View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by INRIA a CCSD electronic archive server

Algorithmes évolutionnaires parallèles sur GPU

Thé Van Luong, Nouredine Melab, El-Ghazali Talbi

To cite this version:

Thé Van Luong, Nouredine Melab, El-Ghazali Talbi. Algorithmes évolutionnaires parallèles sur GPU. Manifestation des Jeunes Chercheurs en Sciences et Technologies de l'Information et de la Communication (Majecstic), 2010, Bordeaux, France. inria-00520471

HAL Id: inria-00520471
https://hal.inria.fr/inria-00520471

Submitted on 23 Sep 2010

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

MajecSTIC 2010
Bordeaux, France, du 13 au 15 octobre 2010

Algorithmes évolutionnaires parallèles sur GPU

Thé Van Luong, Nouredine Melab et El-Ghazali Talbi

Université de Lille 1, INRIA Lille Nord Europe, 40 avenue Halley, 59650 Villeneuve d'Ascq - France.

Contact : `The-Van.Luong@inria.fr`, `Nouredine.Melab@lifl.fr`, `El-Ghazali.Talbi@lifl.fr`

Résumé
Les algorithmes d'optimisation tels que les algorithmes évolutionnaires sont des méthodes efficaces pour résoudre des problèmes complexes en sciences et en industrie. Même si ces heuristiques permettent de réduire de manière significative le temps de calcul de l'exploration de l'espace de recherche d'une solution, ce dernier coût reste exorbitant lorsque de très grandes instances d'un problème sont résolues. Ainsi, l'utilisation du calcul parallèle à base de GPU est requise comme une façon complémentaire d'accélérer la recherche. Dans ce papier, on se concentra ainsi sur leur reconception, leur implémentation et les difficultés associées relatifs au contexte d'exécution du GPU. Les résultats expérimentaux obtenus démontrent l'efficacité des approches proposées et leur capacité d'exploiter pleinement l'architecture du GPU.

Mots-clés : Métaheuristiques, algorithmes évolutionnaires, GPU

1. Introduction

Les problèmes d'optimisation réels sont souvent complexes et NP-difficiles, leur modélisation continue d'évoluer en termes de contraintes et d'objectifs, et leur résolution est coûteuse en temps de calcul CPU. Bien que des algorithmes presque optimaux tels que les métaheuristiques (heuristiques génériques) permettent de réduire la complexité de leur résolution, elles restent insuffisantes pour résoudre des problèmes de grande taille. Le calcul sur GPU est reconnu de nos jours comme une manière puissante de réaliser du calcul haute performance sur des applications scientifiques importantes [7]. Notre challenge réside sur la conception de métaheuristiques sur GPU afin d'obtenir le maximum d'efficacité pour résoudre des problèmes complexes de grande taille. Les métaheuristiques sont fondées sur l'amélioration itérative de soit une population de solutions (p. ex. les algorithmes évolutionnaires ou AEs) ou bien d'une seule solution (p. ex. la recherche tabou) d'un problème d'optimisation donné. Dans ce papier, on ce concentrera sur la première catégorie c.-à-d. les AEs.

Depuis plusieurs années, l'utilisation de processeurs graphiques était dédiée aux applications graphiques. Récemment, leur utilisation a été étendue à d'autres domaines d'application (p. ex. aux sciences informatiques) grâce à la publication du kit de développement CUDA qui permet de programmer sur GPU dans un langage proche du C. Dans certains domaines tels que le calcul numérique, nous sommes maintenant témoin de la prolifération de bibliothèques logiciels tels que CUBLAS pour GPU. Cependant, dans d'autres domaines telle que l'optimisation combinatoire, en particulier les métaheuristiques, l'arrivée du GPU ne connait pas la même croissance. En effet, il n'existe qu'essentiellement que des travaux liés à la programmation génétique [2,4,9] qui est très spécifique à un type de problème. Avec l'arrivée d'OpenCL comme le standard de langage de programmation sur GPU et l'arrivée de futurs compilateurs pour ce langage, comme d'autres domaines d'application, l'optimisation combinatoire sur GPU générera un intérêt grandissant.

Néanmoins, l'utilisation du calcul parallèle sur GPU pour les métaheuristiques n'est pas évidente. En effet, on est confronté à plusieurs challenges scientifiques principalement liés à la gestion de la mémoire hiérarchique. Les difficultés majeures sont la distribution efficace du traitement de données entre le CPU et le GPU, la synchronisation des threads, l'optimisation des transferts de données entre les différentes mémoires, les contraintes de capacité de ces mémoires, etc. L'objectif

de ce papier est de faire face à ces difficultés pour la re-conception des modèles parallèles des AEs pour permettre la résolution de problèmes à grande échelle sur les architectures GPU. Dans ce papier, nous contribuons avec l'entière reconception des AEs sur GPU en spécifiant ces différents paramètres en prenant compte les caractéristiques liés à la fois au processus de l'AE et au calcul sur GPU.
Le reste du papier est organisé comme suit : La section 2 met en avant les principes des AEs et leurs modèles parallèles associés. Dans la section 3, différents schémas de parallélisation pour la conception d'AEs parallèles sur GPU sont présentés. Pour valider les approches proposées dans le papier, la section 4 décrit les résultats de performances obtenus pour un problème d'optimisation continu. Et enfin, une discussion et quelques conclusions de ce travail sont faites dans la section 5.

2. Algorithmes évolutionnaires parallèles

2.1. Principes des algorithmes évolutionnaires

Les AEs sont des techniques de recherche stochastique qui ont été appliqués avec succès pour résoudre plein de problèmes réels et complexes. Un AE est une technique itérative qui applique des opérateurs stochastiques sur un pool d'individu (la population). Chaque individu de la population représente la version encodé d'une tentative de solution. Au départ, cette population est générée aléatoirement. À chaque génération de l'algorithme, les solutions sont sélectionnées, regroupées en pair et recombinées dans le but de générer de nouvelles solutions qui remplaceront les plus mauvais selon un certain critère, et ainsi de suite. Une fonction d'évaluation associe une valeur de fitness à chaque individu pour indiquer sa pertinence par rapport au problème (critère de sélection).

Algorithme 1 : Pseudo-code d'un algorithme évolutionnaire

1: Initialiser($P(0)$) ;
2: $t := 0$;
3: **répéter**
4: Evaluer($P(t)$) ;
5: $P'(t)$:= Sélectionner($P(t)$) ;
6: $P'(t)$:= Appliquer_opérateurs_reproduction($P'(t)$) ;
7: $P(t+1)$:= Remplacer(($P(t)$,$P'(t)$)) ;
8: $t := t+1$;
9: **jusqu'à** Critère_terminaison($P(t)$)

L'algorithme 1 montre les composants génétiques de n'importe quel AE. Il existe plusieurs sous-classes des AEs bien connues dépendant de la représentation des individus ou de comment chaque étape de l'algorithme est conçue. Les principales sous-classes des AEs sont les algorithmes génétiques, la programmation génétique, les stratégies d'évolution, etc. Une revue complète de ces sous-classes est donné dans [8].

2.2. Modèles parallèles

Pour des problèmes difficiles, exécuter le cycle de reproduction d'un simple AE sur des larges individus et/ou sur de grandes populations requiert beaucoup de ressources en termes de calcul. Par conséquent, une variété de difficultés algorithmiques doit être étudiée pour concevoir des AEs efficaces. Ces difficultés consistent habituellement à définir de nouveaux opérateurs, des algorithmes hybrides, des modèles parallèles, etc. Ainsi, le parallélisme arrive naturellement lorsqu'on a affaire à des populations puisque chacun des individus peut être considéré comme une unité indépendante. Plus de détails sur les paradigmes parallèles sont proposés dans [1, 8] pour les AEs. De manière simple, trois modèles parallèles majeurs peuvent être distingués :

- **Le modèle d'évaluation parallèle d'une solution**. Ce modèle est particulièrement intéressant lorsque la fonction d'évaluation peut être elle-même parallélisée dans la mesure où l'évaluation

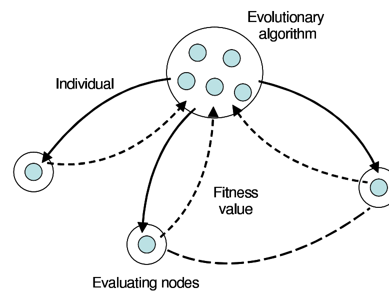

FIGURE 1 – L'évaluation parallèle de la population.

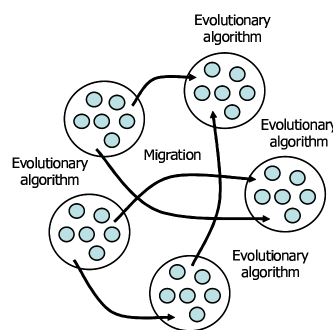

FIGURE 2 – Le modèle en îles coopératifs d'algorithmes évolutionnaires.

serait coûteuse en tant de calcul et/ou en entrées/sorties. Dans ce cas, la fonction peut être vue comme une agrégation d'un certain nombre de fonctions partielles.
- **Le modèle d'évaluation parallèle de la population**. Ce modèle arrive naturellement puisque l'évaluation de la population est souvent la partie la plus coûteuse en temps de calcul (voir FIGURE 1). L'évaluation parallèle suit le modèle Fermier/Travailleurs. À chaque génération, l'ensemble des nouvelles solutions est distribué entre les différents travailleurs. Puis ces solutions sont évaluées et leur résultat est retourné au fermier. Une exécution efficace est souvent obtenue particulièrement lorsque l'évaluation de chaque solution est coûteuse.
- **Le modèle en îles coopératif (a)synchrone**. Différents AEs sont simultanément exécutés pour coopérer afin de calculer des solutions meilleures et robustes (voir FIGURE 2). Les îles échangent de manière (a)synchrone de l'information génétique afin de diversifier la recherche. L'objectif étant de permettre de retarder la convergence globale, plus particulièrement lorsque les opérateurs de variation sont de nature hétérogène. La migration des individus suit une certaine politique définie par quelques paramètres : le critère de décision de migration, la topologie d'échanges, le nombre d'émigrants, la politique de sélection d'émigrants et la politique de remplacement/intégration.

Puisque le modèle d'évaluation parallèle d'une seule solution ne présentant que peu de concepts génériques, on se concentra ainsi exclusivement dans ce papier sur les deux autres modèles.

3. Algorithmes évolutionnaires sur GPU

Avec les avancées récentes sur le calcul parallèle fondé sur le GPU, les modèles parallèles pour les AEs doivent être revisités du point de vue de la conception et de l'implémentation. Pour accomplir cela, nous proposons différents schémas permettant une séparation claire des concepts de gestion de la mémoire hiérarchique du GPU.

3.1. Le GPU

Conduit par la demande pour des graphiques en 3D à haute définition sur des ordinateurs personnels, les GPUs ont évolués en un environnement fortement parallèle, multi-threadés et multi-cœurs. En effet, cette architecture fournit une puissance de calcul importante et une très grande bande passante mémoire en comparaison avec les CPUs traditionnels. Puisque plus de transistors sont dévouées 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 complète revue sur l'architecture du GPU peut être trouvé dans [7].

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 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. Dans les AEs, le type de données qui est manipulé est la représentation de la population. Le transfert de mémoire du CPU à la mémoire périphérique du GPU est une opération synchrone qui est coûteuse en temps de calcul. Dans le cas des AEs, les opérations de copie du CPU vers le GPU sont essentiellement des opérations de duplication de la population de solutions. Le bus de bande passante et la latence entre le CPU et GPU peut significativement diminuer les performances de la recherche. Ainsi ces transferts de données doivent être minimisés.

Chaque processeur du GPU supporte le modèle un seul programme pour des données multiples (SPMD), c.-à-d. de multiple processeurs exécutent simultanément le même programme sur différentes données. Pour accomplir cela, le notion de kernel est défini. Le kernel est une fonction appelable à partir de l'hôte et est exécuté par plusieurs processeurs en parallèle sur le périphérique désigné. Cette gestion du noyau est dépendante du langage de programmation sur GPU utilisé. Par exemple, CUDA et OpenCL sont des environnements de calcul parallèle qui introduisent une interface de programmation d'applications. Ces kits de développement introduisent un modèle de threads qui fournissent une abstraction simple pour des architectures simple instruction et données multiples (SIMD). Un thread sur GPU peut être vu comme un élément de donnée à être traité. Comparé aux threads sur CPU, les threads sur GPU sont légers. Ce qui signifie que changer le contexte parmi deux threads n'est pas une opération coûteuse.

Concernant leur organisation spatiale, 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 ou deux 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. Par conséquent, en ce qui concerne les AEs, une affectation naturelle est d'associer un thread sur GPU avec un individu de la population.

3.2. Évaluation parallèle de la population sur GPU : Modèle Fermier/Travailleur

Un premier modèle naturel pour la conception et la l'implémentation d'un AE sur GPU est le modèle d'évaluation parallèle. En effet, en général, évaluer la fonction d'évaluation pour chaque individu est souvent l'opération la plus coûteuse pour un AE. Ainsi, pour ce modèle, la distribution des tâches est clairement définie : le CPU gère tout le processus évolutionnaire séquentiel et le GPU est dédié seulement à l'évaluation parallèle des solutions. La FIGURE 3 montre ce principe d'évaluation parallèle sur GPU.

Pour commencer, le CPU envoie un certain nombre d'individus à évaluer au GPU à travers la mémoire globale et ensuite ces solutions sont traitées par le GPU. Concernant l'organisation des threads dans le kernel, comme mentionné auparavant, le GPU est organisé selon le modèle SPMD, ce qui signifie que de multiples processeurs autonomes exécutent simultanément le même programme à des points indépendants. Par conséquent, chaque thread GPU associé avec un individu exécute le même kernel c.-à-d. la même fonction d'évaluation. Finalement, les résultats de la fonction d'évaluation sont retournés vers l'hôte à travers la mémoire globale.

De cette façon, le GPU est utilisé comme un coprocesseur d'une manière synchrone. La partie la plus coûteuse en temps de calcul c.-à-d. le kernel d'évaluation est calculée par le GPU et le reste est traité par le CPU. Cependant, selon la taille de la population, le désavantage de ce schéma de parallélisation est que les opérations de copies du CPU vers le GPU (c.-à-d. la population et les structures associées) peuvent devenir fréquentes et donc elles peuvent mener à une baisse des

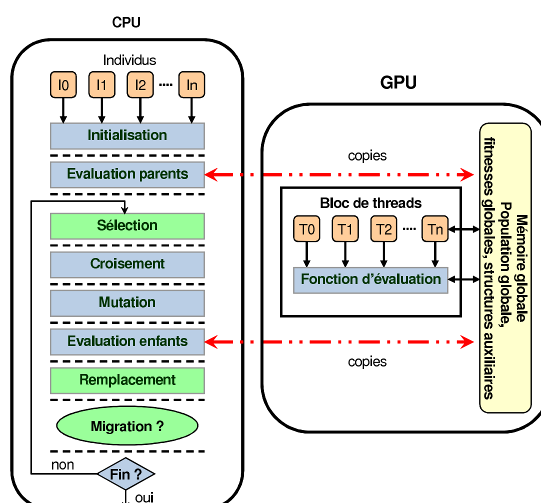

FIGURE 3 – Evaluation parallèle de la population sur GPU.

performances significatives. Le but de schéma de parallélisation est d'accélérer le processus de recherche et il n'altère en aucun cas la sémantique de l'algorithme.

3.3. Modèle en îles sur GPU : Modèle coopératif parallèle

3.3.1. Distribution complète du modèle en île sur GPU

Concernant le MI, une approche naturelle est de paralléliser tout l'algorithme sur GPU. De cetta manière, l'avantage de cette approche est de minimiser les transferts de données entre la mémoire hôte et le GPU. La FIGURE 4 donne une illustration de ce concept. Dans ce schéma, une représentation naturelle est d'associer une île avec un bloc de threads. Un individu est ainsi représenté par un thread et chaque opérateur génétique standard (p. ex. la sélection le croisement ou la mutation) est séparé par des barrières de bloc pour permettre la synchronisation entre les threads. Concernant la politique de migration, les communications sont effectuées au travers de la mémoire globale qui stocke la population globale. De ce manière, chaque île locale peut communiquer avec les autres selon une certaine topologie.
Une des limitations à déplacer l'algorithme en entier sur GPU est le fait qu'une stratégie hétérogène ne peut être facilement appliquée. En effet, puisque les multiprocesseurs sur GPU sont utilisés selon le modèle SPMD, les mêmes paramètres de configuration et les mêmes composants de recherche (p. ex. la mutation ou le croisement) entre les îles doivent être utilisés. Un autre désavantage de ce schéma concerne le nombre maximal d'individus par île puisque ce dernier est limité au nombre maximal de threads par bloc (jusqu'à 512 ou 1024 selon l'architecture GPU). Une idée naturelle pour résoudre cette limitation serait d'associer une île avec plusieurs blocs de threads. Cependant, cela ne peut pas être achevé en pratique puisque (1) les threads travaillent d'une manière asynchrone ; (2) les synchronisations entre les threads sont locales à un même bloc. En effet, on peut imaginer un scénario dans lequel une sélection est faite sur deux individus situés dans des blocs différents où l'un d'un deux threads n'a pas encore mis à jour sa valeur de fitness associée (c.-à-d. qu'un des deux individus n'a pas encore été évalué).

3.3.2. Distribution complète du modèle en îles sur GPU avec utilisation de la mémoire partagée

Concernant la gestion de la mémoire du kernel, d'un point de vue du matériel, les cartes graphiques sont constituées de multiprocesseur, chacun avec des unités de traitement, des registres et de la mémoire embarquée. Accéder à la mémoire globale implique 400 à 600 cycles additionnels de latence mémoire. Par conséquent, puisque cette mémoire n'est pas en cache et son accès est lente, on a besoin de minimiser les accès à la mémoire globale (opérations de lecture/écriture) et réutiliser les données à l'intérieur des mémoires locales aux multiprocesseurs. Pour accomplir cela, la mémoire partagée est une mémoire rapide embarquée localisée sur les multiprocesseurs et partagée par les threads d'un même bloc. Cette mémoire peut être considérée comme un caché

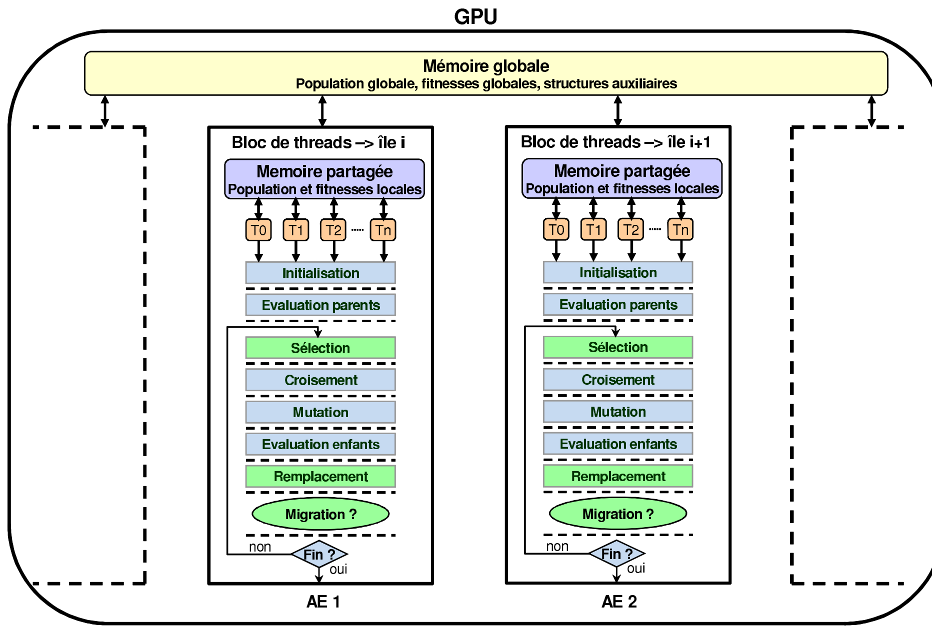

FIGURE 4 – Distribution complète du modèle en îles sur GPU en utilisant la mémoire partagée.

géré par l'utilisateur qui peut produire de fortes accélérations en économisant la bande passante de la mémoire principale [6]. De plus, puisque la mémoire partagée est locale à chaque bloc de threads, cela fournit un moyen pour les threads de communiquer à l'intérieur de chaque bloc.
Ainsi, un autre schéma de parallélisation est d'associer chaque île à un bloc de threads sur GPU en utilisant la mémoire partagée (voir détails FIGURE 4). Ce schéma est similaire au précédent excepté que les populations locales et leur fitnesses associées sont stockées dans la mémoire partagée embarquée. De cette façon, chaque individu (thread) sur chaque île (bloc) effectue le processus évolutionnaire (initialisation, évaluation, etc.) à travers la mémoire partagée.
En ce qui concerne la migration entre les îles, du fait que la migration requiert une communication entre les îles, les opérations de copie de chaque population locale (mémoire partagée) vers la population globale (mémoire globale) doivent être considérées. Même si ce schéma de parallélisation peut améliorer l'efficacité du MI, il présente une limitation majeure : puisque chaque multiprocesseur a une capacité limité de mémoire partagée (16KB), seuls les problèmes de petite taille peuvent être traitées. En effet, la quantité de mémoire partagée allouée par bloc dépend à la fois de la taille de la population local et la taille du problème. Ainsi, un compromis entre le nombre de threads par bloc et la taille du problème traité doit être considéré.

4. Experimentation

Pour valider les différentes approches proposées dans ce papier, les fonctions de Weierstrass-Mandelbrot ont été considérées sur GPU. Ces fonctions appartiennent à la classe des problèmes d'optimisation continue. Selon [5], les fonctions de Weierstrass-Mandelbrot sont définis comme suit :
T

$$W_{b,h}(x) = \sum_{i=1}^{\infty} b^{-ih} \sin(b^i x) \quad \text{with } b > 1 \text{ and } 0 < h < 1 \tag{1}$$

Le paramètre h a un impact sur l'irrégularité (perturbation locale bruitée d'amplitude limitée) et ces fonctions possèdent plusieurs optimaux locaux. Du fait que le calcul d'une somme infinie de sinus est impraticable, un nombre d'itérations est utilisé pour calculer une approximation de la fonction (au lieu de ∞). Plus cette valeur est grande, plus l'évaluation d'un individu prendra du temps. Un autre paramètre qui peut être réglé pour ces fonctions est la dimension du pro-

blème. Ainsi, un problème de dimension 10 prendra plus de temps à évaluer qu'un problème de dimension 2.
Pour ce problème, différentes versions du MI ont été implémentées sur GPU en utilisant CUDA. La configuration utilisée pour les expérimentations est un Intex Xeon 2.4 Ghz avec une carte graphique NVIDIA GTX 280 (30 multi-processeurs). Pour chaque expérience, différentes implémentations sont données : un seul cœur de calcul CPU pour le MI sur (CPU), le MI synchrone combinant l'évaluation parallèle de la population sur GPU (CPU+GPU), le MI asynchrone complètement distribué sur GPU (GPU), la version synchrone (SGPU) et leur versions associées utilisant la mémoire partagée (GPUPart et SGPUPart).
Concernant la fonction d'évaluation du problème (c.-à-d. la fonction Weierstrass (1)), le domaine de définition a été choisis entre $-1 \leq x_k \leq 1$, h a été fixé à $0,25$ et le nombre d'itérations pour calculer l'approximation à 100 (au lieu d'∞). La complexité de la fonction d'évaluation est quadratique. Les opérateurs utilisés dans le processus évolutionnaire pour les implémentations sont les suivants : le croisement est un croisement standard en deux points (taux de croisement fixé à 80%) qui génère deux enfants, la mutation consiste à changer aléatoire un gène avec une valeur réelles prise entre $[-1; 1]$ (avec un taux de mutation de 30%), la sélection est représenté par un tournoi déterministe (la taille du tournoi est fixé à la taille d'un bloc divisé par 4), le remplacement est un remplacement $(\mu + \lambda)$ et le nombre de générations a été fixée a 100. Concernant la politique de migration, une topologie en anneau a été choisie, un tournoi déterministe a été effectué pour à la fois la sélection d'émigrants et le remplacement lors de la migration (la taille du tournoi est fixée à la taille d'un bloc divisé par 4), le taux de migration est égal au nombre d'individus local divisé par 4 et la fréquence de migration a été fixée à 10 générations.
L'objective des expériences suivantes est d'évaluer l'impact d'une implémentation à base de GPU en terme d'efficacité. Seuls les temps d'exécution (en secondes) et les facteurs d'accélérations (comparé à un seul cœur CPU sans le GPU) sont décrits. Le temps moyen a été mesuré en secondes four 50 exécutions. L'écart-type associé n'étant pas représenté car sa valeur est petite pour chaque valeur mesurée.
La première expérience consiste à varier la dimension de la fonction Weierstrass. Les résultats sont décrits à la TABLE 1(a). Au fur à mesure que la taille de la dimension augmente, chaque version GPU donne des accélérations impressionnantes comparés à une version CPU (jusqu'à $\times 1757$ pour la version GPUPart). L'utilisation de la mémoire partagée fournit une manière efficace d'accélérer le processus de recherche même si la version GPU de base est déjà impressionnante. Cependant, du fait de sa capacité limitée, de plus grandes instances comme la dimension de taille 11 ne peuvent être traitées dans aucune des versions utilisant la mémoire partagée.
Concernant les versions synchrones complètement distribuées, puisque des synchronisations implicites sont faites, une certaine baisse des performances (de $\times 63$ à $\times 293$ pour la version SGPU) peut être observée en comparaison avec leur version asynchrone associée. Néanmoins, les facteurs d'accélérations sont quand même impressionnants. Pour le schéma utilisant l'évaluation parallèle de la population (version CPU+GPU), les accélérations sont moins importants même s'ils restent significatifs (de $\times 25$ à $\times 170$). Ce qui peu être expliqué par le nombre important de transferts de données entre le CPU et le GPU
Une conclusion de cette première expérience indique que changer de dimension du problème mène à de meilleures accélérations pour chaque version GPU. Cela semble évident que l'augmentation du nombre d'itérations pour calculer une approximation de la fonction Weierstrass mènerait des résultats similaires. Une deuxième expérience consiste à varier le nombre d'îles dans le but de mesure l'efficacité et la scalabilité de nos approches. Les résultats de cette expérience sont détaillés dans la TABLE 1(b). En ce qui concerne chaque version complètement distribuée, pour un petit nombre d'îles (c.-à-d. une ou deux îles), le facteur d'accélération est significatif mais pas impressionnant (de $\times 7$ à $\times 51$). Cela peut s'expliquer par le fait que puisque la population globale est relativement petite (moins de 1024 threads), le nombre de threads par block n'est pas suffisant pour couvrir pleinement latence liée au accès mémoire. Cependant, l'accélération croît en accord avec l'augmentation du nombre d'îles et reste impressionnant (jusqu'à $\times 2074$ pour GPUPart).
Pour la version CPU+GPU, les accélérations pour une ou deux îles sont plus importantes que celles des versions complètement distribuées. En effet, puisque seule l'évaluation de la population est distribuée sur GPU, moins de registres sont alloués pour chaque thread. De ce fait, cette

TABLE 1 – La TABLE (a) consiste à faire varier la taille de l'instance du problème : le nombre d'individus par île est fixé à 128 et celle de la population globale à 8192 (64 îles). La TABLE (b) consiste à faire varier le nombre d'îles : la dimension du problème est fixée à 2 et le nombre d'individus par îles à 128.

(a)	**CPU**	**CPU+GPU**		**GPU**		**GPUPart**		**SGPU**		**SGPUPart**	
dimension	**temps**	**temps**	**acc.**	**temps**	**acc.**	**temps**	**acc.**	**temps**	**acc.**	**temps**	**acc.**
1	23	0.92	×25	0.16	×143	0.04	×845	0.36	×63	0.06	×375
2	43	0.94	×46	0.16	×268	0.03	×1150	0.36	×119	0.08	×511
3	64	0.95	×67	0.17	×375	0.05	×1365	0.38	×167	0.11	×607
4	85	0.97	×87	0.21	×403	0.06	×1442	0.47	×179	0.13	×641
5	105	1.00	×105	0.22	×479	0.07	×1579	0.50	×213	0.15	×702
6	127	1.02	×125	0.24	×519	0.08	×1639	0.55	×231	0.17	×728
7	148	1.04	×142	0.28	×529	0.09	×1659	0.63	×235	0.20	×737
8	168	1.09	×154	0.30	×554	0.10	×1684	0.68	×246	0.23	×748
9	190	1.19	×159	0.31	×610	0.11	×1736	0.70	×271	0.25	×772
10	211	1.28	×165	0.33	×639	0.12	×1757	0.74	×284	0.27	×781
11	231	1.36	×170	0.35	×666	–	–	0.79	×293	–	–

(b)	**CPU**	**CPU+GPU**		**GPU**		**GPUPart**		**SGPU**		**SGPUPart**	
islands	**temps**	**temps**	**acc.**	**temps**	**acc.**	**temps**	**acc.**	**temps**	**acc.**	**temps**	**acc.**
1	3	0.10	×33	0.20	×17	0.12	×27	0.45	×7	0.27	×12
2	7	0.12	×55	0.20	×33	0.13	×51	0.45	×15	0.29	×23
4	13	0.15	×89	0.20	×65	0.13	×104	0.45	×29	0.29	×46
8	26	0.19	×139	0.20	×132	0.13	×207	0.45	×59	0.29	×92
16	53	0.34	×154	0.21	×256	0.13	×403	0.46	×114	0.29	×179
32	106	0.66	×160	0.26	×406	0.13	×828	0.59	×180	0.29	×368
64	211	1.28	×165	0.33	×644	0.14	×1560	0.74	×286	0.30	×693
128	422	2.68	×158	0.45	×939	0.26	×1596	1.01	×417	0.60	×709
256	845	5.61	×151	0.69	×1222	0.50	×1677	1.56	×543	1.13	×746
512	1692	11.81	×143	1.24	×1365	1.00	×1691	2.79	×607	2.25	×752
1024	3382	25.72	×132	–	–	1.70	×1990	–	–	3.82	×885
2048	6781	53.23	×127	–	–	3.27	×2074	–	–	7.36	×922
4096	13585	143.71	×95	–	–	–	–	–	–	–	–

version bénéficie d'un meilleur taux d'occupation des multiprocesseurs pour un petit nombre d'îles. Le GPU continue d'accélérer le processus avec l'augmentation des îles jusqu'à atteindre un pic de performance de ×165 pour 64 îles. Après cela, le facteur d'accélération décroît avec l'augmentation du nombre d'îles. En effet, pour chaque évaluation parallèle de la population, la quantité de données transférées est proportionnelle au nombre d'individus (p. ex. 52488 threads pour 4096 îles). C'est ainsi, que pour un certain nombre d'îles, le temps consacré pour les opérations de copie devient significatif menant à une baisse des performances.

En ce qui concerne la scalabilité des versions complètement distribuées, à partir d'un certain nombre d'îles, le GPU a échoué à exécuter le programme du fait de la limite matériel des registres. Par exemple, pour un nombre de 1024 îles (131072 threads), l'implémentation SGPU n'a pu être exécutée. Dans les versions GPUPart et SGPUPart, puisque la mémoire partagée est utilisée menant ainsi à moins de registres, cette limite est atteinte pour un plus grand nombre de 4096 îles. Pour la version CPU+GPU, on obtient une plus grande scalabilité du fait que beaucoup moins de registres sont alloués (seul le kernel d'évaluation étant exécuté sur GPU).

5. Conclusion and discussion

Les métaheuristiques parallèles permettent d'améliorer l'efficacité et la robustesse des problèmes d'optimisation. Leur exploitation pour résoudre des problèmes réels est possible seulement en utilisant une grande puissance de calcul. Le calcul à haute performance à base d'accélérateurs GPU a été récemment révélé comme un moyen efficace d'utilise la grande quantité de ressources à disposition. Cependant, l'exploitation des modèles parallèles n'est pas trivial et plusieurs difficultés liées au contexte d'exécution de cette architecture doivent être considérés.

Dans ce papier, nous nous sommes particulièrement concentrés sur la conception de différents schémas de parallélisation sur le GPU hiérarchique. Les approches conçues et implémentées ont été expérimentalement validées sur un problème d'optimisation continu. Les expériences indique

que le GPU se révèle être efficace pour les AEs (accélération jusqu'à $\times 2000$). Un premier schéma de parallélisation consiste à l'évaluation parallèle de la population sur GPU selon le modèle Fermier/Travailleur. D'un point de vue de l'implémentation, cette approche est la plus générique car seul le kernel représentant la fonction d'évaluation est considéré sur GPU. Mais les perfomances de ce schéma sont limités du fait des nombreux transferts de données entre le CPU et le GPU. Concernant le MI, nous avons aussi proposé deux autres schémas de parallélisation basé sur le modèle coopératif parallèle. Pour réduire les transferts de données, une distribution complète du processus de recherche est effectuée sur GPU. Appliquer un tel mécanisme permet d'améliorer de manière considérablement les performances de l'algorithme. Cependant, ces schémas peuvent présenter quelques limitations à cause des contraites mémoires pour des problèmes qui pourraient être plus demandant en termes de ressources. Néanmoins, nous croyons fortement que ces schémas de parallélisation pourrait être facilement étendus à de problèmes de très grande taille en optimisation continu.

Une autre perspective est de tester les différents schémas de parallélisation à des problèmes d'optimisation combinatoire tel que le problème de l'affectation quadratique. Cependant, à la différence des fonctions continues, les données en entrée d'un problème d'optimisation combinatoire doivent être considérées. En effet, en travaillant avec de telles structures, les accès non-alignés à la mémoire globale impliquent plus de transactions mémoire menant à une perte globale de performance. Il en découle que traiter avec de tels problèmes d'optimisation n'est pas évident puisque cela implique d'optimiser l'accès aux données ce qui inclut l'utilisation appropriée des variétés d'espaces mémoire du GPU.

Dans le futur, ces concepts sur le GPU seront intégrés dans la plateforme ParadisEO. Cette librairie a été développée pour la conception de métaheuristiques parallèles hybrides dédiées à la résolution mono/multiobjective [3]. ParadisEO peut être vu comme une boîte blanche orientée objet fondée sur la claire séparation des concepts des métaheuristiques. Le module Parallel Evolving Objects de ParadisEO inclue déjà les modèles parallèles des métaheuristiques de la littérature. Ce module sera étendyu dans le futur avec une implémentation à base de GPU.

Bibliographie

1. Enrique Alba et Marco Tomassini. Parallelism and evolutionary algorithms. *IEEE Trans. Evolutionary Computation*, 6(5) :443–462, 2002.
2. Wolfgang Banzhaf, Simon Harding, William B. Langdon, et Garnett Wilson. Accelerating genetic programming through graphics processing units. In *Genetic Programming Theory and Practice VI*, pages 1–19. Springer, 2009.
3. Sébastien Cahon, Nordine Melab, et El-Ghazali Talbi. Paradiseo : A framework for the reusable design of parallel and distributed metaheuristics. *J. Heuristics*, 10(3) :357–380, 2004.
4. W. Langdon et Wolfgang Banzhaf. A SIMD interpreter for genetic programming on GPU graphics cards. In *Genetic Programming*, pages 73–85. 2008.
5. Evelyne Lutton et Jacques Lévy Véhel. Holder functions and deception of genetic algorithms. *IEEE Trans. Evolutionary Computation*, 2(2) :56–71, 1998.
6. NVIDIA. *CUDA Programming Guide Version 2.3*, 2010.
7. Shane Ryoo, Christopher I. Rodrigues, Sam S. Stone, John A. Stratton, Sain-Zee Ueng, Sara S. Baghsorkhi, et W. W. Hwu. Program optimization carving for gpu computing. *J. Parallel Distribributed Computing*, 68(10) :1389–1401, 2008.
8. El-Ghazali Talbi. *Metaheuristics : From design to implementation*. Wiley, 2009.
9. Tien-Tsin Wong et Man Leung Wong. Parallel evolutionary algorithms on consumer-level graphics processing unit. In *Parallel Evolutionary Computations*, pages 133–155. Springer, 2006.