs pour laquelle nous avons discuté diverses questions de programmation.

En outre, nous avons fourni un ensemble de lignes directrices de conception qui pourraient être utiles pour d'autres modèles, dont la façon de pallier les contraintes usuelles telles que les limitations de la mémoire disponible sous GPU et la synchronisation entre les différentes tâches exécutées sur le GPU (synchronisation entre les différents threads), ces précautions ont pour but principal l'exploitation optimale des GPUs pour des gains de performances.

En réalité, il est plus probable de supposer que l'évolution de l'entrainement et de la simulation des actions du RNN opère sur une échelle de temps plus longue que les autres opérations, qui permettent une séparation d'échelle de temps.

En définitive, cette étude a permis de réaliser avec succès les étapes définissant la mise en œuvre d'un système de contrôle optimisé qui peut exploiter au mieux la combinaison de traitements entre le CPU et le GPU, permettant au développeur de disposer d'un modèle automatique l'assistant dans le traitement de ses étapes de calcul.

Chapitre 5

Coopération CPU-GPU, distribution des tâches et gestion de la mémoire dans pEvoRNN

5.1 Introduction

La communauté de recherche du calcul généraliste sur les accélérateurs graphiques (GPGPU) a eu pour rôle de fournir les lignes directrices générales pour optimiser une application sur GPU en utilisant la plateforme Compute Unified Device Architecture (CUDA). Cependant, la compréhension des effets matériels importants et les performances qui leur sont associées dans la mise en œuvre des applications, restent encore des points à prendre en charge par les programmeurs.

Afin d'exploiter le potentiel de performances des GPUs, diverses questions doivent être abordées telles que la distribution des tâches entre le CPU et le GPU, l'utilisation efficace de la hiérarchie mémoire du GPU, la gestion judicieuse du parallélisme GPU, ainsi que l'optimisation des codes GPU, considérées comme les tâches les plus fastidieuses.

L'exploitation des architectures hétérogènes nécessite généralement une ré-implémentation des codes CPU existants. Cette ré-implémentation nécessite une grande connaissance des techniques de programmation pour ces types d'architectures particulières. Les performances des unités de traitement graphique (GPU) sont sensibles à des références de mémoire irrégulières. Une étude récente montre la nécessité d'éliminer les références irrégulières par l'exécution d'un remappage des threads/données. Cependant, la façon dont on peut déterminer efficacement le mappage optimal reste encore une question ouverte.

Concernant CUDA, l'un des paramètres ajustables [187] réside dans la forme des blocs associés aux threads (grille des blocs). Le choix de la taille et de la forme des blocs est l'une des décisions les plus importantes [186] que les utilisateurs doivent prendre lorsqu'un problème est codé afin de fonctionner d'une façon parallèle sur une architecture GPU. Un bon choix de la taille et de la forme des blocs peut affecter d'une manière significative l'utilisation des ressources des GPUs. Par conséquent, la performance utilisée dans la tâche d'implémentation globale est étroitement liée à la configuration des blocs. Actuellement, de nombreux programmeurs sélectionnent la taille et la forme des blocs de façon empirique via un processus d'essais/erreur, induisant un gaspillage énorme en termes de temps de calcul. Pour éviter une telle dépense en termes de temps de calcul ainsi qu'au le recours à

un réglage manuel, une approche intéressante consiste à automatiser la génération du code CUDA et/ou le tuning des paramètres sensibles. Les paramètres typiques des stratégies de Tuning, tels que le choix de la taille et de la forme appropriées pour les blocs de threads, la programmation d'une bonne coalescence, ou la maximisation de l'occupation, s'avèrent être interdépendants.

De plus, les choix dépendent également des détails de l'architecture sous-jacente, et le motif d'accès à la mémoire globale de la solution élaborée.

Avant de nous intéresser aux détails nécessaires pour une application optimisée sous GPU, nous allons présenter les aspects parallèles dans notre cas d'étude.

5.2 Coopération CPU-GPU

Le but de cette partie est de présenter la façon d'établir une coopération efficace entre le CPU et le GPU, ce qui nécessite le partage de tâches et l'optimisation du transfert de données entre les deux composants.

Il existe de nombreux facteurs qui doivent être considérés avant de décider si le processus doit être alloué au CPU ou au GPU. Certains de ces facteurs sont constants et d'autres peuvent compter sur les états du processus en temps réel.

Un processus correct de gestion de la distribution est important pour deux raisons :

1. Il est souhaitable que le CPU et le GPU aient des charges de processus similaires, en évitant le cas où l'un est surchargé et l'autre est entièrement libre ;
2. Il est pratique de distribuer les processus en tenant compte de l'architecture qui serait la plus efficace pour ce genre de problème.

Tout d'abord, nous allons décrire brièvement le schéma commun de parallélisation des algorithmes évolutionnaires sous GPUs. Ensuite, nous allons nous concentrer sur l'optimisation des transferts de données entre le CPU et le GPU, tant elle représente l'un des enjeux cruciaux pour atteindre les meilleures performances dans les applications GPUs. Finalement, nous allons projeter ces points critiques sur le problème étudié.

5.2.1 Répartition des tâches pour les algorithmes évolutionnaires sur GPU

5.2.1.1 Cas de parallélisation de l'évaluation

La parallélisation des algorithmes évolutionnaires suit différents modèles. L'un d'entre eux consiste à paralléliser seulement la phase d'évaluation, tant cette phase est souvent la plus consommatrice en temps de calcul dans les algorithmes évolutionnaires spécialement et les méta-heuristiques généralement (Le modèle maître/esclave). L'avantage de cette approche est qu'elle garde le même comportement comparativement à un algorithme séquentiel. À chaque itération, le maître génère l'ensemble des solutions à évaluer. Chaque travailleur (esclave) reçoit une partition de l'ensemble des solutions depuis le maître. Ces solutions sont évaluées et retournées au maître. Un challenge de ce modèle est de déterminer la granularité de chaque partition de solutions à allouer à chaque travailleur selon les délais de communication de l'architecture donnée. En termes de généricité, comme le modèle est indépendant du problème, il s'avère être générique et réutilisable.

Dans ce cas, selon le paradigme maître/esclave, l'idée est d'évaluer les solutions en parallèle sur GPU. Dans ce modèle, un Kernel est envoyé au GPU pour être exécuté par un grand nombre de threads groupés en blocs. La granularité de chaque partition est déterminée par le nombre de threads par bloc [202], [213], [2], [125], [176].

Ce genre de parallélisation est utile pour un algorithme dont le temps d'évaluation est prédominant. Ce modèle est décrit par la figure (5.1).

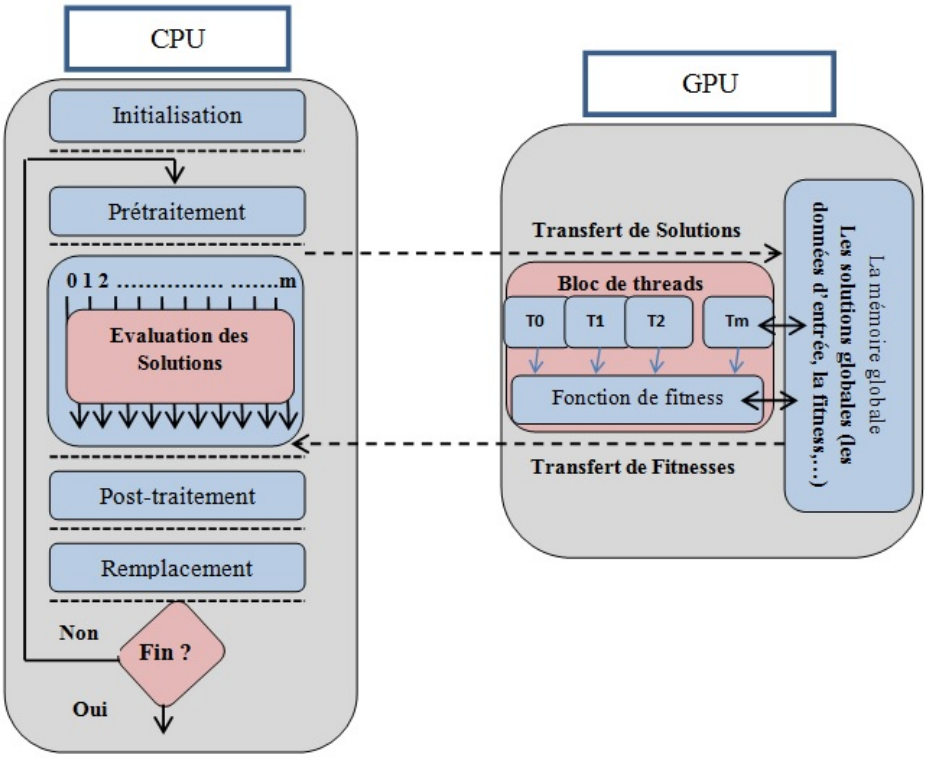


FIGURE 5.1 – Évaluation parallèle des solutions sur GPU.

5.2.1.2 Cas de l'algorithme complet

Notre étude diffère significativement des autres études du fait qu'elle représente une implémentation parallèle à base de GPU, qui de plus ne sera pas consacrée que pour l'évaluation, mais également pour toutes les autres phases, en suggérant des modifications aux opérateurs évolutionnaires existants en fonction de l'architecture du GPU (e.g. [171], [143], [2]), en fonction de la nature du problème étudié et en fonction des opérateurs évolutionnaires utilisés.

Par la suite, nous allons présenter quelques opérateurs évolutionnaires modifiés afin qu'ils soient situables pour l'architecture spécifique du GPU, suivie par les détails de la gestion du parallélisme, ainsi que la gestion de la mémoire du GPU.

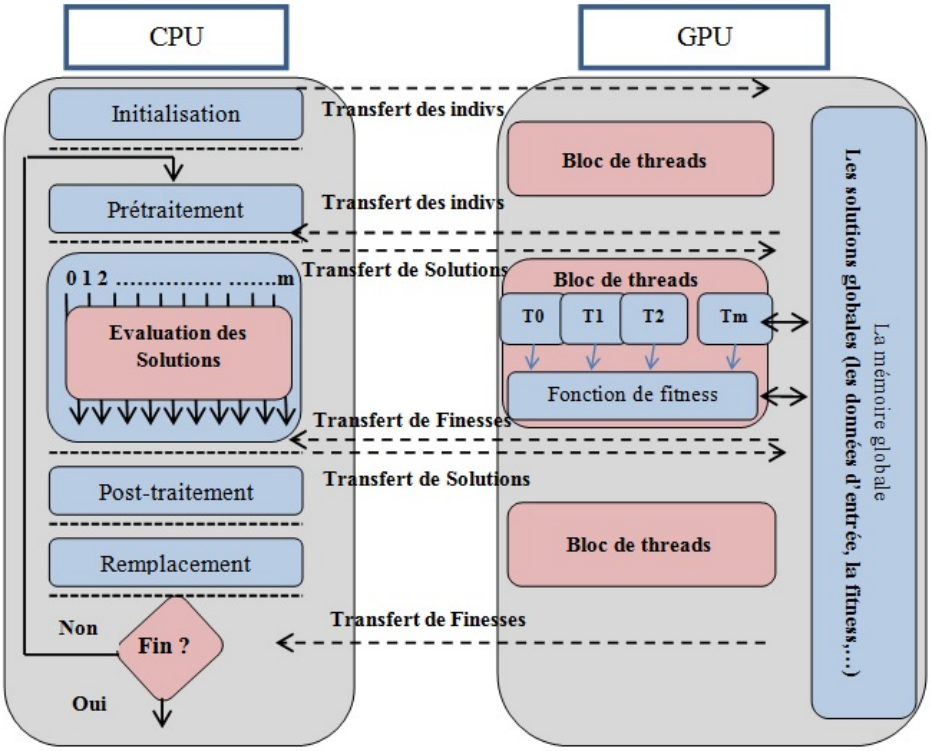


FIGURE 5.2 – Stratégie évolutionnaire complet sur GPU

Afin de converger vers une haute performance d'un GPU, la parallélisation de chaque opérateur évolutionnaire est souhaitable. Un des premiers travaux pionniers sur les opérateurs évolutionnaires a été proposé par Wong et coll. dans [202], [201] dans le but de réaliser une mutation spécifique (Cauchy mutation) sur GPU. D'une manière similaire, Shah et coll. [171]ont proposé des versions modifiées des sélections dites uniforme et roulette pipée, ainsi qu'une autre version du croisement en un seul point, convenable pour les GPUs. Dans [128], Ogier et Coll. ont présenté une étude expérimentale de différents variantes de la sélection en tournoi. Osio et coll. [143] ont présenté les versions modifiées de la sélection de tournoi, de croisement en un seul point et la mutation en un point. Dans la même optique, Jaros et Coll. [98] ont proposé aussi des opérateurs modifiés convenables à une architecture GPU.

Dans notre cas, nous avons utilisé un algorithme évolutionnaire totalement plongé dans l'accélérateur graphique, tous les détails de l'algorithme ainsi que les modifications sont dans la partie 2 du chapitre 4.

5.3 Équilibrage de la Charge

Avoir un système qui intègre un processeur hôte, un ensemble de cartes avec plusieurs cœurs sur chaque carte, ayant un ensemble de tâches à accomplir, nécessite de diviser l'ensemble des tâches en sous-ensembles assignés, aux cartes, aux MPs et finalement aux cœurs.

L'objectif principal d'un processus d'équilibrage de charge est de réduire le temps d'exécution global.

Dans un algorithme évolutionnaire synchrone, il est nécessaire d'équilibrer les tâches entre les unités de calculs indépendantes, afin de leur permettre de terminer leur calcul en même temps afin de commencer la prochaine génération. Sinon, si une unité termine avant les autres, elle deviendra inactive en les attendant, ce qui engendre donc une perte dans la puissance de calcul.

L'équilibrage de charge peut être effectué selon le niveau de la distribution des tâches. Toutefois, la distribution à un niveau déterminé peut prendre en compte la structure d'un niveau inférieur. Typiquement, la distribution MP (multiprocesseur) est également basée sur la structure des cœurs simples.

Les systèmes GPUs ont trois niveaux (*Cartes*, *MP*, *les cœurs simples*), mais l'équilibrage de charge se fait en deux étapes : d'abord inter-GPU et ensuite intra-GPU. Bien sûr, les deux derniers niveaux sont fortement corrélés.

5.3.1 Équilibrage de tâche Inter-GPUs

Dans plusieurs cartes GPU sont à utiliser, il est nécessaire d'attribuer un ensemble de tâches à chaque GPU. Cette étape correspond au premier niveau de la hiérarchie décrit ci-dessus.

Le but de l'équilibrage de la charge inter-GPU est d'affecter les processus aux cartes d'une manière équilibrée, afin de permettre le calcul sur chaque carte pour terminer dans le même temps. Malheureusement, si le type des GPUs utilisés est différent dans la même machine, il est difficile de comparer la puissance des cartes.

5.3.2 Équilibrage de tâche Intra-GPUs

Les Puces GPUs comprennent de nombreux cœurs qui sont structurés sur plusieurs couches. Chaque couche doit répondre à certaines contraintes, telles que le nombre de tâches qui peuvent être prévues, le nombre de registres disponibles dans un MPs.

Le problème réside dans le fait qu'avec un certain nombre de tâches identiques et un nombre de cœurs (SPs) regroupés en un nombre de MPs, nous devons les exploiter pour répartir les tâches selon les critères suivants.

1. Équilibrer la charge entre les différents MPs ;
2. Équilibrer la charge entre les différents cœurs d'un MP, afin de ne pas créer de divergences artificielles, en assignant des tâches à une partie d'un Warp et rien à l'autre partie ;
3. Maximiser la capacité d'ordonnancement de chaque MP, tout en respectant les contraintes sur les registres disponibles par MP et la taille du bassin d'ordonnancement.

5.4 Répartition des threads et mappage efficace de la population

Le GPU Computing est basé sur *l'hyper-threading*, et l'ordre dans lequel les threads sont exécutés est *inconnu*. Alors, un contrôle efficace des threads doit être assuré dans le but de répondre aux contraintes liées à l'utilisation de l'espace mémoire. Ceci permet

d'assurer la robustesse des solutions développées sur GPU et d'améliorer la performance globale. Une cartographie efficace doit être établie entre chaque solution candidate et le thread désigné par un identifiant unique.

Nous nous concentrons dans cette partie sur le mappage de la population sur la carte (GPU). La population doit être affectée aux multiprocesseurs (MPs) sur la carte. Cette étape est importante, car elle assure un bon équilibrage de charge sur chaque processeur ainsi qu'un ordonnancement efficace. Cependant, cette répartition doit être conjuguée avec les contraintes matérielles, qui peuvent être résumées dans le respect de l'utilisation des ressources existantes sur chaque MP et de faire en sorte de ne pas les dépasser. Certains principes liés à l'occupation de la carte et la distribution efficace et automatique des individus sur la carte pour contrôler le parallélisme des threads seront présentés. Toutefois, cette répartition doit être compatible avec les contraintes matérielles.

Un multiprocesseur (SM) peut exécuter deux demi-Warps en même temps. Les Warps des différents blocs de threads peuvent être mis en attente sur le même SM. Lorsque les threads d'un Warp émettent une demande à la mémoire globale, ces threads seront bloqués jusqu'à ce que les données arrivent de la mémoire. Durant cette période de latence élevée, d'autres Warps dans la file d'attente peuvent être planifiés et exécutés. Ainsi, il est important d'avoir suffisamment de Warps dans la file d'attente de chaque SM pour masquer les latences de la mémoire globale en les chevauchant avec le calcul, ou avec d'autres accès mémoire. La première considération pour maximiser l'occupation est de choisir une taille appropriée de bloc.

Dans l'architecture Fermi [186], le nombre de threads par bloc doit être un diviseur entier du nombre maximal de threads par SM et supérieur ou égal à 192, afin de permettre de remplir le nombre maximal de threads par SM avec pas plus de 8 blocs. En plus de l'effet lé à l'occupation, la forme choisie a également un impact significatif sur la coalescence, la partition camping, et les goulets d'étranglement de la mémoire. Une chaîne devrait demander 32 éléments contigus pour réduire la bande passante de la mémoire totale.

Certaines des stratégies communes du code de Tuning [187] interagissent fortement avec, ou sont dépendantes de, la taille de bloc choisie ainsi que de sa forme.

5.4.1 Réglage automatique de la configuration pour l'AE utilisé sur la plateforme CUDA

Il peut sembler d'abord intuitif de choisir la forme des blocs, puis d'appliquer les autres techniques d'optimisation, certains d'entre eux peuvent avoir besoin de tailles de blocs spécifiques ou de formes afin d'exploiter leur potentiel.

Il s'agit d'une question ouverte quand et où, il est possible d'isoler la configuration des blocs de threads de l'application d'autres techniques d'optimisation.

Plusieurs études ont porté sur l'autoréglage (Tuning) des paramètres sensibles dans CUDA. Dans cette partie, nous allons analyser deux travaux liés aux algorithmes évolutionnaires, celui développé par Ogier et coll. [126] et celui réalisé par Luong et coll. [190]. L'idée de l'algorithme conçu par Ogier est de maintenir l'optimisation des SMs en maximisant les threads assignés pour chaque bloc, et ce, en tenant compte à la fois les limitations des registres et de l'ordonnancement. Alors que Luong a proposé une heuristique dynamique pour l'autoréglage des paramètres de l'algorithme LSM (*Local Search Metaheuristique*). L'idée derrière cette approche est d'envoyer les threads par «vagues (*waves*)» au Kernels du GPU pour effectuer le réglage des paramètres pendant les premières ité-

rations de l'algorithme LSM. De ce fait, la mesure du temps pour chaque configuration choisie selon un certain nombre d'essais fournira les meilleurs paramètres de configuration.

5.4.2 Contribution à la détermination de l'aménagement optimal pour l'activité courante

Parmi les méthodes prometteuses permettant le réglage automatique des paramètres les heuristiques qui sont a priori des approches basées sur l'énumération de toutes les différentes valeurs des paramètres sensitifs (threads par bloc et nombre total des threads). Cependant, de telles approches sont excessivement consommatrices en termes de temps de calcul et ne peuvent pas bien être adaptées pour le type de problème étudié dans cette thèse en raison de leur nature.

Pour traiter ce problème, L'algorithme (3) décrit le procédé utilisé pour distribuer la population des robots dans PEvoRNN sur chaque SM de la carte et en même temps trouver les paramètres sensibles les plus appropriés à chaque étape de la simulation.

L'idée principale de cet algorithme est d'effectuer le réglage des paramètres pendant les premières itérations des Kernel GPU de la partie évolutionnaire (y incluent les parties liées à la simulation et la génération des poids des RNNs).

De ce fait, la mesure du temps pour chaque configuration sélectionnée en fonction d'un certain nombre d'essais (lignes 5- 16) donnera les meilleurs paramètres de configuration. En ce qui concerne le nombre de threads par bloc, comme cité ci-dessus, il est fixé à un multiple de la taille du warp (voir la ligne 21). Le nombre total d'unités d'exécution de départ est défini comme le plus proche de la puissance de deux. Dans certains cas, quand un bloc de thread alloue plus de ressources (registres par exemple) que ceux disponibles sur un multiprocesseur, l'exécution du Kernel échoue, car trop de threads sont demandés. Par conséquent, un mécanisme de tolérance aux pannes est prévu pour détecter une telle situation (à partir des lignes 11 à 14). Dans ce cas, l'algorithme se termine et renvoie les meilleurs paramètres de configuration précédemment trouvés.

Les paramètres à déterminer sont le nombre d'essais par configuration, les informations de l'architecture sous-jacente et le temps imparti à chaque essai. Plus la valeur du nombre d'essais est élevée, plus le réglage final sera précis au détriment d'un temps de calcul supplémentaire.

Algorithme 3 Aménagement optimale des données

Entrées: Nb_essaies, GPU_infos, Temps_de_essaie
Sorties: meilleur_Nb_threads, meilleur_Nb_threads_block

1: $Nb\_thread = Nb\_gene\_par\_chromosome$
2: $Dim\_grid = Nb\_MP$
3: **tantque** $(Nb\_thread < popsize)$ **faire**
4: $Nb\_threads\_block = 32$
5: **tantque** $(Nb\_threads\_block <= 1024)$ **faire**
6: **tantque** $Temps\_courant <= Temps\_de\_essaie$ **faire**
7: Initialisation des paramètres SE sur le côté GPU ;
8: Génération de la population initiale sur le côté du GPU. (Initialisation des poids du RNN) ;
9: Lancer l'évaluation sur le GPU ;
10: Exit ;
11: **si** échec kernel **alors**
12: Récupérer (meilleur_Nb_threads) ;
13: Récupérer (meilleur_Nb_threads_block) ;
14: **fin si**
15: Le reste du processus d'évolution sur le côté GPU ; (Manipulation évolutionnaire, replacement, les calculs statistiques).
16: **fin tantque**
17: **si** meilleur_temps **alors**
18: $meilleur\_Nb\_threads \leftarrow Nb\_threads$;
19: $meilleur\_Nb\_threads\_block \leftarrow Nb\_threads\_block$
20: **fin si**
21: $Nb\_threads\_block \leftarrow Nb\_threads + 32$
22: $Dim\_block \leftarrow Nb\_threads\_block$
23: **si** $Dim\_block > Nb\_threads$ **alors**
24: $Dim\_grid \leftarrow Dim\_grid + Nb\_MP$
25: **fin si**
26: **fin tantque**
27: $Nb\_threads \leftarrow Nb\_threads * 2$
28: **fin tantque**

5.5 Gestion de la Mémoire

Pour résoudre le problème du temps de transfert de données, nous proposons un sous-modèle qui joue le rôle d'un gestionnaire d'accès à la mémoire et qui orchestre les transferts des données entre le processeur et la mémoire du GPU.

Avant de rentrer dans les détails, nous proposons de présenter les règles générales d'utilisation de la hiérarchie mémoire.

5.5.1 Concepts communs de gestion de la mémoire

5.5.1.1 Enjeux de coalescence de mémoire

Dans le modèle d'exécution des GPUs, chaque bloc de threads est divisé en groupes de Warps. À chaque cycle d'horloge, chaque processeur du multiprocesseur sélectionne un demi-Warp (16 threads) qui soit prêt à exécuter la même instruction sur des données différentes. La mémoire globale est conceptuellement divisée en une séquence de segments de 128 octets. Le nombre de transactions de mémoire exécutées pour un demi-Warp sera le nombre de segments ayant les mêmes adresses utilisées par cette demi-Warp.

Pour plus d'efficacité, les accès globaux à la mémoire doivent être coalescés, ce qui signifie qu'une demande de mémoire effectuée par des threads consécutifs dans un demi-Warp est strictement associée à un segment. L'exigence est que les threads du même Warp doivent lire de la mémoire globale dans un modèle ordonné. Si les accès mémoires des threads d'un seul demi-Warp constituent une plage contiguë des adresses, les accès seront fusionnés en une transaction de mémoire unique. Autrement, l'accès à des endroits dispersés se traduit par une divergence de mémoire et nécessite l'intervention du processeur pour produire une transaction mémoire par thread. La pénalité associée à la performance pour les accès de mémoire non coalescés varie en fonction de la taille de la structure de données.

En effet, en ce qui concerne les structures de problèmes d'optimisation, la coalescence est parfois difficilement réalisable puisque les accès à la mémoire globale ont un modèle non structuré dépendant des données. Par conséquent, les accès non coalescés à la mémoire impliquent de nombreuses opérations de mémoire qui conduisent à une diminution significative des performances pour les algorithmes évolutionnaires.

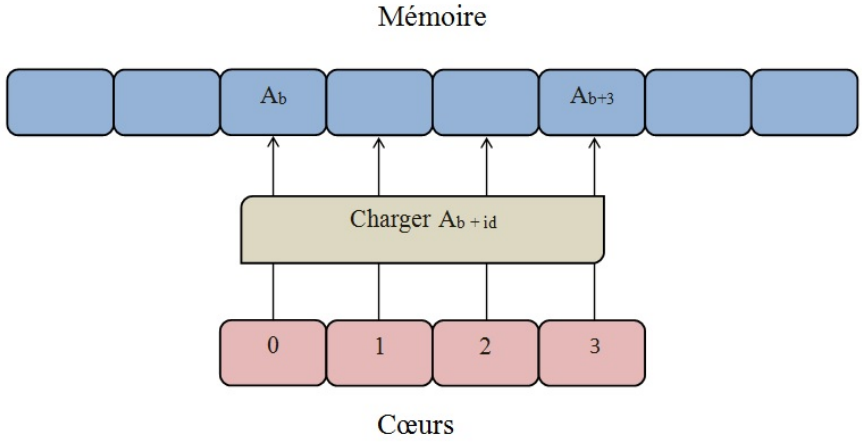


FIGURE 5.3 – Des coeurs SIMD réalisant le chargement à partir d'une adresse de base A_b et le coeur *id*.

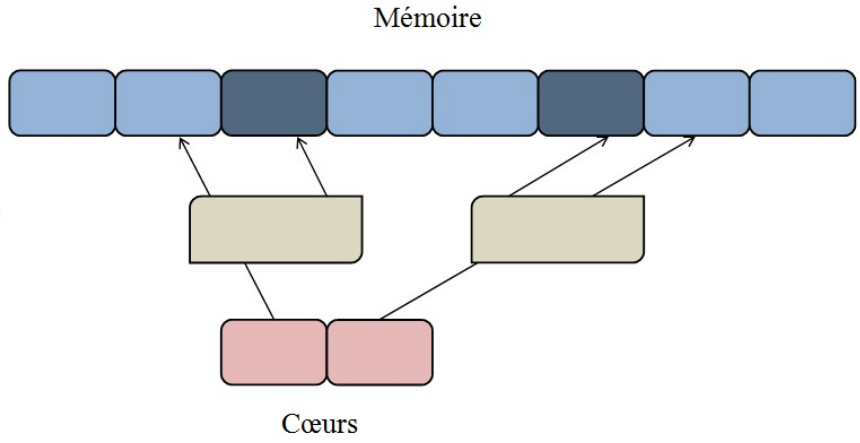


FIGURE 5.4 – Accès mémoire non-coalescé.

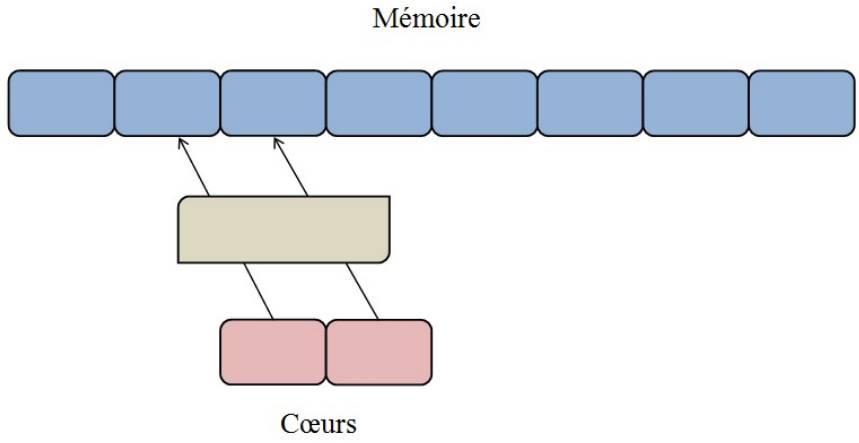


FIGURE 5.5 – Accès mémoire coalescé.

5.5.2 Transformation vers la coalescence

Néanmoins, pour certaines structures d'optimisation qui sont particulières à un thread donné, la coalescence sur la mémoire globale peut être effectuée. Cela est généralement le cas pour l'organisation des données d'une population dans les algorithmes évolutionnaires ; où de grandes structures locales sont utilisées pour la fonction d'évaluation. Cependant, afin de déterminer si les accès vers la mémoire l'off-chip peuvent être coalescés, le calcul des adresses de chaque accès mémoire dans la fonction de Kernel pour différents threads doit être effectué.

Comme les tableaux sont les structures de données les plus communes dans le traitement scientifique, quatre types d'indices pour les tableaux et des transformations affines de ces indices sont considérées.

1. L'indice constant : La valeur constante est utilisée dans un index de tableau, par exemple, la constante entière 5 dans un [idy][i + 5] ;
2. Indice prédéfini : les numéros prédéfinis, tels que les ids absolus de thread, idx, Idy, et des identifiants relatifs, tidx (c.-à-d., threadIdx.x), tidy (c.-à-d., threadIdx.y) sont utilisés comme un indice de tableau. Par exemple, dans un Idy [Idy][i + 5].
3. Indice de boucle : une variable d'itération de boucle est utilisée comme un indice de tableau, par exemple, i en b [i][idx] ;
4. Indice non résolus : Par exemple, un accès indirect a[x], où x est une valeur chargée de la mémoire. Dans ce cas-là, la vérification de la coalescence est ignorée.

Après avoir déterminé les types d'indices des tableaux dans la fonction du Kernel, et pour chaque instruction d'accès mémoire, les adresses des 16 threads consécutifs dans le même Warp sont calculées pour voir si elles peuvent être coalescées. Pour répondre à l'exigence de coalescence, l'adresse de base doit être un multiple de 64 et les Offests doivent être de 1 à 15 mots. Après l'analyse de chaque accès au tableau dans le code du Kernel, les accès non coalescés à la mémoire globale sont convertis en des accès coalescés à travers la mémoire partagée.

La figure (5.6) présente un exemple d'une transformation de coalescence pour les structures locales. Comme illustré en haut de la figure, une mauvaise approche naturelle pour organiser les éléments est d'aligner les différentes structures une après l'autre. Ainsi, chaque thread peut avoir accès à des éléments de sa propre structure avec un modèle logique AdresseDeBase x id + Offset. Par exemple, sur la figure, chaque thread a accès au deuxième élément de la structure avec AdresseDeBase =3 et Offset = 2.

Même si cette façon d'organiser les éléments sur la mémoire globale est naturelle, il est clair qu'elle n'est pas efficace. En effet, pour obtenir une meilleure performance de la mémoire globale, les accès mémoire doivent constituer une plage contiguë d'adresses pour être coalescés. Ceci est illustré en bas de la figure, où les éléments des structures sont dispatchés de telle façon que chaque accès de threads sera coalescé en une seule transaction de mémoire avec le modèle AdresseDeBase2 x id + Offset.

5.5.3 Notre proposition

Pour résoudre le problème du temps de transfert de données, nous proposons un sous module qui joue le rôle d'un gestionnaire d'accès à la mémoire et qui orchestre les transferts des données entre la mémoire du CPU et la mémoire du GPU.

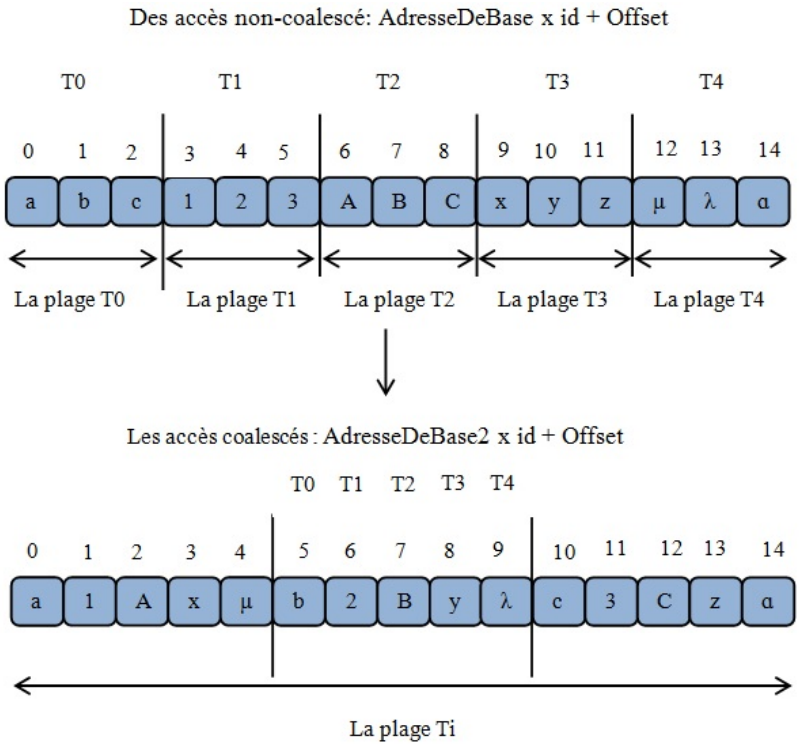


FIGURE 5.6 – Largeur de bus de la mémoire et l'importance de la vitesse de la mémoire.

Algorithme 4 Optimisation de la gestion de la mémoire.

ENTRÉES: Bloc_données [N] ,CPU_addr, Taille_Données, temps_dernier-accése, mis_à_jour, GPU_addr, reside = on_cpu, on_gpu

```
1: Allouer/Transférer les données (en utilisant CudaMalloc, CudaMemcpy) ;
2: Assurer la cohérence des données ;
3: si blockID ∈ DataBlock[N] alors
4:     reside = on_gpu ;
5:     Mettre_à_jour_les_paramètres_de_CPU (void *CPU_addr, size_t size) ;
6: fin si
7: si execution_CPU_kernel_call_time_termine alors
8:     Marquer_Sorties_CPU_memorie (void *cpu_addr, size_t size) ;
9:     Mettre_à_jour_les_paramètres_de_CPU (void *cpu_addr, size_t size) ;
10: fin si
11: GPU_addr = acquérir (void *CPU_addr, size_t size, bool update) ;
12: free (void *cpu_addr, size_t size) ;
```

Pour ce faire, ce module gère un mappage entre les régions de la mémoire du CPU et du GPU. Notamment, la mémoire globale du GPU est considérée comme un ensemble de blocs de données non chevauchants, chacun d'entre eux correspondant à un bloc de données du CPU. Le mappage est stocké dans une liste de structure nommée Bloc_données et qui représente les blocs de données. Chaque bloc de données dispose d'une adresse CPU cpu_addr, une adresse GPU gpu_addr, d'un état de synchronisation (sync) et d'un horodatage indiquant le dernier accès au bloc. Le statut de la synchronisation indique si

le contenu des blocs CPU et GPU est synchronisé ou si la copie mise à jour des données réside dans la mémoire CPU/GPU (on_cpu/on_gpu). Notons que, puisque l'application tourne sur le CPU et le Runtime fonctionne à la granularité de l'appel de fonction, ce module n'alloue de la mémoire GPU (et ne met à jour la liste des blocs de données) que lorsque le Runtime appelle la mise en œuvre du GPU pour une fonction interceptée.

La partie majeure de la manipulation du mappage de la mémoire CPU/GPU est réalisée dans les lignes de code (1 et 2), qui sont invoquées par l'exécution de tous les paramètres d'un appel du Kernel GPU. Étant donné un bloc de mémoire CPU, la ligne (1) retourne un pointeur vers le bloc de mémoire GPU correspondant. Si le paramètre mis_à_jour est réglé sur vrai, alors le contenu du bloc de mémoire GPU doit être mis à jour. L'exécution d'un Kernel GPU affecte uniquement la mémoire du GPU. Le Runtime n'impose aucun transfert de mémoire GPU/CPU après l'invocation d'un Kernel GPU. La cohérence des données est assurée par l'ensemble des invocations mis sur les paramètres de sortie du Kernel appelé. Étant donné une adresse du CPU, l'attribut de synchronisation du bloc de données correspondant sera mis à on_gpu. Lorsqu'un Kernel est invoqué sur CPU, le Runtime doit assurer que la mémoire du CPU a une copie à jour de tous les paramètres d'entrée.

Après l'exécution d'un appel Kernel CPU, les paramètres de sortie doivent être marqués comme résidants dans la mémoire du CPU. Ceci définit l'attribut de synchronisation des blocs de données contenant la plage d'adresses donné à ON_CPU. Enfin, ce module fournit une primitive pour libérer l'espace mémoire (ligne 12). La primitive free élimine, de la liste du bloc de données, toutes les entrées contenant des adresses de la plage d'adresses donnée, et libère la mémoire GPU correspondante. Cette fonction est invoquée dans deux cas : (1) lorsque l'application libère les données, et (2) lorsque la mémoire est pleine. Dans ce dernier cas, le Runtime utilise le champ Horodatage dans la structure Bloc_données afin de déterminer les blocs moins récemment utilisés. Les blocs déterminés comme les moins utilisés seront copiés sur CPU avant la désallocation GPU.

5.6 Conclusion

Dans ce chapitre, nous avons traité les trois challenges importants et nécessaires dans une application GPU dans un cadre général avec comme objectif de les projeter sur notre cas d'étude. Premièrement, nous avons proposé une coopération efficace entre le CPU et le GPU pour notre système pEvoRNN. Deuxièmement, nous avons traité avec un contrôle efficace du parallélisme. Pour cette fin, une compréhension claire du modèle d'exécution des threads permet de concevoir de nouveaux algorithmes, et d'améliorer la performance des algorithmes évolutionnaires sur GPU. Dans ce cadre, nous avons proposé un aménagement efficace des parties de notre simulation sur le GPU. Il est à noter qu'en dernier, l'accent a été mis sur la gestion de la mémoire pour le processus de simulation dans notre système pEvoRNN. Ce point est essentiel vu que la compréhension de l'organisation hiérarchique des différentes mémoires disponibles sur les architectures GPU est la question cruciale à laquelle il est impératif de répondre pour réussir à concevoir des algorithmes parallèles efficaces. Dans ce but, nous avons étudié la refonte complète du modèle pour le processus de simulation.

Chapitre 6

Étude Comparative

6.1 Introduction

L'étude comparative que nous avons élaborée permet de tirer des conclusions sur les principaux points étudiés dans notre modèle et qui sont en commun avec les travaux existants dans la littérature et s'intéressant au même domaine et à la même problématique. Cette étude nous permet une auto-évaluation de notre travail en présentant les différents points positifs et négatifs des différentes contributions effectives. De plus, une étude comparative sera réalisée entre, d'une part les propositions que nous avons élaborées et d'autre part les modèles de la littérature déjà présentés dans la partie correspondante de l'état de l'art.

Cette comparaison sera centrée autour de la problématique présentée au niveau de la deuxiéme partie du quatriéme chapitre qui a eu pour objet l'étude et la mise en œuvre d'une nouvelle implémentation sur GPU d'un problème lié à la robotique évolutionnaire, en répondant aux défis et aux questions liées à l'utilisation d'une telle architecture dans ce domaine. Les critères de comparaison sont définis comme suit :

1. Le type du contrôleur évolutionnaire simulé ;
2. La coopération entre le CPU et le GPU pour les tâches de simulation effectuées ;
3. L'utilisation de la hiérarchie mémoire ;
4. La gestion du parallélisme : le mappage des tâches de simulation sur la carte graphique.

6.2 Notre approche évolutive à l'égard d'autres approches : une brève comparaison

Le but de l'étude présentée dans le chapitre 4 de la présente thèse consiste à évaluer l'influence de l'utilisation des accélérateurs graphiques dans le processus d'évolution des comportements des robots évolutionnaires.

Le problème lié au développement, par évolution, de comportements de robots évolutionnaires et cognitifs est d'une complexité élevée liée à l'énorme quantité de puissance de calcul nécessaire telle que vue dans les travaux de la littérature [149], [69]. Ceci nous permet de prendre conscience et conforter la nécessité d'accélérer le processus d'évolution

à réaliser. En définitive, nous allons considérer l'étude de cas sur notre robot humanoïde, et ce, dans le but d'opérer l'accélération de l'évolution du contrôleur du robot dans l'environnement de simulation.

Notre travail se recoupe avec les travaux récents existants dans la littérature, dont nous avons présenté certains au niveau du troisième et quatrième chapitre de cette thèse (voir les sections des réseaux de neurones et les algorithmes évolutionnaires sous GPUs). Par ailleurs, les objectifs visés par notre travail sont différents des autres travaux, en plus de certains points forts qui différencient notre modèle par rapport aux autres.

Afin de mieux situer et comparer notre approche basée GPU pour l'accélération de l'évolution du contrôleur du robot humanoïde, par rapport à l'ensemble des travaux existant dans la littérature, il est indispensable de s'intéresser aux travaux qui traitent la même problématique, qui de plus traite la même technique accélérée. Cependant, notre travail est parmi les premiers travaux qui ont abordé l'accélération d'un processus évolutionnaire dédié à un problème de robotique évolutionnaire, sur les accélérateurs graphiques. Ceci nous conduit à faire le constat suivant : la comparaison qui pourrait être faite, serait incohérente, puisque dans les travaux existants, la technique généralement accélérée est celle d'un type bien précis de réseaux de neurones tels que les réseaux de neurones dits "Spiking Neural Networks", comme ceux utilisés par [69], et [16].

Pour cela, nous proposons d'établir d'abord une comparaison de notre approche évolutionnaire (chapitre 4-section 4.7-) par rapport aux approches présentées dans le chapitre 3 et 4 (chapitre 3- section 3.8, chapitre 4- section 4.5-), en étudiant les éléments critiques d'une application supportée par une architecture basée GPU : l'organisation des données au niveau de la hiérarchie mémoire, le nombre de threads attribués pour chaque individu dans la population de l'évolution (solution), et la hiérarchie de mémoire utilisée au sein de l'application, comme repris dans le tableau (6.1).

Fondamentalement, la plupart des approches de la littérature sont basées soit sur l'évaluation parallèle des solutions sur GPU ou l'exécution d'algorithmes indépendants/coopératifs simultanée. Ces approches peuvent aussi être classées en des approches entièrement parallélisées ou des approches partiellement parallélisées, où quelques-uns de ces travaux ont exploité le parallélisme entre les gènes conduisant à deux types d'exploitation du parallélisme : parallélisme inter-chromosome et parallélisme intra-chromosome.

Concernant la représentation des données (c.-à-d. la représentation de la structure), des tableaux à 1 et 2 dimensions servant à recueillir les données sur les objets sont généralement utilisés pour stocker les individus de la population. La majorité des approches suggèrent qu'un individu soit composé d'un génome auquel on associe d'autres champs tels que la valeur de la fitness et une variable booléenne qui indique si l'individu a été déjà évalué ou non. Cependant, nous avons préconisé dans notre approche évolutive que les attributs de chaque robot (individuel) soient rassemblés dans une même structure de données, y compris les différents paramètres des simulations physiques ainsi que les paramètres évolutionnaires (par exemple : la masse, les positions du contact avec le sol, les dimensions des primitives utilisées, la valeur de la fitness, les informations statistiques [c.-à-d. la fitness, les min/max de la fitness], et le nombre de neurones et de couches). Notre implémentation regroupe les mêmes individus dans des tampons contigus visant à transférer toutes les informations nécessaires en un seul transfert en mode DMA (Direct Memory Access). De plus, la longueur des chromosomes est trop grande, où les gènes ne représentent pas seulement les valeurs du RNA utilisé, mais également les poids qui leur sont associés. Les paramètres de la stratégie évolutionnaire sont stockés dans la mémoire

constante du GPU, alors que la mémoire partagée est utilisée pour faciliter l'accès aux données réutilisées dans l'ensemble du processus d'évolution.

Concernant le nombre de threads affectés, la plupart des implémentations que nous avons étudiées associent un thread à un individu (solution) pour le modèle maître/esclave. En outre, un bloc de threads est affecté à une sous-population pour le modèle en ilot ou les algorithmes de coopération. Toutefois, en essayant de pousser nos investigations, nous sommes arrivés au constat qu'il existe peu d'investigations qui répondent à la gestion efficace des threads avec les contraintes d'utilisation de la mémoire, en particulier lorsqu'ils traitent un grand nombre de solutions ou de grandes instances de problèmes. Notons, également, que ces solutions sont généralement spécifiques à un problème particulier ou à des problèmes de la même famille. Dans notre travail, nous proposons d'attribuer un bloc de threads à chaque individu avec une gestion efficace de la distribution des threads sur la carte.

Concernant la gestion de la mémoire de données, il y a peu d'efforts explicites qui gèrent les structures d'optimisation avec les différentes mémoires disponibles. Ceux-ci sont d'ailleurs strictement dépendants du problème d'optimisation étudiée. Pour ces raisons, nous avons proposé notre propre système de gestion de mémoire en tenant compte de toutes les parties du processus de simulation.

6.3 Bilan

Les différents points sur lesquels nous avons réalisé notre étude comparative ont mis en avant les points forts et les capacités offertes par notre modèle accéléré de simulation dans un environnement artificiel, et ce, du fait qu'il exploite les points suivants : la prise en compte d'un simulateur physique dont quelques parties sont accélérées, l'exploitation du parallélisme existant et qui est utilisé dans la carte utilisé pour le processus évolutionnaire pour accomplir une simulation accélérée.

En conclusion, nous présentons à titre de récapitulatif une comparaison entre nos objectifs initiaux présentés dans cette thèse, ceux effectivement réalisés et ceux visés dans le futur immédiat en termes de perspectives et ce relativement aux comportements parallèles évolués réalisés, et des résultats obtenus.

En résumé, les points forts de notre modèle de simulation accéléré relèvent du constat qu'ils traitent plusieurs des facteurs qui entrent dans la mesure d'une simulation évolutive accélérée qui sont :

1. L'utilisation parallèle de quelques parties du simulateur physique ;
2. L'émergence du comportement de populations dans un environnement hautement parallèle ;
3. Le partage des tâches de la simulation entre le CPU et le GPU ;
4. Le mappage entre la population des robots et les éléments élémentaires (threads) dans le processus de la simulation ;
5. La gestion efficace des transferts de données entre les parties de la simulation.

Classification des AEs basés GPU : à base de gène et à base de chromosome

	Étude	Organisation des données	Approches à base d'un ou plusieurs threads	Gestion de la mémoire de données
À base de chromosome	Arora et coll. (2010)	Des entiers 1D et des tableaux flottants.	Différents threads d'un bloc pour différents individus de la population.	Les variables d'un même type et qui appartiennt à des individus différents sont stockées de manière adjacente.
	Ogier et coll. (2012)	Collection d'objets représentant les individus.	Un thread par individu pour la phase de l'évaluation.	Les individus sont regroupés dans des tampons contigus.
	Pospical et coll. (2010)	Des tableaux séparés 1-D, où chaque tableau représente la sous-population.	Chaque individu est contrôlé par un seul thread CUDA.	Les populations de l'îlot local sont stockées dans la mémoire partagée de la puce GPU.
	Zhu (2009)	Des tablaux 1-D séparés, où chaque tableau représente la sous-population.	Chaque individu est contrôlé par un seul thread CUDA.	Les mémoires : globales, partagée, et de texture.
À base de gène	Kromer et coll. (2011)	Toute la population peut être considérée comme une matrice réelle.	Chaque vecteur (solution) est traité par un bloc de threads.	Toute la population réside dans la mémoire principale.
	Osio et coll. (2011)	Collection d'objets représentant la population.	Chaque individu est traité par une SM, et chaque gène par un SP.	Toute la population réside dans la mémoire principale.
	Jaros et coll. (2012)	C structure constituée de deux tableaux à une dimension.	Chaque vecteur (solution) est traité par un bloc de threads.	Les chromosomes sont stockés dans le cache L1 du GPU.
	Shah et coll. (2010)	Toute la population peut être considérée comme une matrice réelle.	Chaque vecteur (solution) est traité par un bloc de threads.	La matrice de la population réside dans la mémoire principale du processeur graphique.
	Benalia et coll. (2015)	**Collection de différents objets représentant la population.**	**Plusieurs threads CUDA travaillent sur un chromosome pour évaluer sa fitness.**	**Les mémoires : global, partagé, et constante.**

TABLE 6.1 – Table comparative selon certains critères d'une implémentation sur les GPUs.

Auteurs	**Le robot simulé**	**Fonction de fitness**	**Composants du contrôleur**	**L'environnement**	**La gestion de mémoire et le contrôle du parallélisme**
Peniak et coll. (2011)	Robot humanoïde	Simple	Réseau de neurones	Physique 3D	/
Nalda et coll. (2011)	Robot humanoïde	simple	Réseau de neurones	Physique 3D	/
Benalia et coll. (2015)	**Robot humanoïde**	**simple**	**Combinaison d'un réseau de neurones avec un une stratégie évolutionnaire**	**Physique 3D dont les parties mathématiques et physiques sont plongées dans le GPU.**	**Proposition d'un mécanisme de transfert de données plus un autre mécanisme qui permet de trouver les bons paramètres des configurations des Kernels pour l'évolution dans la simulation.**

TABLE 6.2 – Table comparative entre les travaux selon certains critères de simulation

Chapitre 7

Conclusion Générale

7.1 Récapitulatif des Contributions

Les accélérateurs GPUs sont aujourd'hui omniprésents : dans nos ordinateurs portables, dans les stations de travail graphiques, dans les clusters hybrides, dans les grilles de calcul et dans le Cloud Computing. Cette disponibilité va augmenter d'une manière exponentielle avec l'arrivée de machines exaflopiques annoncées pour 2018-2020 .

Le défi est, et sera dans un futur proche, de répondre à la question : comment concevoir et mettre en œuvre des algorithmes efficaces pour les environnements informatiques GPU améliorés ?

Dans cette thèse, l'accent a été mis sur l'accélération des techniques d'évolution artificielle développées dans le domaine de la vie artificielle. En effet, nous avons revu la conception et la mise en œuvre des algorithmes évolutionnaires pour résoudre un problème lié à la robotique évolutionnaire sur des plateformes de calcul hétérogènes. Sans perte de généralité, le problème étudié est considéré comme une étude de cas.

Pour y parvenir, nous avons d'abord étudié l'utilisation d'un seul processeur couplé à un dispositif GPU, en nous intéressant à différentes questions liées aux caractéristiques des GPUs, à la nature des algorithmes évolutionnaires, et au contexte du problème dans la vie artificielle : la divergence des threads, la gestion de la mémoire du GPU, l'adaptation du dimensionnement des données transférées et l'optimisation du transfert de données entre le CPU et le GPU.

Au cours de cette thèse, nous nous sommes intéressés à l'utilisation générique des cartes graphiques (GPUs) en tant que machine de calcul parallèle pour une certaine gamme de problèmes dans la vie artificielle, en expliquant leurs structures et les caractéristiques de leur modèle de programmation.

Les algorithmes évolutionnaires parallèles permettent de fournir l'efficacité et la robustesse dans plusieurs domaines tels que celui lié à l'évolution dans la vie artificielle. Leur exploitation pour résoudre les problèmes du monde réel est possible mais nécessite l'utilisation d'une puissance de calcul importante. Le calcul haute performance basé sur l'utilisation des GPU s'est révélé être un excellent moyen de fournir une telle puissance de calcul. Cependant, l'exploitation des modèles parallèles n'est pas triviale et de nombreuses issues liées à la gestion de la hiérarchie de mémoire de l'architecture GPU doivent être ciblées et sérieusement résolues.

Cette thèse a permis, entre autres, de présenter un état de l'art sur les algorithmes

évolutionnaires (chapitre 2), et les algorithmes évolutionnaires parallèles exploitant les accélérateurs graphiques (chapitre 3). Une analyse sur le parallélisme existant dans notre problème a été menée au niveau du chapitre 4, suivie de notre Framework pEvoRNN, qui a exploré l'application du GPU Computing dans le domaine de la robotique évolutionnaire avec une mise en évidence des travaux les plus importants dans des sous-domaines divers qui utilisaient les GPUs afin d'accélérer la convergence des réseaux de neurones et des algorithmes évolutionnaires.

Une autre caractéristique intéressante des algorithmes évolutionnaires parallèles réside dans le fait qu'ils permettent d'utiliser efficacement les machines parallèles sur des problèmes inverses séquentiels et que plutôt que de devoir paralléliser l'évaluation d'un individu, ils exécutent des milliers d'évaluations, à l'origine séquentielles, en parallèle.

Cependant, l'utilisation efficace du paradigme GPGPU reste un défi, puisque le portage direct d'un algorithme sur ces architectures est conditionné par de nombreuses contraintes, car ces dispositifs informatiques sont très différents d'un processeur standard. Les techniques de programmation sont différentes et sont peu susceptibles d'être à la portée des chercheurs dans les sciences appliquées tels que les chimistes ou les physiciens.

En effet, la complexité de programmation rend très peu probable leur faculté d'être en mesure de reproduire les algorithmes décrits dans de nombreux articles publiés sur GPGPU.

Dans la majorité des implémentations, les auteurs ont comparé une mise en œuvre optimisée avec une mise en œuvre mono-thread non optimisée. Ceci dit, la plupart des implémentations ont atteint des taux d'accélérations considérables qui ne devraient en aucun cas être une surprise puisque de nombreux algorithmes dans la vie artificielle ont tendance à être intrinsèquement parallèles.

Le chapitre 5 a pris en considération les détails de la gestion de la mémoire ainsi que le mappage du problème sur le GPU. Le chapitre 6 et 7 représentent respectivement une étude comparative, et une conclusion de ce manuscrit en soulignant l'apport de la thèse, les limites de l'approche présentée ainsi que les perspectives entrevues. Une étude concernant les outils de calcul de performance et les modèles analytiques existants ont été présentés en annexe.

7.2 Discussions et Perspectives

Pour des orientations futures, les approches développées à l'occasion de cette thèse pourraient être intégrées à un Toolkit plus large, dont le but serait de gérer les techniques évolutionnistes utilisées dans la robotique évolutionnaire sous architectures GPU (d'autres types de réseaux de neurones avec d'autres types des techniques évolutionnistes).

D'autres perspectives sont liées aux évolutions récentes dans le contexte du calcul haute performance (HPC). En effet, le HPC se généralise au calcul basé Cloud, le logiciel prend conscience du facteur énergie, et les accélérateurs GPUs sont de plus en plus massivement parallèles.

En tant qu'orientations futures de recherche pour ce travail, nous avons identifié quelques perspectives pouvant considérées comme constituant des défis à relever, elles sont résumées ci-aprés :

1. Avec l'arrivée de ressources GPU dans les infrastructures du Cloud Computing, le défi consiste à revoir les approches que nous avons proposées sur les environnements

virtualisés. Ceci est une première étape naturelle vers les économies d'énergie. Toutefois, le critère de consommation d'énergie devrait être explicitement pris en charge dans la conception et la mise en œuvre de nos approches.

2. Récemment, la technologie des GPUs a connu une évolution qui est arrivée avec de nouvelles générations de dispositifs, y compris de fonctionnalités avancées. Par exemple, les machines basés sur l'architecture Kepler permettent un parallélisme dynamique (NVidia GPUDirect). Le parallélisme dynamique consiste à permettre au GPU de générer de nouveaux travaux par lui-même, la synchronisation sur les résultats, et le contrôle de l'ordonnancement par des voies matérielles dédiées et accélérées, le tout en n'impliquant pas le CPU, ainsi nous envisageons l'exploration de ces techniques comme extension potentielle et intéressante à notre travail.
3. Dans cette thèse, et dans la prise en charge du problème étudié, nous avons considéré une fonction d'évaluation linéaire dans les techniques évolutionnistes utilisées. Nous croyons que certaines fonctions pourraient être efficaces sur une architecture CPU monocœurs ou multicœurs, mais inefficace sur GPU vue la difficulté de leur parallélisation sur ce dispositif et vice versa. Une amélioration très encourageante dans le cadre de ce travail consiste à concevoir et mettre en œuvre une bibliothèque qui rassemble de nombreux types de fonctions utilisées pour le même problème sur des architectures mono et multicœurs CPU et GPU. En tant que niveau supplémentaire d'adaptabilité, la technique évolutionniste utilisée choisit dynamiquement et automatiquement la mise en œuvre qui convient le mieux à la machine d'exécution sous-jacente.

7.2.1 Nouvelle vision pour une future démarche exploitant les nouveaux accélérateurs graphiques

Dans les années qui suivirent le développement du Toolkit CUDA, il est indéniable que dans le monde computationnel, les GPUs ont pris de plus en plus d'importance, leurs structures ayant inspiré davantage les concepteurs de CPU.

Suite à l'étude élaborée dans cette thèse, et vu l'importance des algorithmes évolutionnaires dans plusieurs domaines, nous nous proposons la construction d'une architecture GPU générique qui intègre toutes les phases d'un processus évolutionnaire et ce selon une approche câblée.

La construction d'une telle architecture est un grand défi, vu le nombre important de critères que l'on doit considérer, tels que :

1. Avoir la capacité de traiter différents types de problèmes.
2. Avoir la capacité de traiter différents types de fonctions d'évaluation (mono-objective, multi-objective).
3. Avoir la capacité de traiter différents opérateurs évolutionnaires.
4. Avoir la capacité d'automatiser la génération des paramètres sensitives pour les kernels élaborés.

Annexe : PEvoRNN Performance : Mesures de performance d'un algorithme parallèle du point de vue GPU

1 Introduction

Malgré la révolution constatée dans le domaine du parallélisme boostée par l'apparition des architectures multi-coeurs, très peu d'applications ont pu être optimisées pour ces systèmes, ceci étant dû au fait qu'il soit encore difficile de développer efficacement et de façon rentable des applications parallèles. Ces dernières années, de plus en plus de chercheurs dans le domaine du calcul hautement parallèle (HPC) ont eu davantage recours à l'exploitation des GPUs pour accélérer des applications parallélisables. Bien que peu d'effort soit nécessaire pour adapter de manière fonctionnelle les applications aux GPUs, les programmeurs doivent consacrer davantage de temps et d'effort pour optimiser leurs applications dans le but d'atteindre de meilleures performances en termes de temps de calcul.

Afin de mieux appréhender les résultats de performance et optimiser efficacement les applications sur GPU, plusieurs thématiques intéressantes ont été abordées par la communauté GPGPU. Des modèles analytiques de performances ont été élaborés pour aider les développeurs à mieux analyser et comprendre les résultats de performances et localiser les différents goulots d'étranglement. La difficulté évidente pour l'analyse des performances des applications basées GPGPU réside dans le fait que l'architecture sous-jacente des GPUs est très peu documentée. La plupart des approches développées jusqu'à présent n'ont pas pu présenter un niveau d'optimisation efficace pour des applications du monde réel. Par ailleurs, l'architecture des GPUs évoluant très rapidement, la communauté doit fournir un effort soutenu à l'effet de perfectionner les modèles existants et développer de nouvelles approches qui permettraient aux développeurs de mieux optimiser les applications basées GPU.

Considérant toutes ces raisons, et vu le nombre restreint d'applications dans le domaine de la vie artificielle qui exploitent actuellement les GPUs, le processus consistant à estimer la performance des applications de ce domaine reste un défi tant l'estimation de performances dans son sens le plus large reste encore une question ouverte.

Dans cette partie, nous allons essayer de présenter les modèles analytiques existants dans le but d'étudier la possibilité d'en adapter un pour notre cas.

2 Outils d'analyse de performance des applications CUDA

Dans cette section, nous présentons quelques outils importants dont le rôle est d'assister les développeurs dans l'analyse des performances des applications développées sur la plateforme CUDA [26].

2.1 CUDA Visual Profiler

NVIDIA a prévu dans le Hardware GPU des compteurs et de nombreux événements de mesure de performances prédéfinis pouvant être utilisables. L'outil Visual Profiler de CUDA utilise des compteurs de performance et des événements pour collecter un grand nombre de statistiques utiles d'un Kernel en cours d'exécution. En outre, Visual profiler de CUDA délivre plusieurs informations importantes comme le nombre d'instructions, les demandes d'accès mémoire, les transactions, l'occupation mémoire, le nombre de branches, etc. Ces informations peuvent être utiles pour tout processus d'analyse de la performance. Il est doté, à la fois d'une ligne de commande ainsi que d'une interface graphique. Il peut, par ailleurs, analyser les données mentionnées précédemment via des indicateurs de performance tels que le débit global de la mémoire, le débit des instructions (instructions scalaires par cycle) (ipc), etc. C'est un outil développé par NVIDIA et livré avec CUDA Toolkit.

2.2 GPUOcelot

GPUOcelot est un Framework de compilation dynamique pour les systèmes hétérogènes. Il réalise de nombreux backends cibles pour des programmes CUDA. Ainsi, un programme écrit sous CUDA peut être exécuté sans recompilation sur les processeurs de la famille x86 en utilisant l'émulation PTX ou la traduction LLVM ainsi que l'exécution en mode natif sur AMD et NVIDIA GPU. Le terme PTX désigne un ensemble d'instructions de l'architecture virtuelle CUDA. Il fournit un modèle de programmation stable et un ensemble d'instructions pour la programmation parallèle à usage général.

En utilisant Ocelot, on peut aussi instrumentaliser le code PTX d'un Kernel CUDA tout en effectuant l'émulation en écrivant les analyseurs d'instructions, et ce, en utilisant l'API fournie avec Ocelot. Ces instructions analytiques peuvent être utilisées pour collecter un grand nombre de statistiques et d'indicateurs de performances pertinents à partir d'un Kernel CUDA.

2.3 Decuda

Le code d'assemblage PTX n'est pas une vraie représentation du code binaire qui se trouve dans le fichier binaire NVIDIA CUDA (.cubin). On peut noter une certaine optimisation qui a lieu entre le PTX et le fichier (.cubin). DECUDA est un désassembleur pour le format binaire utilisé avec NVIDIA CUDA. Il offre un aperçu des instructions internes générées et permet de mieux comprendre les aspects liés aux performances de ces instructions.

2.4 Cuobjdump

Cuobjdump est un outil permettant de manipuler les fichiers objets CUDA. Les entrées prises en charge sont pré-CUDA 3.0 cubins textuels ou cubins base-ELF CUDA 3.0 et plus.

Cuobjdump peut afficher les instructions de l'assembleur pour un Kernel particulier, ce qui est utile pour l'optimisation et le débogage. C'est un outil fourni par NVIDIA CUDA et disponible pour les développeurs enregistrés.

3 Comparaison des applications basées CPU vs GPU, Évaluation de performance et métriques de mesure

Jusqu'à présent, pour atteindre l'efficacité maximale en termes de performances et malgré la puissance de calcul offerte par les versions les plus puissantes des cartes graphiques les plus récentes, l'optimisation des Kernels GPU a constitué un défi perpétuel en raison des grandes différences organiques fonctionnelles entre le CPU et le GPU et le manque d'outils pour la programmation et l'analyse de performances. Pour la communauté des systèmes évolutionnistes, il est primordial de fournir le même effort et les mêmes conditions aux deux implémentations lors d'une analyse comparative entre le CPU et le GPU.

Pour les processeurs à haute performance, un riche corpus d'utilitaires qui proposent des modèles analytiques pour l'analyse des performances est fourni. Cependant, comme le calcul généraliste sur les processeurs GPUs est encore un domaine de recherche relativement nouveau, les modèles et les approches proposées pour comprendre les résultats de performances du GPU ont encore besoin plus de raffinement. Certains travaux intéressants sur la façon d'estimer les performances des applications CUDA ont mis à la disposition des développeurs de méthodes analytiques ou de simulation [109].

Dans cette partie de thèse, nous allons essayer de montrer comment calculer la performance du calcul parallèle dans le but d'établir une comparaison légale entre les applications CPU et GPU, puis nous analysons certaines mesures existantes et quelques méthodes analytiques.

3.1 Comparaison des applications CPU vs GPU

Il est assez courant et facile de trouver des travaux annonçant « Notre implémentation GPU montre un ordre de grandeur d'accélération contre le CPU ». Mais, ils font souvent une comparaison entre une mise en oeuvre GPU hautement optimisée et une mise en œuvre mono-coeur non optimisée. Ainsi, la question devant être posée est la suivante : «Comment arriver à une comparaison cohérente et équitable entre des applications CPUs et des applications GPUs ? ».

De nombreuses études ont été faites pour répondre à cette question, où la majorité des études ont prouvé qu'une comparaison équitable entre les performances du GPU et du CPU pour une application spécifique doit être basée sur une implémentation maintenue sur CPU à un niveau raisonnablement acceptable.

L'algorithme doit être parallélisé sur plusieurs coeurs du processeur. L'accès à la mémoire cache doit être convivial autant que faire se peut. Le code ne devrait pas confondre le prédicteur de branchement. Les opérations SIMD, comme SSE (Streaming SIMD Ex-

tensions) qui englobe un jeu de 70 instructions supplémentaires pour microprocesseurs x86, et qui sont cruciales pour exploiter le parallélisme au niveau instruction [98].

Les performances du GPU varient considérablement entre les différents dispositifs le composant. Afin de quantifier la performance d'un GPU, nous devons répondre à trois questions :

1. Dans combien de temps les données peuvent être envoyées au GPU ou relues de lui-même ?
2. Dans combien de temps le Kernel GPU peut lire et écrire des données ?
3. Dans combien de temps le GPU peut-il effectuer des calculs ?

Après la mesure de celles-ci, la performance du GPU peut être comparée à celle du CPU. Ceci fournit un repère sur le nombre de données ou de calculs nécessaires pour le GPU afin de fournir un avantage par rapport au CPU.

3.2 Informations et métriques nécessaires à la mesure de performances sous GPU ?

Afin de mesurer la performance des interactions CPU et GPU, nous disposons de trois sources d'information potentielles [129] :

1. **Le Langage de programmation GPU :** Deux standards émergents pour les développements GPU sont connus, CUDA et OpenCL, où chacun d'entre eux dispose d'un support pour observer les événements associés à l'utilisation du GPU, à partir de l'exécution du Kernel.
2. **Le pilote du GPU :** Représente l'interface du dispositif du GPU et est délivré par le fabricant du GPU.
3. **Le dispositif de GPU :** Représente un soutien pour la mesure de la performance du matériel.

Il est important de mentionner ici que les outils de mesure et d'analyse de performances doivent travailler avec ce qui est à leur disposition sur une plate-forme particulière qui peut limiter la couverture de ces outils.

3.3 Évaluation de la performance des approches à base de GPU utilisant les modèles analytiques

Certaines orientations générales sont prévues pour optimiser une application GPU. Pour trouver la solution optimale, les développeurs doivent modifier la configuration des combinaisons de réglages. Néanmoins, *l'architecture des GPUs et le résultat de performance des applications CUDA* restent un défi pour les développeurs.

Des approches analytiques ont été proposées afin de régler ces exigences.

Généralement, *le modèle de performance GPU analytique n'a pas besoin de tous les détails de l'architecture*, mais seulement d'un ensemble de paramètres qui pourraient être obtenus par des études comparatives ou présentes dans des documents publics.

Par ailleurs, des travaux intéressants sur la façon de prédire la performance des applications CUDA, exploitant des méthodes analytiques ou basées simulation ont été développés. L'avantage principal des modèles analytiques est qu'ils peuvent être utilisés statiquement sans exécuter une application sur GPU.

Le tableau ci-dessous représente un récapitulatif des travaux les plus importants dans ce domaine, et dans lequel où nous indiquons le principe et les limites de chaque modèle.

Récapitulation

Auteurs / années	**Modèle**	**Idée**	**Les Limites**
Hong et Kim (2009) [87]	Le modèle MWP-CWP (Memory Warp Parallelism, Computation Warp Parallelism)	L'idée clé consiste à estimer le nombre de demandes parallèles d'accès mémoire.	Le modèle est du type coarse grain, car il suffit de séparer l'exécution d'une application en période de calcul plus la période d'accès mémoire.
Sim et al. (2012) [173]	Le modèle MWP-CWP étendu	Basé sur le modèle MWP-CWP, exige une variété d'informations, y compris les compteurs matériels à partir d'une exécution réelle.	Le modèle nécessite une exécution réelle du programme de collecte des informations d'exécution, ce qui rend la prévision des performances moins significative.
Zhang et Owens (2011) [209]	Modèle quantitative d'analyse de performance	L'idée générale consiste à modéliser le processeur GPU comme trois composants majeurs : le pipeline instruction, la mémoire partagée et la mémoire globale et de modéliser l'exécution d'une application GPU comme des instructions servies aux différentes composantes en fonction du type d'instruction.	Le modèle divise simplement le processeur GPU en 3 composants séparés. Le temps d'exécution de chaque composant est calculé séparément. Toutefois, dans l'exécution réelle d'une application sur GPU, les instructions mathématiques ainsi que les opérations mémoire ont une dépendance complexe et l'exécution des différents composants peut avoir un impact de l'une sur l'autre.

Meng et al. (2011) [135]	Projection de la performance GPU à partir du code squelette CPU.	Les auteurs affirment que ce Framework peut estimer l'avantage de performance des applications GPU sans programmation réelle hard ou soft. Le Framework permet aux développeurs d'évaluer la performance réalisable à partir du code squelette CPU.	La limite la plus évidente de cette proposition réside dans la fait que l'utilisateur a besoin de développer un squelette de code annoté pour chaque code existant séparément.

TABLE 1: Les modèles analytiques d'estimation de performance existants.

D'après [88], les Kernels GPU peuvent être classés en deux catégories :

1. **Borne de calcul :** elle correspond à des conditions où le débit de calcul est inférieur au débit mémoire, ce qui signifie que les demandes d'accès à la mémoire globale arrivent à l'interface de la mémoire globale à une vitesse relativement lente.
2. **Borne de mémoire :** cette catégorie renvoie à la situation où le débit calcul est plus grand que le débit mémoire, ce qui signifie que les demandes d'accès mémoire arrivent plus rapidement que la vitesse avec laquelle ces demandes quittent la mémoire à laquelle elles sont arrivées.

Par ailleurs, la philosophie de l'architecture GPU est de découvrir les opérations ayant une latence longue avec entrelacement d'exécution des opérations de calcul à partir d'une grande quantité de Warp. La performance finale dépend principalement de l'efficacité de la latence cachée. Par conséquent, le temps d'exécution peut être décomposé en deux parties : la durée d'exécution du calcul et la partie non couverte de la latence de la mémoire.

Dû à un taux de calcul d'accès mémoire ou à un accès coalescé parfait à la mémoire globale, le débit de calcul est plus grand que le débit d'accès mémoire, et les demandes d'accès mémoire peuvent être manipulées à une plus grande vitesse que celles arrivant à l'interface mémoire.

En 2014, Hu et coll. [88] ont proposé un modèle qui partage plusieurs caractéristiques avec le modèle MWP-CWP proposé par Hong et Kim en 2009 :

1. Les deux modèles analytiques extraient le parallélisme à partir des Kernels GPU, de la granularité des Warps et la totalité du temps exécution est comptabilisé sur la capacité de dissimulation de la latence des accès mémoire par les calculs.
2. La latence d'une transaction non coalescée au niveau de la mémoire globale

A part les points communs, les deux modèles présentent des différences remarquables :

1. Dans le travail de Hu et coll., le retard du délai entre deux accès mémoires non coalescés est référé à la latence des accès à la DRAM d'une seule transaction mémoire, qui peut être et qui peut être calculé sur la base des valeurs délivrées dans la fiche technique de la GDDR, au lieu de prolifération.

2. Hu et coll. ont construit un modèle de pipeline pour les accès à la mémoire globale et ont utilisé le pipeline throughput pour caractériser la performance de la mémoire.
3. Les calculs et les opérations des accès mémoires aux Kernels sont séparés et les performances des deux parties sont représentées respectivement par la borne de calcul ainsi que la borne de la mémoire étendue.

Comme les programmes GPU sont classifiés en borne de calcul et de mémoire, l'amélioration de la performance potentielle nécessite une augmentation ainsi qu'une amélioration de ces valeurs.

4 Récapitulation

Dans cette section, plusieurs modèles importants d'analyse de performances pour les GPUs ont été brièvement présentés. En la comparant avec l'approche simulation, l'approche analytique est beaucoup plus facile à construire et à utiliser. Généralement, l'approche analytique utilise uniquement un ensemble de paramètres matériels qui sont soit fournis par les constructeurs ou susceptibles d'être recueillies à partir des benchmarks. En fait, l'approche simulation et l'approche analytique ne présentent pas beaucoup de différences. Par ailleurs, plus des paramètres matériels sont introduits dans les modèles, et plus des implémentations sous-jacentes sont exploitées. De plus, les modèles analytiques devraient être les plus précis. Une tendance claire est que de nombreux modèles récents d'analyse de performances tentent d'utiliser directement le code machine et avant que les modèles analytiques n'utilisent normalement le niveau algorithmique, (l'information C/C++ du niveau PTX de l'assembleur de CUDA).

D'une manière générale et d'un point de vue de l'utilisateur, nous devons adopter le modèle analytique et ce pour avoir les raisons suivantes :

- tout d'abord, un modèle analytique doit pouvoir être construit par le biais de paramètres pouvant être effectivement obtenus.
- deuxièmement, un modèle devrait être en mesure de prédire la performance de certaines mises en œsuvre avec suffisamment de précision.
- en troisième lieu, un modèle devrait être en mesure de briser le temps d'exécution de sorte que les pénalités de performances pourraient être quantifiées.

Pour satisfaire ces exigences, nous croyons qu'un bon outil de prédiction de l'analyse des performances devrait être une combinaison entre un simulateur fonctionnel et un outil de synchronisation analytique.

Pour comprendre l'état d'exécution sous-jacent d'une demande de GPU, l'entrée du modèle analytique doit être le code machine.

Le but initial de ce chapitre était de prédire la performance des Kernels proposés et implémentés dans notre système pEvoRNN en utilisant un des modèles analytiques présentés dans cette partie. Cependant, et vue la complexité de notre système PEvoRNN, les modèles proposés pour prédire la performance doivent avoir des améliorations et plus de clarifications afin qu'ils soient adaptés aux différents types de problèmes.

Bibliographie

[1] Enrique Alba and Marco Tomassini. Parallelism and evolutionary algorithms. *Evolutionary Computation, IEEE Transactions on*, 6(5) :443–462, 2002.

[2] Ramnik Arora, Rupesh Tulshyan, and Kalyanmoy Deb. Parallelization of binary and real-coded genetic algorithms on cuda. 2010.

[3] Adham Atyabi and David MW Powers. Review of classical and heuristic-based navigation and path planning approaches. *International Journal of Advancements in Computing Technology*, 5(14), 2013.

[4] Haldun Aytug and Gary J Koehler. Stopping criteria for finite length genetic algorithms. *INFORMS Journal on Computing*, 8(2) :183–191, 1996.

[5] Thomas Back. *Evolutionary algorithms in theory and practice*. Oxford Univ. Press, 1996.

[6] Michela Becchi, Surendra Byna, Srihari Cadambi, and Srimat Chakradhar. Data-aware scheduling of legacy kernels on heterogeneous platforms with distributed memory. In *Proceedings of the twenty-second annual ACM symposium on Parallelism in algorithms and architectures*, pages 82–91. ACM, 2010.

[7] Nour El-Houda BENALIA, NourEddine DJEDI, Salim BITAM, Nesrine OUANNES, and Yves DUTHEN. An improved cuda-based hybrid metaheuristic for fast controller of an evolutionary robot (in press). *Int. J. Embedded Systems*, x(x) :xxx–xxx, 2015.

[8] Hans-Georg Beyer, Hans-Paul Schwefel, and Ingo Wegener. How to analyse evolutionary algorithms. *Theoretical Computer Science*, 287(1) :101–130, 2002.

[9] Pavol Bezák. Gpu accelerated robotic arm optimal trajectory generation using genetic algorithms.

[10] Yann Boniface. Un outil de développement parallèle des réseaux de neurone. In *Neurosciences et Sciences pour l'Ingénieur*, pages 4–p, 2000.

[11] Yann Boniface, Frédéric Alexandre, and Stéphane Vialle. Aide à la parallélisation des réseaux connexionnistes. In *Neurosciences et Sciences pour l'Ingénieur*, pages 4–p, 1998.

[12] Dane L Brown, Mehrdad Ghaziasgar, and James Connan. Faster upper body pose estimation using cuda. In *Proc. Southern Africa Telecommunication Networks and Applications Conference*, 2011.

[13] Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, and Pat Hanrahan. Brook for gpus : stream computing on graphics hardware. In *ACM Transactions on Graphics (TOG)*, volume 23, pages 777–786. ACM, 2004.

[14] Erick Cantú-Paz. Designing efficient master-slave parallel genetic algorithms. 1997.

[15] Genci Capi and Kenji Doya. Evolution of recurrent neural controllers using an extended parallel genetic algorithm. *Robotics and Autonomous Systems*, 52(2) :148–159, 2005.

[16] Kristofor D Carlson, Jayram Moorkanikara Nageswaran, Nikil Dutt, and Jeffrey L Krichmar. An efficient automated parameter tuning framework for spiking neural networks. *Frontiers in neuroscience*, 8, 2014.

[17] Maria Charalambous, Pedro Trancoso, and Alexandros Stamatakis. Initial experiences porting a bioinformatics application to a graphics processor. In *Advances in Informatics*, pages 415–425. Springer, 2005.

[18] Alexandre Chariot. *Quelques applications de la programmation des processeurs graphiques à la simulation neuronale et à la vision par ordinateur*. PhD thesis, Ecole des Ponts ParisTech, 2008.

[19] Darren M Chitty. Improving the performance of gpu-based genetic programming through exploitation of on-chip memory. *Soft Computing*, pages 1–20, 2014.

[20] Scott Christley, Briana Lee, Xing Dai, and Qing Nie. Integrative multicellular biological modeling : a case study of 3d epidermal development using gpu algorithms. *BMC systems biology*, 4(1) :107, 2010.

[21] Adam Coates, Paul Baumstarck, Quoc Le, and Andrew Y Ng. Scalable learning for object detection with gpu hardware. In *Intelligent Robots and Systems, 2009. IROS 2009. IEEE/RSJ International Conference on*, pages 4287–4293. IEEE, 2009.

[22] Florent Cohen, Philippe Decaudin, and Fabrice Neyret. Gpu-based lighting and shadowing of complex natural scenes. In *ACM SIGGRAPH 2004 Posters*, page 91. ACM, 2004.

[23] Mark Colbert and Jaroslav Krivanek. Real-time dynamic shadows for image-based lighting. *ShaderX 7-Advanced Rendering Techniques*, 2009.

[24] Teodor Gabriel Crainic and Michel Toulouse. Parallel meta-heuristics. In *Handbook of metaheuristics*, pages 497–541. Springer, 2010.

[25] Valentin CRISTEA. Conception and design of parallel and distributed applications. *Proceedings of the Romanian academy, Series A*, 5(1) :1–8, 2004.

[26] C Cuda. Programming guide. *NVIDIA Corporation, July*, 2012.

[27] Van-Dat Cung, Simone L Martins, Celso C Ribeiro, and Catherine Roucairol. Strategies for the parallel implementation of metaheuristics. In *Essays and surveys in metaheuristics*, pages 263–308. Springer, 2002.

[28] Paulo Henrique da Fonseca Silva, Adaildo Gomes D'Assunção, Clarissa de Lucena Nóbrega, and Marcelo Ribeiro da Silva. *Application of Bio-Inspired Algorithms and Neural Networks for Optimal Design of Fractal Frequency Selective Surfaces*. INTECH Open Access Publisher, 2012.

[29] B Kirk David and W Hwu Wen-mei. Programming massively parallel processors : A hands-on approach. *Burlington, MA, USA*, 2010.

[30] Laurence Dawson and Iain Stewart. Improving ant colony optimization performance on the gpu using cuda. In *Evolutionary Computation (CEC), 2013 IEEE Congress on*, pages 1901–1908. IEEE, 2013.

[31] CRYPTOGRAPHIE ET SÉCURITÉ DE. Cours des méthodes de résolution exactes heuristiques et métaheuristiques.

[32] Joachim de Greeff and Cognitive Artificial Intelligence. *Evolving communication in evolutionary robotics*. PhD thesis, Citeseer, 2007.

[33] Kalyanmoy Deb and Ram Bhushan Agrawal. Simulated binary crossover for continuous search space. *Complex systems*, 9(2) :115–148, 1995.

[34] Kailash Devrari and K Vinay Kumar. Fast face detection using graphics processor. *International Journal of Computer Science and Information Technologies*, 2(3) :1082–1086, 2011.

[35] Helge Ülo Dinkelbach, Julien Vitay, Frederik Beuth, and Fred H Hamker. Comparison of gpu-and cpu-implementations of mean-firing rate neural networks on parallel hardware. *Network : Computation in Neural Systems*, 23(4) :212–236, 2012.

[36] Ringo Doe. This is a test entry of type @ONLINE, June 2009.

[37] Balázs Domonkos and Gábor Jakab. A programming model for gpu-based parallel computing with scalability and abstraction. In *Proceedings of the 25th Spring Conference on Computer Graphics*, pages 103–111. ACM, 2009.

[38] Marco Dorigo. Ant colony optimization. *Scholarpedia*, 2(3) :1461, 2007.

[39] Stefan Droste, Thomas Jansen, and Ingo Wegener. On the optimization of unimodal functions with the (1+ 1) evolutionary algorithm. In *Parallel Problem Solving from NatureâPPSN V*, pages 13–22. Springer, 1998.

[40] Stefan Droste, Thomas Jansen, and Ingo Wegener. A rigorous complexity analysis of the (1+ 1) evolutionary algorithm for separable functions with boolean inputs. *Evolutionary Computation*, 6(2) :185–196, 1998.

[41] Stefan Droste, Thomas Jansen, and Ingo Wegener. On the analysis of the (1+ 1) evolutionary algorithm. *Theoretical Computer Science*, 276(1) :51–81, 2002.

[42] Dominic Dunlop, Sebastien Varrette, and Pascal Bouvry. On the use of a genetic algorithm in high performance computer benchmark tuning. In *Performance Evaluation of Computer and Telecommunication Systems, 2008. SPECTS 2008. International Symposium on*, pages 105–113. IEEE, 2008.

[43] Antoine Dutot and Damien Olivier. Optimisation par essaim de particules application au problème des n-reines. *Laboratoire Informatique du Havre, Université du Havre*, 2002.

[44] AE Eiben and JE Smith. Introduction to evolutionary computing (natural computing series). 2008.

[45] Agoston E Eiben and Günter Rudolph. Theory of evolutionary algorithms : a bird's eye view. *Theoretical Computer Science*, 229(1) :3–9, 1999.

[46] Agoston E Eiben and James E Smith. *Introduction to evolutionary computing*. Springer Science & Business Media, 2003.

[47] Anders Eklund, Paul Dufort, Daniel Forsberg, and Stephen M LaConte. Medical image processing on the gpu–past, present and future. *Medical image analysis*, 17(8) :1073–1094, 2013.

[48] Christer Ericson. *Real-time collision detection*. CRC Press, 2004.

[49] CONCEPTION ET. *Franck ELIE*. PhD thesis, UNIVERSITÉ DâORLÉANS, 1997.

[50] Dario Floreano, Claudio Mattiussi, and Bio-Inspired Artificial Intelligence. Theories, methods, and technologies, 2008.

[51] David B Fogel. An analysis of evolutionary programming. *Fogel and Atmar*, 684 :43–51, 1992.

[52] Ka-Ling Fok, Tien-Tsin Wong, and Man-Leung Wong. Evolutionary computing on consumer graphics hardware. *IEEE Intelligent Systems*, (2) :69–78, 2007.

[53] Christian Folkers and Wolfgang Ertel. High performance realtime vision for mobile robots on the gpu. In *VISAPP (Workshop on on Robot Vision)*, pages 27–35, 2007.

[54] James Fung and Steve Mann. Openvidia : parallel gpu computer vision. In *Proceedings of the 13th annual ACM international conference on Multimedia*, pages 849–852. ACM, 2005.

[55] David Gallup. *Efficient 3D reconstruction of large-scale urban environments from street-level video*. PhD thesis, THE UNIVERSITY OF NORTH CAROLINA AT CHAPEL HILL, 2011.

[56] Yong Gao. Population size and sampling complexity in genetic algorithms. In *Proc. of the Bird of a Feather Workshops*, pages 178–181. Citeseer, 2003.

[57] Michael R Garey and David S Johnson. Computer and intractability. *A Guide to the Theory of NP-Completeness*, 1979.

[58] Aloysius George, BR Rajakumar, and D Binu. Genetic algorithm based airlines booking terminal open/close decision system. In *proceedings of the International Conference on Advances in Computing, Communications and Informatics*, pages 174–179. ACM, 2012.

[59] Stevie Giovanni and KangKang Yin. Locotest : deploying and evaluating physics-based locomotion on multiple simulation platforms. In *Motion in Games*, pages 227–241. Springer, 2011.

[60] Stephane Gobron, Francois Devillard, and Bernard Heit. Retina simulation using cellular automata and gpu programming. *Machine Vision and Applications*, 18(6) :331–342, 2007.

[61] David E Golberg. Genetic algorithms in search, optimization, and machine learning. *Addion wesley*, 1989, 1989.

[62] David E Goldberg. Sizing populations for serial and parallel genetic algorithms. In *Proceedings of the 3rd International Conference on Genetic Algorithms*, pages 70–79. Morgan Kaufmann Publishers Inc., 1989.

[63] David E Goldberg, Kalyanmoy Deb, and James H Clark. Genetic algorithms, noise, and the sizing of populations. *Complex systems*, 6 :333–362, 1991.

[64] David Edward Goldberg. *Optimal initial population size for binary-coded genetic algorithms*. Clearinghouse for Genetic Algorithms, Department of Engineering Mechanics, University of Alabama, 1985.

[65] Marin Golub and Leo Budin. An asynchronous model of global parallel genetic algorithms. In *Second ICSC Symposium on Engineering of Intelligent Systems EIS2000, University of Paisley, Scotland, UK*, pages 353–359, 2000.

[66] Marin Golub and Domagoj JakoboviC. A new model of global parallel genetic algorithm. In *Information Technology Interfaces, 2000. ITI 2000. Proceedings of the 22nd International Conference on*, pages 363–368. IEEE, 2000.

[67] Marin Golub, Domagoj Jakobović, and Leo Budin. Parallelization of elimination tournament selection without synchronization. In *Proc. 5th Int. Conf. on Intelligent Engineering Systems INES*, pages 16–18, 2001.

[68] C Gondro and BP Kinghorn. A simple genetic algorithm for multiple sequence alignment. *Genetics and Molecular Research*, 6(4) :964–982, 2007.

[69] Pablo González-Nalda and Blanca Cases. Topos 2 : spiking neural networks for bipedal walking in humanoid robots. In *Hybrid Artificial Intelligent Systems*, pages 479–485. Springer, 2011.

[70] David Greenhalgh and Stephen Marshall. Convergence criteria for genetic algorithms. *SIAM Journal on Computing*, 30(1) :269–282, 2000.

[71] John J Grefenstette. *Parallel Adaptive Algorithms for Function Optimization :(preliminary Report)*. Computer Science Department, Vanderbilt University, 1981.

[72] Khronos OpenCL Working Group et al. The opencl specification, version 1.1, 2010. *Document Revision*, 44.

[73] Jin-Kao Hao, Philippe Galinier, and Michel Habib. Métaheuristiques pour lâoptimisation combinatoire et lâaffectation sous contraintes. *Revue dâintelligence artificielle*, 13(2) :283–324, 1999.

[74] Jin-Kao Hao and Christine Solnon. Méta-heuristiques et intelligence artificielle.

[75] Simon Harding and Wolfgang Banzhaf. Fast genetic programming and artificial developmental systems on gpus. In *High Performance Computing Systems and Applications, 2007. HPCS 2007. 21st International Symposium on*, pages 2–2. IEEE, 2007.

[76] Simon Harding and Wolfgang Banzhaf. Fast genetic programming on gpus. In *Genetic Programming*, pages 90–101. Springer, 2007.

[77] Simon L Harding, Julian F Miller, and Wolfgang Banzhaf. Self-modifying cartesian genetic programming. In *Cartesian Genetic Programming*, pages 101–124. Springer, 2011.

[78] George Harik, Erick Cantú-Paz, David E Goldberg, and Brad L Miller. The gambler's ruin problem, genetic algorithms, and the sizing of populations. *Evolutionary Computation*, 7(3) :231–253, 1999.

[79] Jun He and Xin Yao. Drift analysis and average time complexity of evolutionary algorithms. *Artificial Intelligence*, 127(1) :57–85, 2001.

[80] Jun He and Xin Yao. From an individual to a population : An analysis of the first hitting time of population-based evolutionary algorithms. *Evolutionary Computation, IEEE Transactions on*, 6(5) :495–511, 2002.

[81] Everton Hermann. *Interactive Physical Simulation on Multi-core and Multi-GPU Architectures*. PhD thesis, Institut National Polytechnique de Grenoble-INPG, 2010.

[82] Jose Herrera, Eduardo Huedo, Rubén S Montero, and Ignacio Martín Llorente. A grid-oriented genetic algorithm. In *Advances in Grid Computing-EGC 2005*, pages 315–322. Springer, 2005.

[83] Manato Hirabayashi, Shinpei Kato, Masato Edahiro, and Yuki Sugiyama. Toward gpu-accelerated traffic simulation and its real-time challenge. 2012.

[84] Roger W Hockney and Chris R Jesshope. *Parallel Computers 2 : architecture, programming and algorithms*, volume 2. CRC Press, 1988.

[85] Johannes Hofmann. Evolving neural networks on gpus. GECCO, 2011.

[86] Johannes Hofmann, Steffen Limmer, and Dietmar Fey. Performance investigations of genetic algorithms on graphics cards. *Swarm and Evolutionary Computation*, 12 :33–47, 2013.

[87] Sunpyo Hong and Hyesoon Kim. An analytical model for a gpu architecture with memory-level and thread-level parallelism awareness. In *ACM SIGARCH Computer Architecture News*, volume 37, pages 152–163. ACM, 2009.

[88] Zhidan Hu, Guangming Liu, and Wenrui Dong. A throughput-aware analytical performance model for gpu applications. In *Advanced Computer Architecture*, pages 98–112. Springer, 2014.

[89] Thomas Jansen. *Analyzing evolutionary algorithms : The computer science perspective*. Springer Science & Business Media, 2013.

[90] Thomas Jansen. General limits in black-box optimization. In *Analyzing Evolutionary Algorithms*, pages 45–84. Springer, 2013.

[91] Thomas Jansen. Black-box complexity for bounding the performance of randomized search heuristics. In *Theory and Principled Methods for the Design of Metaheuristics*, pages 85–110. Springer, 2014.

[92] Thomas Jansen, Kenneth A De Jong, and Ingo Wegener. On the choice of the offspring population size in evolutionary algorithms. *Evolutionary Computation*, 13(4) :413–440, 2005.

[93] Thomas Jansen and Ingo Wegener. *On the analysis of evolutionary algorithmsâa proof that crossover really can help*. Springer, 1999.

[94] Thomas Jansen and Ingo Wegener. Evolutionary algorithms-how to cope with plateaus of constant fitness and when to reject strings of the same fitness. *Evolutionary Computation, IEEE Transactions on*, 5(6) :589–599, 2001.

[95] Thomas Jansen, Ingo Wegener, and PM Kaufmann. On the utility of populations in evolutionary algorithms. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO)*, pages 1034–1041. Citeseer, 2001.

[96] Thomas Jansen and Christine Zarges. Analysis of evolutionary algorithms : from computational complexity analysis to algorithm engineering. In *Proceedings of the 11th workshop proceedings on Foundations of genetic algorithms*, pages 1–14. ACM, 2011.

[97] Jiri Jaros. Multi-gpu island-based genetic algorithm for solving the knapsack problem. In *Evolutionary Computation (CEC), 2012 IEEE Congress on*, pages 1–8. IEEE, 2012.

[98] Jiri Jaros and Petr Pospichal. A fair comparison of modern cpus and gpus running the genetic algorithm under the knapsack benchmark. In *Applications of Evolutionary Computation*, pages 426–435. Springer, 2012.

[99] Simon Jones, Matthew Studley, and Alan Winfield. Mobile gpgpu acceleration of embodied robot simulation.

[100] Mark Joselli, Esteban Clua, Anselmo Montenegro, Aura Conci, and Paulo Pagliosa. A new physics engine with automatic process distribution between cpu-gpu. In *Proceedings of the 2008 ACM SIGGRAPH symposium on Video games*, pages 149–156. ACM, 2008.

[101] Jun-Sik Kim, Myung Hwangbo, and Takeo Kanade. Parallel algorithms to a parallel hardware : Designing vision algorithms for a gpu. In *Computer Vision Workshops (ICCV Workshops), 2009 IEEE 12th International Conference on*, pages 862–869. IEEE, 2009.

[102] David B Kirk and W Hwu Wen-mei. *Programming massively parallel processors : a hands-on approach*. Newnes, 2012.

[103] John R Koza. *Genetic programming : on the programming of computers by means of natural selection*, volume 1. MIT press, 1992.

[104] John R Koza. Genetic programming 2-automatic discovery of reusable programs. complex adaptive systems, 1994.

[105] John R Koza, Martin A Keane, Matthew J Streeter, William Mydlowec, Jessen Yu, and Guido Lanza. *Genetic programming IV : Routine human-competitive machine intelligence*, volume 5. Springer Science & Business Media, 2006.

[106] Pavel Krömer, Václav Snášel, Jan Platoš, and Ajith Abraham. Many-threaded implementation of differential evolution for the cuda platform. In *Proceedings of the 13th annual conference on Genetic and evolutionary computation*, pages 1595–1602. ACM, 2011.

[107] Rudolf Kruse, Christian Borgelt, Frank Klawonn, Christian Moewes, Georg Ruß, and Matthias Steinbrecher. Computational intelligence. *Computational Intelligence, Vieweg+ Teubner Verlag/Springer Fachmedien Wiesbaden GmbH, Wiesbaden*, 2012.

[108] Patrick Lacz and John C Hart. Procedural geometry synthesis on the gpu. *nation*, 8 :21, 2004.

[109] Junjie Lai. *Throughput-oriented analytical models for performance estimation on programmable hardware accelerators*. PhD thesis, Université Rennes 1 ; Université européenne de Bretagne, 2013.

[110] William B Langdon. Evolving genechip correlation predictors on parallel graphics hardware. In *Evolutionary Computation, 2008. CEC 2008.(IEEE World Congress on Computational Intelligence). IEEE Congress on*, pages 4151–4156. IEEE, 2008.

[111] William B Langdon and Andrew P Harrison. Gp on spmd parallel graphics hardware for mega bioinformatics data mining. *Soft Computing*, 12(12) :1169–1183, 2008.

[112] William B Langdon, Marc Modat, Justyna Petke, and Mark Harman. Improving 3d medical image registration cuda software with genetic programming. In *Proceedings of the 2014 conference on Genetic and evolutionary computation*, pages 951–958. ACM, 2014.

[113] Steve Lawrence, C Lee Giles, and Ah Chung Tsoi. What size neural network gives optimal generalization ? convergence properties of backpropagation. 1998.

[114] R Lindblad, Peter Nordin, and Krister Wolff. Evolving 3d model interpretation of images using graphics hardware. In *wcci*, pages 225–230. IEEE, 2002.

[115] Markus Lipp, Peter Wonka, and Michael Wimmer. Parallel generation of multiple l-systems. *Computers & Graphics*, 34(5) :585–593, 2010.

[116] Hod Lipson. Evolutionary robotics and open-ended design automation. *Biomimetics*, 17(9) :129–155, 2005.

[117] Jinfeng Liu and Lei Guo. Implementation of neural network backpropagation in cuda. In *Intelligence Computation and Evolutionary Computation*, pages 1021–1027. Springer, 2013.

[118] Weiguo Liu, Bertil Schmidt, Gerrit Voss, Andre Schroder, and Wolfgang Muller-Wittig. Bio-sequence database scanning on a gpu. In *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*, pages 8–pp. IEEE, 2006.

[119] Youquan Liu and Suvranu De. Cuda-based real time surgery simulation. *Studies in Health Technology and Informatics*, 132 :260–262, 2007.

[120] Zhongwen Luo, Hongzhi Liu, and Xincai Wu. Artificial neural network computation on graphic process unit. In *Neural Networks, 2005. IJCNN'05. Proceedings. 2005 IEEE International Joint Conference on*, volume 1, pages 622–626. IEEE, 2005.

[121] TV Luong. Métaheuristiques parallèles sur gpu. *Université des Sciences et Technologie de Lille*, page 2, 2011.

[122] TS Lyes and KA Hawick. Implementing stereo vision of gpu-accelerated scientific simulations using commodity hardware. In *Proc. International Conference on Computer Graphics and Virtual Reality (CGVRâ11). Number CGV4047, Las Vegas, USA, CSREA*, pages 76–82, 2011.

[123] Liam P Maguire, T Martin McGinnity, Brendan Glackin, Arfan Ghani, Ammar Belatreche, and Jim Harkin. Challenges for large-scale implementations of spiking neural networks on fpgas. *Neurocomputing*, 71(1) :13–29, 2007.

[124] Ogier Maitre. *GPGPU for Evolutionary Algorithms*. PhD thesis, 2011.

[125] Ogier Maitre, Laurent A Baumes, Nicolas Lachiche, Avelino Corma, and Pierre Collet. Coarse grain parallelization of evolutionary algorithms on gpgpu cards with easea. In *Proceedings of the 11th Annual conference on Genetic and evolutionary computation*, pages 1403–1410. ACM, 2009.

[126] Ogier Maitre, Frédéric Krüger, Stéphane Querry, Nicolas Lachiche, and Pierre Collet. Easea : specification and execution of evolutionary algorithms on gpgpu. *Soft Computing*, 16(2) :261–279, 2012.

[127] Ogier Maitre, Nicolas Lachiche, Philippe Clauss, Laurent Baumes, Avelino Corma, and Pierre Collet. Efficient parallel implementation of evolutionary algorithms on gpgpu cards. In *Euro-Par 2009 Parallel Processing*, pages 974–985. Springer, 2009.

[128] Ogier Maitre, Deepak Sharma, Nicolas Lachiche, and Pierre Collet. Dispar-tournament : a parallel population reduction operator that behaves like a tournament. In *Applications of Evolutionary Computation*, pages 284–293. Springer, 2011.

[129] Allen D Malony, Scott Biersdorff, Sameer Shende, Heike Jagode, Stanimire Tomov, Guido Juckeland, Robert Dietrich, Duncan Poole, and Christopher Lamb. Parallel performance measurement of heterogeneous parallel systems with gpus. In *Parallel Processing (ICPP), 2011 International Conference on*, pages 176–185. IEEE, 2011.

[130] Ricardo Marroquim and André Maximo. Introduction to gpu programming with glsl. In *Computer Graphics and Image Processing (SIBGRAPI TUTORIALS), 2009 Tutorials of the XXII Brazilian Symposium on*, pages 3–16. IEEE, 2009.

[131] Michael D McCool, Zheng Qin, and Tiberiu S Popa. Shader metaprogramming. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 57–68. Eurographics Association, 2002.

[132] Nouredine Melab, El-Ghazali Talbi, et al. Algorithmes évolutionnaires parallèles sur gpu. In *Manifestation des Jeunes Chercheurs en Sciences et Technologies de l'Information et de la Communication (Majecstic)*, 2010.

[133] Nouredine Melab, El-Ghazali Talbi, et al. Gpu-based island model for evolutionary algorithms. In *Proceedings of the 12th annual conference on Genetic and evolutionary computation*, pages 1089–1096. ACM, 2010.

[134] Nouredine Melab, El-Ghazali Talbi, et al. Parallel hybrid evolutionary algorithms on gpu. In *IEEE Congress on Evolutionary Computation (CEC)*, 2010.

[135] Jiayuan Meng, Vitali A Morozov, Kalyan Kumaran, Venkatram Vishwanath, and Thomas D Uram. Grophecy : Gpu performance projection from cpu code skeletons. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, page 14. ACM, 2011.

[136] Melanie Mitchell. *An introduction to genetic algorithms*. MIT press, 1998.

[137] Oscar Montiel, Roberto Sepúlveda, and Ulises Orozco-Rosas. Optimal path planning generation for mobile robots using parallel evolutionary artificial potential field. *Journal of Intelligent & Robotic Systems*, pages 1–21.

[138] Asim Munawar, Mohamed Wahib, Masaharu Munetomo, and Kiyoshi Akama. A survey : Genetic algorithms and the fast evolving world of parallel computing. In *High Performance Computing and Communications, 2008. HPCC'08. 10th IEEE International Conference on*, pages 897–902. IEEE, 2008.

[139] Jayram Moorkanikara Nageswaran, Nikil Dutt, Jeffrey L Krichmar, Alex Nicolau, and Alex Veidenbaum. Efficient simulation of large-scale spiking neural networks using cuda graphics processors. In *Neural Networks, 2009. IJCNN 2009. International Joint Conference on*, pages 2145–2152. IEEE, 2009.

[140] Gabriel Noaje, Christophe Jaillet, and Michaël Krajecki. Programmation multigpu–openmp versus mpi.

[141] Stefano Nolfi, Dario Floreano, Orazio Miglino, and Francesco Mondada. How to evolve autonomous robots : Different approaches in evolutionary robotics. In *Artificial life IV : Proceedings of the 4th International Workshop on Artificial Life*, number LIS-CONF-1994-002, pages 190–197. MA : MIT Press, 1994.

[142] Kazuhiro Ohkura, Toshiyuki Yasuda, Yoshiyuki Matsumura, and Masaki Kadota. Gpu implementation of food-foraging problem for evolutionary swarm robotics systems. In *Swarm Intelligence*, pages 238–245. Springer, 2014.

[143] Masashi Oiso, Yoshiyuki Matsumura, Toshiyuki Yasuda, and Kazuhiro Ohkura. Implementing genetic algorithms to cuda environment using data parallelization. *Tehnički vjesnik*, 18(4) :511–517, 2011.

[144] Pietro S Oliveto, Jun He, and Xin Yao. Time complexity of evolutionary algorithms for combinatorial optimization : A decade of results. *International Journal of Automation and Computing*, 4(3) :281–293, 2007.

[145] Nesrine Ouannes, NourEddine Djedi, Yves Duthen, and Hervé Luga. Gait evolution for humanoid robot in a physically simulated environment. In *Intelligent Computer Graphics 2011*, pages 157–173. Springer, 2012.

[146] Ritesh A Patel, Yao Zhang, Jason Mak, Andrew Davidson, and John D Owens. *Parallel lossless data compression on the GPU*. IEEE, 2012.

[147] Ashwini A Patil and Pankaja A Shahapure. A gpu-accelerated framework for image processing and computer vision. 2013.

[148] Catalin Patulea, Robert Peace, and James Green. Cuda-accelerated genetic feedforward-ann training for data mining. In *Journal of Physics : Conference Series*, volume 256, page 012014. IOP Publishing, 2010.

[149] Martin Peniak, Anthony Morse, Christopher Larcombe, Salomon Ramirez-Contla, and Angelo Cangelosi. Aquila : An open-source gpu-accelerated toolkit for cognitive and neuro-robotics research. In *Neural Networks (IJCNN), The 2011 International Joint Conference on*, pages 1753–1760. IEEE, 2011.

[150] Pavel Petrovic. Distributed system for evolutionary robotics experiments. Technical report, Technical Report 05/04, Norwegian University of Science and Technology, Department of Computer and Information Science, 2004.

[151] Julien Pettré, Pablo de Heras Ciechomski, Jonathan Maïm, Barbara Yersin, Jean-Paul Laumond, and Daniel Thalmann. Real-time navigating crowds : scalable simulation and rendering. *Computer Animation and Virtual Worlds*, 17(3-4) :445–455, 2006.

[152] Petr Pospichal, Jiri Jaros, and Josef Schwarz. Parallel genetic algorithm on the cuda architecture. In *Applications of Evolutionary Computation*, pages 442–451. Springer, 2010.

[153] Raghavendra D Prabhu. Gneuron : parallel neural networks with gpu. In *Posters, International Conference on High Performance Computing (HiPC)*. Citeseer, 2007.

[154] Aristid Lindenmayer Przemyslaw Prusinkiewicz, Aristid Lindenmayer, James S Hanan, F David Fracchia, and Deborah Fowler. The algorithmic beauty of plants with. 1990.

[155] Kari Pulli, Anatoly Baksheev, Kirill Kornyakov, and Victor Eruhimov. Real-time computer vision with opencv. *Communications of the ACM*, 55(6) :61–69, 2012.

[156] AK Qin, Federico Raimondo, Florence Forbes, and Yew Soon Ong. An improved cuda-based implementation of differential evolution on gpu. In *Proceedings of the 14th annual conference on Genetic and evolutionary computation*, pages 991–998. ACM, 2012.

[157] Colin R Reeves. *Modern heuristic techniques for combinatorial problems*. John Wiley & Sons, Inc., 1993.

[158] J Reggia, M Tagamets, J Contreras-Vidal, D Jacobs, S Weems, W Naqvi, R Winder, T Chabuk, J Jung, and C Yang. Development of a large-scale integrated neurocognitive architecture-part 2 : Design and architecture. 2006.

[159] James Neal Richter et al. On mutation and crossover in the theory of evolutionary algorithms. 2010.

[160] Raúl Rojas. *Neural networks : a systematic introduction*. Springer Science & Business Media, 1996.

[161] Günter Rudolph. Convergence analysis of canonical genetic algorithms. *Neural Networks, IEEE Transactions on*, 5(1) :96–101, 1994.

[162] Günter Rudolph. Finite markov chain results in evolutionary computation : a tour d'horizon. *Fundamenta Informaticae*, 35(1) :67–89, 1998.

[163] John K Salmon, Mark Moraes, Ron O Dror, David E Shaw, et al. Parallel random numbers : as easy as 1, 2, 3. In *High Performance Computing, Networking, Storage and Analysis (SC), 2011 International Conference for*, pages 1–12. IEEE, 2011.

[164] Ernesto Sanchez, Massimiliano Schillaci, and Giovanni Squillero. *Evolutionary Optimization : the μGP toolkit : The UGP Toolkit*. Springer Science & Business Media, 2011.

[165] Pedro V Sander and Jason L Mitchell. Progressive buffers : view-dependent geometry and texture lod rendering. In *ACM SIGGRAPH 2006 Courses*, pages 1–18. ACM, 2006.

[166] Nadathur Satish, Mark Harris, and Michael Garland. Designing efficient sorting algorithms for manycore gpus. In *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pages 1–10. IEEE, 2009.

[167] John E Savage. Models of computation. *Exploring the Power of Computing*, 1998.

[168] Michael C Schatz, Cole Trapnell, Arthur L Delcher, and Amitabh Varshney. High-throughput sequence alignment using graphics processing units. *BMC bioinformatics*, 8(1) :474, 2007.

[169] Dominik Scherer, Hannes Schulz, and Sven Behnke. Accelerating large-scale convolutional neural networks with parallel graphics multiprocessors. In *Artificial Neural Networks–ICANN 2010*, pages 82–91. Springer, 2010.

[170] Olena Schuessler and Diego Loyola. Parallel training of artificial neural networks using multithreaded and multicore cpus. In *Adaptive and Natural Computing Algorithms*, pages 70–79. Springer, 2011.

[171] Rajvi Shah, P Narayanan, and Kishore Kothapalli. Gpu-accelerated genetic algorithms. *cvit. iiit. ac. in*, 2010.

[172] Minghui Shi, Wei Pan, Hugo de Garis, and K Chen. Approach to controlling robot by artificial brain based on parallel evolutionary neural network. In *Industrial Mechatronics and Automation (ICIMA), 2010 2nd International Conference on*, volume 2, pages 502–505. IEEE, 2010.

[173] Jaewoong Sim, Aniruddha Dasgupta, Hyesoon Kim, and Richard Vuduc. A performance analysis framework for identifying potential benefits in gpgpu applications. In *ACM SIGPLAN Notices*, volume 47, pages 11–22. ACM, 2012.

[174] Sudipta N Sinha, Jan-Michael Frahm, Marc Pollefeys, and Yakup Genc. Gpu-based video feature tracking and matching. In *EDGE, Workshop on Edge Computing Using New Commodity Architectures*, volume 278, page 4321, 2006.

[175] Zbigniew Skolicki. An analysis of island models in evolutionary computation. In *Proceedings of the 7th annual workshop on Genetic and evolutionary computation*, pages 386–389. ACM, 2005.

[176] Nicolás Soca, José Luis Blengio, Martin Pedemonte, and Pablo Ezzatti. Pugace, a cellular evolutionary algorithm framework on gpus. In *Evolutionary Computation (CEC), 2010 IEEE Congress on*, pages 1–8. IEEE, 2010.

[177] DA Sofge, Mitchell A Potter, Magdalena D Bugajska, and Alan C Schultz. Challenges and opportunities of evolutionary robotics. *arXiv preprint arXiv :0706.0457*, 2007.

[178] Tobias Storch and Ingo Wegener. Real royal road functions for constant population size. In *Genetic and Evolutionary ComputationâGECCO 2003*, pages 1406–1417. Springer, 2003.

[179] Omar Syed. *Applying genetic algorithms to recurrent neural networks for learning network parameters and architecture*. PhD thesis, Case Western Reserve University, 1995.

[180] El-Ghazali Talbi. *Metaheuristics : from design to implementation*, volume 74. John Wiley & Sons, 2009.

[181] Zhe Tang and Meng Joo Er. Humanoid 3d gait generation based on inverted pendulum model. In *Intelligent Control, 2007. ISIC 2007. IEEE 22nd International Symposium on*, pages 339–344. IEEE, 2007.

[182] Jun Tani and Stefano Nolfi. Learning to perceive the world as articulated : an approach for hierarchical learning in sensory-motor systems. *Neural Networks*, 12(7) :1131–1141, 1999.

[183] Aram Ter-Sarkisov. *Computational complexity of elitist population-based evolutionary algorithms : a thesis presented in partial fulfilment of the requirements for the degree of Doctor of Philosophy in Computer Science at Massey University, Palmerston North, New Zealand*. PhD thesis, 2012.

[184] Rahul Thota, Sharan Vaswani, Amit Kale, and Nagavijayalakshmi Vydyanathan. Fast 3d salient region detection in medical images using gpus. *arXiv preprint arXiv :1310.6736*, 2013.

[185] Emanuel Todorov, Tom Erez, and Yuval Tassa. Mujoco : A physics engine for model-based control. In *Intelligent Robots and Systems (IROS), 2012 IEEE/RSJ International Conference on*, pages 5026–5033. IEEE, 2012.

[186] Yuri Torres, Arturo Gonzalez-Escribano, and Diego R Llanos. Understanding the impact of cuda tuning techniques for fermi. In *High Performance Computing and Simulation (HPCS), 2011 International Conference on*, pages 631–639. IEEE, 2011.

[187] Yuri Torres, Arturo Gonzalez-Escribano, and Diego R Llanos. Measuring the impact of configuration parameters in cuda through benchmarking. In *The 12th International Conference Computational and Mathematical Methods in Science and Engineering, CMMSE*, volume 2012, 2012.

[188] Shigeyoshi Tsutsui and Pierre Collet. *Massively Parallel Evolutionary Computation on GPGPUs*. Springer, 2013.

[189] Peter JM Van Laarhoven and Emile HL Aarts. *Simulated annealing*. Springer, 1987.

[190] The Van Luong, Nouredine Melab, and E-G Talbi. Gpu computing for parallel local search metaheuristic algorithms. *Computers, IEEE Transactions on*, 62(1) :173–185, 2013.

[191] Gregory Vorobyev, Andrew Vardy, and Wolfgang Banzhaf. Supervised learning in robotic swarms : From training samples to emergent behavior. In *Distributed Autonomous Robotic Systems*, pages 435–448. Springer, 2014.

[192] Shouyi Wang, Wanpracha Chaovalitwongse, and Robert Babuska. Machine learning algorithms in bipedal robot control. *Systems, Man, and Cybernetics, Part C : Applications and Reviews, IEEE Transactions on*, 42(5) :728–743, 2012.

[193] Ingo Wegener. *Methods for the analysis of evolutionary algorithms on pseudo-boolean functions*. Springer, 2002.

[194] Karsten Weicker and Nicole Weicker. Basic principles for understanding evolutionary algorithms. *Fundamenta Informaticae*, 55(3) :387–403, 2003.

[195] Thomas Weise. Global optimization algorithms-theory and application. *Self-Published,*, 2009.

[196] J Teahan William. Directions for bio-inspired artificial intelligence. *Journal of Computer Science & Systems Biology*, 2012.

[197] Carsten Witt. Population size vs. runtime of a simple ea. In *Evolutionary Computation, 2003. CEC'03. The 2003 Congress on*, volume 3, pages 1996–2003. IEEE, 2003.

[198] Carsten Witt. Runtime analysis of the (μ+ 1) ea on simple pseudo-boolean functions. *Evolutionary Computation*, 14(1) :65–86, 2006.

[199] David H Wolpert and William G Macready. No free lunch theorems for optimization. *Evolutionary Computation, IEEE Transactions on*, 1(1) :67–82, 1997.

[200] Ka-Chun Wong, Chengbin Peng, Man-Hon Wong, and Kwong-Sak Leung. Generalizing and learning protein-dna binding sequence representations by an evolutionary algorithm. *Soft Computing*, 15(8) :1631–1642, 2011.

[201] Man-Leung Wong and Tien-Tsin Wong. Parallel hybrid genetic algorithms on consumer-level graphics hardware. In *Evolutionary Computation, 2006. CEC 2006. IEEE Congress on*, pages 2973–2980. IEEE, 2006.

[202] Man-Leung Wong, Tien-Tsin Wong, and Ka-Ling Fok. Parallel evolutionary algorithms on graphics processing unit. In *Evolutionary Computation, 2005. The 2005 IEEE Congress on*, volume 3, pages 2286–2293. IEEE, 2005.

[203] Li Xu. Robotstudio : a universal ide for teaching undergraduate computer system courses. *Journal of Computing Sciences in Colleges*, 22(6) :65–72, 2007.

[204] GN Xuan and RW Cheng. Genetic algorithms and engineering optimization. *Tsinghua University Press, Beijing, China (in Chinese). 0*, 20 :40–60, 2004.

[205] Yi Yang, Ping Xiang, Jingfei Kong, Mike Mantor, and Huiyang Zhou. A unified optimizing compiler framework for different gpgpu architectures. *ACM Transactions on Architecture and Code Optimization (TACO)*, 9(2) :9, 2012.

[206] Yang Yu and Zhi-Hua Zhou. A new approach to estimating the expected first hitting time of evolutionary algorithms. *Artificial Intelligence*, 172(15) :1809–1832, 2008.

[207] Marcelo PM Zamith, Esteban WG Clua, Aura Conci, Anselmo Montenegro, Regina CP Leal-Toledo, Paulo A Pagliosa, Luis Valente, and Bruno Feij. A game loop architecture for the gpu used as a math coprocessor in real-time applications. *Computers in Entertainment (CIE)*, 6(3) :42, 2008.

[208] Jun Zhang, Henry Chung, and Wai-Lun Lo. Pseudocoevolutionary genetic algorithms for power electronic circuits optimization. *Systems, Man, and Cybernetics, Part C : Applications and Reviews, IEEE Transactions on*, 36(4) :590–598, 2006.

[209] Yao Zhang and John D Owens. A quantitative performance analysis model for gpu architectures. In *High Performance Computer Architecture (HPCA), 2011 IEEE 17th International Symposium on*, pages 382–393. IEEE, 2011.

[210] Jian Feng Zhao, Wen Hua Zeng, Guang Ming Li, and Min Liu. Simple parallel genetic algorithm using cloud computing. *Applied Mechanics and Materials*, 121 :4151–4155, 2012.

[211] Long Zheng, Yanchao Lu, Mengwei Ding, Yao Shen, Minyi Guoz, and Minyi Guo. Architecture-based performance evaluation of genetic algorithms on multi/many-core systems. In *Computational Science and Engineering (CSE), 2011 IEEE 14th International Conference on*, pages 321–334. IEEE, 2011.

[212] Yuren Zhou, Jun Zhang, and Yong Wang. Performance analysis of the (1+ 1) evolutionary algorithm for the multiprocessor scheduling problem. *Algorithmica*, pages 1–21, 2014.

[213] Weihang Zhu. A study of parallel evolution strategy : pattern search on a gpu computing platform. In *Proceedings of the first ACM/SIGEVO Summit on Genetic and Evolutionary Computation*, pages 765–772. ACM, 2009.

[214] Wenjun Zhuang, Foo Hanyang, Shen Zhaoxuan, and Divya Rajesh. Hpc application in dsm/vdsm ic chip planning. In *High Performance Computing in the Asia-Pacific Region, 2000. Proceedings. The Fourth International Conference/Exhibition on*, volume 2, pages 1125–1131. IEEE, 2000.