Université de Reims Champagne-Ardenne
École Doctorale Sciences Technologies et Santé

Thèse

présentée par **Audrey DELÉVACQ**

pour l'obtention du grade de

Docteur de l'Université de Reims Champagne-Ardenne

Spécialité : Informatique

Métaheuristiques pour l'optimisation combinatoire sur processeurs graphiques (GPU)

Soutenue publiquement le 04 Février 2013 devant le jury composé de :

Président	Alain BUI	Professeur à l'Université de Versailles St-Quentin-en-Yvelines
Rapporteur	Laetitia JOURDAN	Professeur à l'Université de Lille 1
Rapporteur	Van-Dat CUNG	Professeur à l'École d'ingénieurs Grenoble INP
Examinateur	Gilles DEQUEN	Maître de Conférence HDR à l'Université de Picardie Jules Verne
Directeur	Michaël KRAJECKI	Professeur à l'Université de Reims Champagne-Ardenne
Co-encadrant	Pierre DELISLE	Maître de Conférence à l'Université de Reims Champagne-Ardenne

Centre de Recherche en STIC, équipe Systèmes Communicants

"I always start new work without any detailed designs. My hope is for a piece to complete itself beyond my imagination. I sometimes say that artists are parallel to scientists. Scientists have no way of knowing the result of an experiment beforehand. The same can be said about creating art", Yosuke Ueno

Remerciements

En premier lieu, je tiens à exprimer toute ma reconnaissance à monsieur Michaël Krajecki, Professeur à l'Université de Reims Champagne-Ardenne, pour avoir dirigé mon travail pendant mes années de thèse. Je le remercie chaleureusement pour son encadrement, son oeil critique dans la correction de mon manuscrit et pour son soutien constant tout au long de ma thèse.

Mes remerciements vont conjointement à mon co-encadrant de thèse Pierre Delisle, Maître de Conférence à l'Université de Reims Champagne-Ardenne, pour sa gentillesse et pour toute l'aide qu'il m'a apportée au cours de ce travail. Son intérêt, sa disponibilité, son suivi quotidien de mes travaux et ses précieux conseils m'ont été d'un grand profit. Je le remercie infiniment.

Je remercie mes rapporteurs madame Laetita Jourdan, Professeur à l'Université de Lille 1 et monsieur Van-Dat Cung, Professeur à l'École d'ingénieurs Grenoble INP, pour avoir rapporté mes travaux et pour les questions constructives qu'ils ont soulevées.

Merci également à monsieur Alain Bui, Professeur à l'Université de Versailles St-Quentin-en-Yvelines, et à monsieur Gilles Dequen, Maître de Conférence HDR à l'Université de Picardie Jules Verne, pour avoir accepté de participer à mon jury de thèse et pour l'intérêt qu'ils ont porté à mes travaux.

Je remercie aussi monsieur Marc Gravel et madame Caroline Gagné, Professeurs à l'Université du Québec à Chicoutimi, pour leur accueil chaleureux lors de mon séjour au Canada et pour leurs précieuses remarques lors de la rédaction des articles ou lors des communications.

J'adresse mes remerciements à toute l'équipe du laboratoire CReSTIC : les enseignants - chercheurs, les techniciens et mes collègues doctorants pour tous les échanges techniques et scientifiques mais également pour leur sympathie pendant ces trois ans de thèse.

Je mentionne également le support financier de la Région Champagne-Ardenne pour la réalisation de ma thèse et le support technique du Centre de Calcul de Champagne-Ardenne ROMEO pour la disponibilité des ressources de calcul utilisées dans le cadre de ces travaux. Ce travail a également été soutenu par l'Agence Nationale de la Recherche (ANR) n° ANR-2010-COSI-003-03.

Merci à mes amies Priscille, Sarah, Nathalie et Charlotte pour leur patience.

Finalement, merci à ma famille pour leur soutien et leurs encouragements : mes parents Christine et Guy, mon frère Loïc et mes grands-parents Josette et Michel.

Résumé

Plusieurs problèmes d'optimisation combinatoire sont dits $\mathcal{NP}$-*difficiles* et ne peuvent être résolus de façon optimale par des algorithmes exacts que pour des instances de petite taille. Les métaheuristiques ont prouvé qu'elles pouvaient être efficaces pour résoudre un grand nombre de ces problèmes en leur trouvant des solutions approchées en un temps raisonnable. Cependant, face à des instances de grande taille, elles ont besoin d'un temps de calcul et d'une quantité d'espace mémoire considérables pour être performantes dans l'exploration de l'espace de recherche. Par conséquent, l'intérêt voué à leur déploiement sur des architectures de calcul haute performance a augmenté durant ces dernières années. Les approches de parallélisation existantes suivent généralement les paradigmes de passage de messages ou de mémoire partagée qui conviennent aux architectures traditionnelles à base de microprocesseurs, aussi appelés CPU (*Central Processing Unit*). Cependant, la recherche évolue très rapidement dans le domaine du parallélisme et de nouvelles architectures émergent, notamment les accélérateurs matériels qui permettent de décharger le CPU de certaines de ses tâches. Parmi ceux-ci, les processeurs graphiques ou GPU (*Graphics Processing Units*) présentent une architecture massivement parallèle possédant un grand potentiel mais aussi de nouvelles difficultés d'algorithmique et de programmation. En effet, les modèles de parallélisation de métaheuristiques existants sont généralement inadaptés aux environnements de calcul de type GPU. Certains travaux ont d'ailleurs abordé ce sujet sans toutefois y apporter une vision globale et fondamentale.

L'objectif général de cette thèse est de proposer un cadre de référence permettant l'implémentation efficace des métaheuristiques sur des architectures parallèles basées sur les GPU. Elle débute par un état de l'art décrivant les travaux existants sur la parallélisation GPU des métaheuristiques et les classifications générales des métaheuristiques parallèles. Une taxonomie originale est ensuite proposée afin de classifier les implémentations recensées et de formaliser les stratégies de parallélisation sur GPU dans un cadre méthodologique cohérent. Cette thèse vise également à valider cette taxonomie en exploitant ses principales composantes pour proposer des stratégies de parallélisation originales spécifiquement adaptées aux architectures GPU. Plusieurs implémentations performantes basées sur les métaheuristiques d'Optimisation par Colonie de Fourmis et de Recherche Locale Itérée sont ainsi proposées pour la résolution du problème du Voyageur de Commerce. Une étude expérimentale structurée et minutieuse est réalisée afin d'évaluer et de comparer la performance des approches autant au niveau de la qualité des solutions trouvées que de la réduction du temps de calcul.

Mots-clés : optimisation combinatoire, problèmes $\mathcal{NP}$-*difficiles*, problème du Voyageur de Commerce, métaheuristiques, Optimisation par Colonie de Fourmis, Recherche Locale, Recherche Locale Itérée, parallélisme, GPU.

Abstract

Several combinatorial optimization problems are $\mathcal{NP}$-*hard* and can only be solved optimally by exact algorithms for small instances. Metaheuristics have proved to be effective in solving many of these problems by finding approximate solutions in a reasonable time. However, dealing with large instances, they may require considerable computation time and amount of memory space to be efficient in the exploration of the search space. Therefore, the interest devoted to their deployment on high performance computing architectures has increased over the past years. Existing parallelization approaches generally follow the message-passing and shared-memory computing paradigms which are suitable for traditional architectures based on microprocessors, also called CPU (*Central Processing Unit*). However, research in the field of parallel computing is rapidly evolving and new architectures emerge, including hardware accelerators which offloads the CPU of some of its tasks. Among them, graphics processors or GPUs (*Graphics Processing Units*) have a massively parallel architecture with great potential but also imply new algorithmic and programming challenges. In fact, existing parallelization models of metaheuristics are generally unsuited to computing environments like GPUs. Few works have tackled this subject without providing a comprehensive and fundamental view of it.

The general purpose of this thesis is to propose a framework for the effective implementation of metaheuristics on parallel architectures based on GPUs. It begins with a state of the art describing existing works on GPU parallelization of metaheuristics and general classifications of parallel metaheuristics. An original taxonomy is then designed to classify identified implementations and to formalize GPU parallelization strategies in a coherent methodological framework. This thesis also aims to validate this taxonomy by exploiting its main components to propose original parallelization strategies specifically tailored to GPU architectures. Several effective implementations based on Ant Colony Optimization and Iterated Local Search metaheuristics are thus proposed for solving the Travelling Salesman Problem. A structured and thorough experimental study is conducted to evaluate and compare the performance of approaches on criteria related to solution quality and computing time reduction.

Keywords : combinatorial optimization, $\mathcal{NP}$-*hard* problems, Travelling Salesman Problem, metaheuristics, Ant Colony Optimization, Local Search, Iterated Local Search, parallel computing, GPU.

Table des matières

Remerciements i

Résumé iii

Table des Matières vii

Liste des Figures ix

Liste des Tableaux x

Liste des Algorithmes xiii

1 Introduction 1

2 Métaheuristiques parallèles sur GPU 5
- 2.1 Architecture GPU . . . 7
- 2.2 Métaheuristiques à solution unique . . . 13
  - 2.2.1 Recherche Locale . . . 13
  - 2.2.2 Recherche avec Tabous . . . 16
  - 2.2.3 Recuit Simulé . . . 18
- 2.3 Métaheuristiques à population de solutions . . . 21
  - 2.3.1 Algorithmes Évolutionnaires . . . 21
  - 2.3.2 Optimisation par Essaims Particulaires . . . 26
  - 2.3.3 Optimisation par Colonie de Fourmis . . . 29
- 2.4 Classification des métaheuristiques parallèles . . . 33
  - 2.4.1 Stratégies $1C$ de bas niveau . . . 34
  - 2.4.2 Stratégies basées sur une décomposition du domaine . . . 36
  - 2.4.3 Stratégies reposant sur de multiples recherches . . . 37
- 2.5 Objectifs de la recherche . . . 38

3 Taxonomie des métaheuristiques parallèles sur GPU 41
- 3.1 Première dimension : élément de calcul . . . 41
  - 3.1.1 Stratégies de parallélisation . . . 45
  - 3.1.2 Stratégies de parallélisation hybrides . . . 46
- 3.2 Deuxième dimension : mémoire . . . 49
- 3.3 Conclusion . . . 52

4 Stratégies de parallélisation GPU pour l'Optimisation par Colonie de Fourmis 53
4.1 Problème du Voyageur de Commerce . . . 54
4.2 Optimisation par Colonie de Fourmis . . . 55
4.3 MAX-MIN Ant System . . . 56
4.4 Stratégies de parallélisation GPU . . . 59
4.4.1 Stratégies de parallélisation pour l'approche $FOURMI$ . . . 59
4.4.2 Stratégies de parallélisation pour l'approche $COLONIE$ . . . 63
4.5 Résultats expérimentaux . . . 65
4.5.1 Approche $FOURMI$ . . . 66
4.5.1.1 Sans listes de candidats . . . 66
4.5.1.2 Avec listes de candidats . . . 71
4.5.2 Approche $COLONIE$ . . . 74
4.5.2.1 Stratégie $COLONIE_{bloc}$ . . . 74
4.5.2.2 Stratégie $COLONIE_{gpu}$ . . . 76
4.6 Conclusion . . . 78

5 Stratégies de parallélisation GPU pour la Recherche Locale Itérée 81
5.1 Recherche Locale . . . 82
5.2 Recherche Locale Itérée . . . 86
5.3 Stratégies de parallélisation GPU . . . 88
5.3.1 Stratégies de parallélisation pour l'approche $RLI - FI$ . . . 89
5.3.2 Stratégies de parallélisation pour l'approche $RLI - RKBI$ . . . 92
5.4 Résultats expérimentaux . . . 96
5.4.1 Approche $RLI - FI$ . . . 96
5.4.1.1 Qualité des solutions de l'approche séquentielle . . . 97
5.4.1.2 Qualité des solutions et accélérations des implémentations parallèles . . . 98
5.4.1.3 Variation du nombre de candidats . . . 104
5.4.2 Approche $RLI - RKBI$ . . . 105
5.5 Conclusion . . . 111

6 Conclusion et perspectives 113

Liste des publications 116

Bibliographie 117

Liste des Figures

2.1 Architectures d'un CPU multicœur et d'un GPU. 8
2.2 Schéma de l'architecture d'un GPU. 8
2.3 Schéma du modèle de programmation CUDA. 10
2.4 Création des warps. 10
2.5 Divergence de branchement au sein d'un warp. 11
2.6 *CUDA Occupancy Calculator*. 12
2.7 Nombre de publications sur la parallélisation des métaheuristiques sur GPU. . . 32
2.8 Trois dimensions de la classification de Crainic [Cra05]. 33
2.9 Classification des algorithmes de Recherche Locale parallèles proposée par Verhoeven et Aarts [VA95]. 35
2.10 Classification des algorithmes d'Optimisation par Colonie de Fourmis parallèles proposée par Pedemonte *et al.* [PNC11]. 36

3.1 Niveaux de parallélisation du GPU et du modèle de programmation CUDA. . . 42
3.2 Niveaux de parallélisation des métaheuristiques. 42
3.3 Taxonomie des métaheuristiques parallèles sur GPU. 43
3.4 Types de mémoires GPU et types de données des métaheuristiques. 49

4.1 Problème du Voyageur de Commerce. 54
4.2 Problème du VC symétrique (à gauche) et asymétrique (à droite). 55
4.3 Déplacement des fourmis. 56
4.4 Stratégie de parallélisation générale de l'approche $FOURMI$. 61
4.5 Stratégie de parallélisation $FOURMI_{thread}$. 61
4.6 Stratégie de parallélisation $FOURMI_{bloc}$. 62
4.7 Stratégie de parallélisation $COLONIE_{bloc}$. 63
4.8 Stratégie de parallélisation $COLONIE_{gpu}$. 64
4.9 Accélérations des stratégies de parallélisation du MMAS. 70
4.10 Accélérations des stratégies de parallélisation du MMAS avec candidats. 73
4.11 Accélérations de la stratégie $COLONIE_{bloc}$ obtenues avec 1 à 500 colonies. . . 75
4.12 Accélérations de la stratégie $COLONIE_{gpu}$ obtenues avec 1 et 2 colonies. . . 77

5.1 Exemple de transformations locales. 82
5.2 Mouvements 3-opt : 7 façons de reconnecter les tournées partielles quand 3 arêtes sont supprimées. 83
5.3 Recherche de voisinage à rayon fixe pour le 2-opt. 85

5.4 Recherche de voisinage à rayon fixe pour le 3-opt. 85
5.5 Mouvement double-bridge. 87
5.6 Stratégie de parallélisation générale de l'approche $RLI - FI$. 89
5.7 Stratégie de parallélisation $RLI - FI_{thread}$. 91
5.8 Stratégie de parallélisation $RLI - FI_{bloc}$. 91
5.9 Stratégie de parallélisation générale de l'approche $RLI - RKBI$. 93
5.10 Stratégie de parallélisation générale $RLI - RKBI_{blocs}$. 94
5.11 Accélérations obtenues par les stratégies de parallélisation pour les différents problèmes et les différentes valeurs de nb_{sol}. 100
5.12 Accélérations obtenues pour les stratégies $RLI-RKBI_{blocs}$ (rkbi) et $RLI-FI_{bloc}$ (fi) pour chaque problème selon nb_{sol}. 109

Liste des Tableaux

2.1 Caractéristiques typiques des différentes mémoires d'un GPU moderne. 9

3.1 Classification des stratégies de parallélisation GPU utilisées dans la littérature selon la dimension "élément de calcul" (partie 1). 45
3.2 Classification des stratégies de parallélisation GPU utilisées dans la littérature selon la dimension "élément de calcul" (partie 2). 46
3.3 Classification des stratégies de parallélisation GPU hybrides utilisées dans la littérature selon la dimension "élément de calcul" (partie 1). 47
3.4 Classification des stratégies de parallélisation GPU hybrides utilisées dans la littérature selon la dimension "élément de calcul" (partie 2). 48
3.5 Classification des stratégies de parallélisation GPU selon les dimensions "élément de calcul" et "mémoire" (partie 1). 50
3.6 Classification des stratégies de parallélisation GPU selon les dimensions "élément de calcul" et "mémoire" (partie 2). 51

4.1 Nombre de fourmis m pour chaque problème. 67
4.2 Accélérations pour la stratégie $FOURMI_{thread}$ (configurations blocs/threads). . 67
4.3 Accélérations obtenues et nombre de blocs actifs (entre parenthèses) pour la stratégie $FOURMI_{bloc}^{globale}$ en faisant varier le nombre de threads. 68
4.4 Accélérations pour la stratégie $FOURMI_{bloc}^{partagée}$ (configurations blocs/threads). 69
4.5 Longueur minimale et moyenne des solutions obtenues et pourcentage de déviation moyen (entre parenthèses) du MMAS séquentiel proposé par rapport à la version originale de Stützle et Hoos [SH97]. 69
4.6 Longueur minimale et moyenne des solutions obtenues et pourcentage de déviation (entre parenthèses) des versions parallèles par rapport à la version séquentielle. 70
4.7 Temps séquentiel de l'algorithme MMAS en secondes. 70
4.8 Longueur minimale et moyenne des solutions obtenues et pourcentage de déviation moyen (entre parenthèses) du MMAS séquentiel proposé par rapport à celui présenté par Stützle et Hoos [SH00]. 71
4.9 Longueur minimale et moyenne des solutions obtenues et pourcentage de déviation (entre parenthèses) des versions parallèles par rapport à la version séquentielle. 72
4.10 Temps séquentiel en secondes de l'algorithme MMAS avec candidats. 73
4.11 Longueur minimale et moyenne des solutions obtenues par la version séquentielle de l'algorithme MMAS et la stratégie de parallélisation $COLONIE_{bloc}$. 75

4.12 Pourcentage de déviation minimum Δ_{min}^{seq} et moyen Δ_{moy}^{seq} de la stratégie de parallélisation $COLONIE_{bloc}$ par rapport à la version séquentielle de l'algorithme MMAS. 76
4.13 Longueur minimale et moyenne des solutions obtenues et pourcentage de déviation (entre parenthèses) de la stratégie $COLONIE_{gpu}$ par rapport à la version séquentielle. 77

5.1 Temps d'exécution t_{25000} pour $it_{25000} = 25000$ itérations et limites de temps t_{lim} fournies par Stützle et Hoos permettant d'estimer le nombre maximum moyen d'itérations it_{lim}^{a}. Nombre maximum d'itérations it_{lim}^{b} utilisé dans les travaux de Lourenço *et al.* 97
5.2 Fréquence d'obtention de l'optimum f_{opt}, pourcentage moyen de déviation par rapport à l'optimum Δ_{moy}, temps moyen t_{moy} et nombre moyen d'itérations it_{moy} pour trouver la meilleure solution d'un essai et temps total d'exécution t_{total} obtenus avec l'implémentation proposée (Del.) en comparaison des résultats fournis par Stützle et Hoos (Stü.). 97
5.3 Nombre maximum d'itérations it_{lim} utilisées et pourcentage moyen de déviation par rapport à l'optimum Δ_{moy} obtenus avec l'implémentation proposée (Del.) en comparaison des résultats de Lourenço *et al.* (Lou.). Pourcentage moyen de déviation Δ_{moy}^{Lou} par rapport aux résultats de Lourenço *et al.*. 98
5.4 Temps séquentiel de l'algorithme pour chaque problème et chaque valeur de nb_{sol}. 99
5.5 Configurations du nombre de blocs et du nombre de threads par bloc pour la stratégie $RLI - FI_{thread}$. 99
5.6 Nombre d'essais effectués selon la taille du problème, nb_{sol} et la stratégie de parallélisation. 99
5.7 Fréquence d'obtention de la solution optimale connue f_{opt} et pourcentage moyen de déviation par rapport à l'optimum Δ_{moy} pour chaque problème et chaque valeur de nb_{sol} pour les versions séquentielle et parallèles. Moyenne des pourcentages moyens de déviation Δ_{moy}^{seq} par rapport à la version séquentielle (partie 1). 101
5.8 Fréquence d'obtention de la solution optimale connue f_{opt} et pourcentage moyen de déviation par rapport à l'optimum Δ_{moy} pour chaque problème et chaque valeur de nb_{sol} pour les versions séquentielle et parallèles. Moyenne des pourcentages moyens de déviation Δ_{moy}^{seq} par rapport à la version séquentielle (partie 2). 102
5.9 Fréquence d'obtention de la solution optimale connue f_{opt} et du pourcentage moyen de déviation par rapport à l'optimum Δ_{moy} pour chaque problème et chaque valeur de nb_{sol} pour les versions séquentielle et parallèles. Moyenne des pourcentages moyens de déviation Δ_{moy}^{seq} par rapport à la version séquentielle (partie 3). 103
5.10 Temps séquentiel et parallèle, accélération et qualité des solutions (fréquence d'obtention de la solution optimale connue f_{opt}, pourcentage de déviation moyen par rapport à l'optimum Δ_{moy} et pourcentage de déviation moyen Δ_{moy}^{40} de la version utilisant 64 candidats par rapport celle utilisant 40 candidats) pour la stratégie $RLI - FI_{bloc}^{globale}$ avec $nb_{sol} = 2048$. 105

5.11 Nombre de blocs par solution selon le problème et le nombre de voisins k. 106
5.12 Nombre de voisins k et d'itérations it_{total}, temps séquentiel, temps parallèle et accélérations pour la stratégie $RLI-FI_{bloc}^{globale}$ (FI) et la stratégie $RLI-RKBI_{blocs}$ (RKBI) pour chaque problème et pour $nb_{sol} = 64$. L'accélération FI_m correspond à la meilleure accélération obtenue par problème quelque soit nb_{sol} et quelque soit la stratégie de l'approche $RLI - FI$. 107
5.13 Nombre de voisins k et d'itérations it_{total}, fréquence d'obtention de l'optimum f_{opt} et pourcentage moyen de déviation par rapport à l'optimum Δ_{moy} pour les stratégies $RLI - FI_{bloc}^{globale}$ (FI) et $RLI - RKBI_{blocs}$ (RKBI) avec $nb_{sol} = 64$. Pourcentage de déviation moyen Δ_{moy}^{FI} de la stratégie $RLI - RKBI_{blocs}$ par rapport à la stratégie $RLI - FI_{bloc}^{globale}$. 108
5.14 Exemple de classement par rang pour le problème de 5915 villes. 109
5.15 Nombre de blocs par solution pour les différents problèmes et valeurs de nb_{sol}. . 109
5.16 Temps séquentiel et qualité de solution (fréquence d'obtention de la solution optimale connue f_{opt} et pourcentage de déviation par rapport à l'optimum Δ_{moy}). 110

Liste des Algorithmes

2.1 Recherche Locale 13
2.2 Recherche avec Tabous 17
2.3 Recuit Simulé 19
2.4 Algorithme Évolutionnaire 22
2.5 Optimisation par Essaims Particulaires 27
2.6 Optimisation par Colonie de Fourmis 30
4.1 Algorithme du MAX-MIN Ant System. 57
4.2 Pseudo-code de l'approche générale $FOURMI$. 60
4.3 Pseudo-code de la stratégie $COLONIE_{bloc}$. 64
4.4 Pseudo-code de la stratégie $COLONIE_{gpu}$. 65
5.1 Recherche Locale 3-opt 83
5.2 Recherche Locale Itérée 87
5.3 Pseudo-code de l'approche générale $RLI - FI$. 90
5.4 Recherche Locale 3-opt avec règle de pivotement random-k-best-improvement. . 93
5.5 Pseudo-code de la stratégie $RLI - RKBI_{blocs}$ (partie Recherche Locale). 95

Chapitre 1

Introduction

L'optimisation combinatoire cherche à résoudre divers problèmes pouvant être rencontrés dans de nombreux domaines académiques et industriels. La plupart de ces problèmes sont considérés comme étant $\mathcal{NP}$-*difficiles* et ne peuvent être résolus de façon optimale par des algorithmes exacts que pour des instances de petite taille et ce, peu importe la technologie utilisée. Il est donc nécessaire de trouver un compromis entre le temps de résolution du problème et la qualité des solutions obtenues. Les méthodes approchées représentent une alternative aux méthodes exactes en fournissant une solution réalisable à ces problèmes en un temps raisonnable.

Les métaheuristiques sont des méthodes approchées qui explorent efficacement l'espace de recherche d'un problème d'optimisation combinatoire. Elles possèdent généralement des mécanismes évitant l'emprisonnement dans des régions non prometteuses de l'espace de recherche et favorisant l'atteinte des optima globaux. Depuis les deux dernières décennies, une quantité considérable de travaux ont été consacrés à l'amélioration de leurs performances. Néanmoins, face à des problèmes de taille et de complexité toujours croissantes, les métaheuristiques ont souvent encore besoin d'un temps de calcul et d'une quantité d'espace mémoire considérables pour être efficaces. De ce fait, l'utilisation du calcul parallèle devient de plus en plus privilégiée pour réduire leur temps de calcul et améliorer la qualité des solutions qu'elles permettent d'obtenir.

Les approches de parallélisation existantes suivent généralement les paradigmes de passage de messages ou de mémoire partagée qui conviennent aux architectures traditionnelles à base de microprocesseurs, aussi appelés CPU (*Central Processing Unit*). Cependant, la recherche évolue rapidement dans le domaine du parallélisme et de nouvelles architectures émergent, notamment les accélérateurs matériels qui permettent de décharger le CPU de certains de ses calculs. Parmi ceux-ci, les processeurs graphiques ou GPU (*Graphics Processing Units*) sont typiquement utilisés pour réaliser le traitement d'images. Ils présentent une architecture massivement parallèle qui contient plusieurs multiprocesseurs composés eux-mêmes d'un ensemble de processeurs. Ils comprennent également plusieurs mémoires qui diffèrent par leur taille, leur temps de latence et leur type d'accès. Les GPU fournissent donc un grand potentiel de calcul à un coût abordable, mais aussi de nouvelles difficultés d'algorithmique et de programmation. En effet, les modèles de parallélisation existants pour les métaheuristiques sont généralement inadaptés aux environnements de calcul basés sur les accélérateurs matériels comme les GPU et les travaux apportant une vision globale et fondamentale à ce problème sont encore manquants.

L'objectif général de cette thèse est de proposer un cadre de référence permettant l'implémentation efficace des métaheuristiques sur des architectures parallèles basées sur les GPU. Une taxonomie originale est élaborée pour formaliser et structurer l'ensemble des approches de parallélisation dans un cadre méthodologique cohérent. À partir de celle-ci, des stratégies innovantes sont proposées et implémentées dans un environnement de calcul haute performance de dernière génération. Afin de démontrer leur validité et leur généricité, elles sont développées dans des contextes de résolution par métaheuristiques à solution unique, métaheuristiques à population de solutions et méthodes de Recherche Locale. Une étude expérimentale approfondie est finalement réalisée pour évaluer la performance des implémentations sur des critères de qualité des solutions obtenues et de réduction des temps de calcul.

Ce mémoire est organisé en six chapitres.

Dans le Chapitre 2, une revue de la littérature sur les métaheuristiques parallèles sur GPU est présentée. Une première partie de ce chapitre est consacrée à l'architecture GPU et au modèle de programmation CUDA. Elle décrit les caractéristiques spécifiques de cet environnement de calcul comme les différentes mémoires à gérer et les granularités de parallélisation multiples à exploiter. Une deuxième partie porte sur la description des principales métaheuristiques et sur la revue des travaux de parallélisation GPU existants pour ces méthodes. Finalement, la troisième partie présente les travaux de référence sur les classifications qui apportent une vue globale et fondamentale aux métaheuristiques parallèles. Quelques travaux pertinents de parallélisation CPU de métaheuristiques sont également soulignés.

Le Chapitre 3 propose une taxonomie des métaheuristiques parallèles sur GPU basée sur deux dimensions. La première associe les composantes de la métaheuristique aux différents niveaux de parallélisme offerts par la structure des éléments de calcul du GPU. La seconde spécifie le type de mémoire du GPU utilisée pour stocker les différentes données nécessaires au fonctionnement de la métaheuristique. À partir de ces dimensions, plusieurs stratégies de parallélisation sont ensuite formalisées et mises en relation avec les différentes approches retrouvées dans la littérature.

Les Chapitres 4 et 5 présentent la démarche de validation de cette taxonomie dans un contexte appliqué de conception et de développement de méthodes d'optimisation parallèles compétitives. Plus spécifiquement, plusieurs stratégies de parallélisation de métaheuristiques à population de solutions et à solution unique sont appliquées à la résolution du problème du Voyageur de Commerce. Une étude expérimentale détaillée permet d'évaluer et de comparer la performance des stratégies sur les critères de qualité des solutions obtenues et de réduction du temps de calcul. Ces deux critères sont d'ailleurs toujours mis en opposition en suivant le principe qu'une méthode d'optimisation s'exécutant efficacement sur une architecture parallèle n'a d'intérêt que si elle demeure compétitive dans l'optimisation du problème traité. Dans cette optique, un effort particulier est mis en œuvre afin de confronter les résultats obtenus à ceux des implémentations séquentielles originales définies dans la littérature de référence.

Le Chapitre 4 est consacré à la parallélisation d'une métaheuristique à population de solutions : l'Optimisation par Colonie de Fourmis. Plusieurs implémentations parallèles performantes de l'algorithme *MAX-MIN Ant System* sont proposées où la construction des tournées est déportée sur le GPU. Chaque fourmi est associée à un élément de calcul et les tournées sont ainsi construites en parallèle. Ces implémentations se distinguent par leur type d'approche : $FOURMI$ et $COLONIE$. Dans le premier cas, une seule colonie est utilisée. Les stratégies

consistent alors à associer la notion d'élément de calcul à un processeur ou à un multiprocesseur, avec la possibilité de stocker les données des fourmis dans la mémoire rapide et de petite taille du GPU. Dans le second cas, plusieurs colonies évoluent en parallèle et sont associées aux GPU ou aux multiprocesseurs. L'influence de la répartition des processus de recherche est tout d'abord étudiée. Le comportement des stratégies de parallélisation proposées est ensuite analysé selon différents critères liés à la qualité des solutions et à la réduction du temps de calcul. Cette étude comparative souligne l'impact sur la performance des paramètres définissant le comportement de l'Optimisation par Colonie de Fourmis, des configurations techniques du GPU, des structures de mémoire et de la granularité de la parallélisation.

Le Chapitre 5 est consacré à la parallélisation d'une métaheuristique à solution unique : la Recherche Locale Itérée. Les implémentations parallèles proposées reposent sur de multiples recherches et se distinguent par leur type d'approche : $RLI - FI$ et $RLI - RKBI$. Dans le premier cas, la règle de pivotement *first-improvement* est utilisée. La méthodologie de parallélisation appliquée à l'Optimisation par Colonie de Fourmis est transposée à la Recherche Locale Itérée démontrant la généricité du cadre proposé. Les stratégies consistent alors à associer chaque recherche à un processeur ou à un multiprocesseur, avec la possibilité de stocker les données des solutions dans la mémoire rapide de petite taille du GPU. Dans le second cas, une stratégie est spécialement conçue pour tirer avantage des capacités du GPU dans l'application des transformations locales. Elle est basée sur une règle de pivotement, nommée *random-k-best-improvement*, qui offre un compromis entre les règles *first-improvement* et *best-improvement*. Cette stratégie hybride permet ainsi d'associer une recherche à plusieurs multiprocesseurs. Le comportement des stratégies de parallélisation proposées est ensuite analysé et évalué expérimentalement sur les critères combinés de qualité des solutions trouvées et de réduction du temps de calcul. Cette étude comparative souligne l'influence de la règle de pivotement, de la taille du voisinage et de la granularité de la parallélisation sur le niveau de performance obtenu.

Finalement, le Chapitre 6 conclut cette thèse par une appréciation générale des résultats obtenus et de leur apport à l'atteinte des objectifs, ainsi que sur l'identification de futures avenues de recherches pertinentes.

Chapitre 2

Métaheuristiques parallèles sur GPU

L'optimisation combinatoire cherche à résoudre divers problèmes pouvant être rencontrés dans de nombreux domaines académiques et industriels. Un problème d'optimisation combinatoire consiste alors à trouver la meilleure solution possible dans un ensemble discret et fini de solutions. Pour ce faire, une fonction objectif est définie et associe à chaque solution une valeur qu'il faut minimiser ou maximiser pour atteindre l'optimum. Bien souvent, des contraintes spécifiques doivent également être respectées afin d'obtenir des solutions réalisables. La plupart de ces problèmes sont considérés comme étant $\mathcal{NP}$-*difficiles* et ne peuvent être résolus par un algorithme polynomial. Dans ce contexte, deux familles de méthodes sont généralement utilisées. La première regroupe les méthodes exactes, les plus connues étant les méthodes de séparation-évaluation (*branch and bound*) et la programmation dynamique. Elles permettent d'obtenir la solution optimale au problème en temps exponentiel dans le pire des cas [Tal09]. La seconde famille rassemble les méthodes approchées (ou heuristiques) fournissant en temps polynomial une solution approchée réalisable mais pas nécessairement optimale.

Les métaheuristiques font partie de cette seconde famille. Osman et Laporte [OL96] les définissent comme étant des processus itératifs guidant une heuristique subordonnée dans le but de trouver des solutions les plus rapprochées possibles de la solution optimale du problème traité. Pour ce faire, elles contiennent généralement des mécanismes évitant l'emprisonnement dans des régions non prometteuses de l'espace de recherche (optima locaux) et permettant de se diriger vers les optima globaux. Dréo *et al.* [DPST03] résument la progression des métaheuristiques par l'alternance de trois phases principales : diversification/exploration, intensification/exploitation et mémoire/apprentissage. La première phase est un processus de récolte d'informations sur le problème. La seconde phase les utilise ensuite afin de parcourir les solutions les plus intéressantes. La dernière phase permet à l'algorithme, grâce à l'expérience acquise, de mieux guider le processus de recherche en évitant les mauvais optima locaux.

Les métaheuristiques les plus performantes intègrent un mécanisme de Recherche Locale. Ce dernier vise à améliorer une solution initiale par des transformations locales, remplaçant la solution courante par un meilleur voisin jusqu'à ce que plus aucun mouvement ne soit améliorant. Osman et Laporte [OL96] considèrent la Recherche Locale comme étant une heuristique trouvant des optima locaux souvent éloignés de l'optimalité. Certaines métaheuristiques, comme la Recherche avec Tabous ou le Recuit Simulé, sont des techniques de Recherche Locale tentant

de remédier à ce problème d'optimalité [DPST03]. Des métaheuristiques telles que l'Optimisation par Colonie de Fourmis sont, quant à elles, hybridées avec une Recherche Locale afin d'améliorer la qualité des solutions construites.

Les métaheuristiques ont prouvé qu'elles pouvaient être efficaces pour résoudre de nombreux problèmes d'optimisation combinatoire académiques et industriels. Cependant, face à des problèmes de grande taille, elles ont besoin d'un temps de calcul et d'une quantité d'espace mémoire considérables pour être efficaces dans l'exploration de l'espace de recherche. L'intérêt voué à leur parallélisation a donc augmenté durant ces dernières années. En effet, un des buts principaux du parallélisme est de réduire le temps d'exécution d'un algorithme en utilisant un ordinateur parallèle. Les différentes architectures d'ordinateurs existantes ont été classifiées par la taxonomie de Flynn [Fly66] en 1966. Parmi celles-ci, le modèle *SISD* (*Single Instruction, Single Data*) correspond à la machine conventionnelle de von Neumann. Cette dernière fonctionne séquentiellement et est composée principalement d'une mémoire et d'un processeur nommé *CPU* (*Central Processing Unit*). Les modèles *SIMD* (*Single Instruction, Multiple Data*) et *MIMD* (*Multiple Instruction, Multiple Data*) correspondent, quant à eux, aux ordinateurs parallèles. Ces derniers contiennent plusieurs processeurs connectés par un réseau. Le modèle SIMD possède une seule unité de contrôle ce qui implique l'exécution de la même instruction par tous les processeurs. En revanche, le modèle MIMD en possède plusieurs ce qui lui permet donc d'effectuer différentes instructions sur différentes données.

L'exécution d'une application sur une architecture parallèle implique la répartition de son travail sur les différents processeurs. La granularité d'une application parallèle mesure alors la quantité de travail réalisée par chaque processeur. Plus spécifiquement, un gros grain correspond à un parallélisme au niveau du programme et un grain fin, à un parallélisme au niveau de l'instruction. Une granularité intermédiaire est également retrouvée avec, par exemple, un parallélisme au niveau de la tâche. Plus le grain de l'application est gros, plus le nombre de processeurs utilisé doit être faible et plus il est fin, plus ce nombre doit être grand. Après avoir décidé de la granularité de l'application parallèle, le mode de communication des processeurs doit être défini. En effet, ils peuvent interagir selon deux paradigmes : passage de messages ou mémoire partagée. Le modèle à passage de messages est particulièrement adapté aux architectures à mémoire distribuée de type MIMD. Chaque processeur possède sa mémoire locale et communique avec les autres processeurs en envoyant et en recevant des données. Le modèle à mémoire partagée, quant à lui, est aussi bien adapté aux architectures de type SIMD qu'à celles de type MIMD. Les processeurs communiquent entre eux par des lectures et écritures dans une mémoire globale partagée.

Dans un contexte de résolution de problèmes d'optimisation par des métaheuristiques, le parallélisme peut permettre de réduire le temps de calcul mais également d'améliorer la qualité des solutions obtenues. En effet, une métaheuristique parallèle peut, par l'utilisation de mécanismes tels que l'échange d'informations, obtenir de meilleures solutions que l'algorithme séquentiel en un temps inférieur ou égal. Par l'application de paramètres appropriés, elle peut également améliorer la robustesse de la recherche par rapport à sa version séquentielle sans nécessiter de calibration excessive [CT10]. Enfin, les métaheuristiques parallèles permettent de résoudre des problèmes de grande taille en des temps de calcul raisonnables en comparaison des versions séquentielles. Le parallélisme peut donc représenter un apport considérable aux métaheuristiques et un nombre grandissant de travaux sont effectués afin d'exploiter ce

potentiel.

La plupart de ces travaux abordent la conception des métaheuristiques parallèles à un niveau d'abstraction relativement élevé qui convient aux ordinateurs parallèles conventionnels. Cependant, la recherche évolue très rapidement dans le domaine du calcul parallèle et de nouvelles architectures émergent comme les processeurs graphiques ou *GPU* (*Graphics Processing Unit*). Ces derniers sont caractérisés par une architecture massivement parallèle et une organisation de la mémoire complexe. L'exploitation du potentiel des GPU dans le domaine des métaheuristiques parallèles est donc souhaitable, mais les approches de parallélisation existantes sont généralement inadéquates dans un contexte de calcul GPU. Il devient alors nécessaire d'adapter les métaheuristiques à cette nouvelle architecture et d'exploiter au mieux ses ressources. Afin de comprendre les problématiques liées au GPU, ce chapitre débute par une description de son architecture ainsi que de son modèle de programmation. Une revue des travaux existants de parallélisation GPU des métaheuristiques est ensuite détaillée. Enfin, les travaux de référence sur les classifications qui apportent une vue globale et fondamentale aux métaheuristiques parallèles sont présentés. Cet état de l'art permettra de determiner des axes de recherche intéressants qui seront décrits à la fin de ce chapitre.

2.1 Architecture GPU

Avant l'arrivée des processeurs multicœurs, la puissance des processeurs monocœurs était principalement améliorée en augmentant la densité des transistors [Pac11]. Cependant, cette amélioration nécessitait l'augmentation de la puissance électrique et donc, de l'énergie thermique générée. Des systèmes de refroidissement à air ont été créés pour dissiper cette chaleur mais ont rapidement atteint leurs limites. Afin de contourner ce problème tout en continuant cette quête de performance, la voie des processeurs multicœurs a été empruntée. Plusieurs cœurs ou *UAL* (Unité Arithmétique et Logique) simples ont été intégrés au sein d'une seule puce permettant ainsi d'obtenir théoriquement un processeur de même puissance nécessitant une consommation électrique réduite. Le fabricant AMD propose actuellement des processeurs possédant jusqu'à 16 cœurs [amd12] qui opèrent en parallèle et partagent les caches et la mémoire. Les GPU, quant à eux, sont des architectures massivement parallèles comprenant des centaines d'unités de calcul comme le montre la Figure 2.1. De plus, ils ont l'avantage d'être accessibles du fait de leur prix raisonnable. En effet, en comparaison avec les processeurs possédant 4 cœurs, les GPU Tesla C2050 et C2070 permettent d'obtenir des performances de calcul équivalentes pour une consommation 20 fois moins importante et un prix 10 fois plus faible [nvi10].

Les tout premiers GPU destinés au grand public ont été inventés en 1999 par la société NVIDIA [nvi12]. Ils étaient à la base conçus pour le traitement des données graphiques issues des jeux et des applications multimédia. Leur potentiel étant considéré comme sous-exploité, des chercheurs ont commencé à les utiliser pour le calcul scientifique. Le calcul générique sur processeurs graphiques (*GPGPU : General-Purpose Processing on Graphics Processing Units*) s'est alors développé. Toutefois, l'accessibilité aux performances des GPU était très limitée puisque les scientifiques devaient utiliser des langages de programmation graphique tels que GLSL (*OpenGL Shading Language*) [gls06] et Cg (*C for Graphics*) [cg12]. NVIDIA a donc dé-

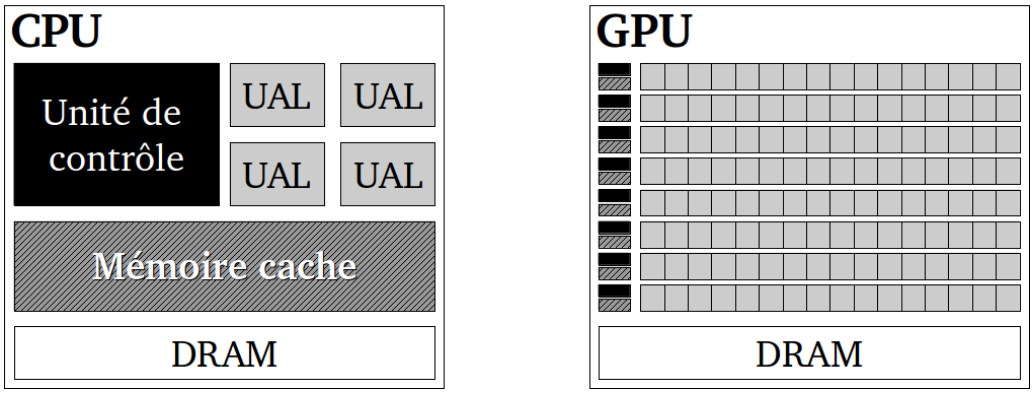


Figure 2.1 – Architectures d'un CPU multicœur et d'un GPU.

cidé de faciliter la programmation des GPU pour les applications scientifiques et de la rendre compatible avec des langages hautes performances tels que C et C++. Pour ce faire, cette société a conçu le modèle de programmation parallèle CUDA (*Compute Unified Device Architecture*) [cud11] qui permet ainsi d'exploiter la puissance de leurs GPU. D'autres langages tels que OpenCL (*Open Computing Langage*) [ope12] peuvent également être utilisés pour les programmer. Cependant, la majorité des travaux scientifiques rapportés dans la littérature utilisant spécifiquement les GPU de NVIDIA, CUDA est généralement choisi comme modèle de programmation.

La parallélisation GPU nécessite un système composé d'un *hôte* (le CPU) et d'un *accélérateur* (le GPU), chacun possédant ses propres mémoires. Le CPU gère le programme principal et les communications avec le GPU. Ce dernier, quant à lui, fonctionne comme un co-processeur dédié aux parties du programme exécutées en parallèle. La Figure 2.2 illustre l'architecture simplifiée typique d'un GPU qui inclut plusieurs multiprocesseurs (*Streaming Multiprocessors*) fonctionnant en parallèle et la mémoire du GPU (mémoire de l'accélérateur). Plus spécifiquement, chaque multiprocesseur est composé d'un ensemble de processeurs (*Streaming Processors*), de registres, d'une mémoire partagée, d'un cache mémoire constant et d'un cache mémoire texture.

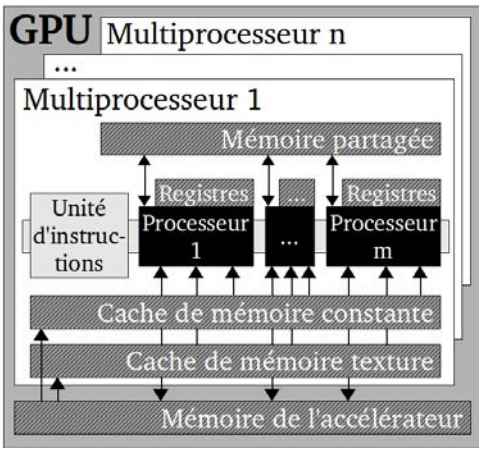


Figure 2.2 – Schéma de l'architecture d'un GPU.

Les différentes mémoires distinguées sur cette architecture diffèrent par leur taille, leur latence mémoire et leur type d'accès (lecture seule ou lecture/écriture). Leurs caractéristiques sont résumées dans le Tableau 2.1.

Mémoire	Taille	Latence mémoire	Type d'accès	Localité
globale	3-6 Go	400-600 cycles d'horloge	lecture/écriture	GPU
locale	-	400-600 cycles d'horloge	lecture/écriture	GPU
texture	$\approx$ 3-6 Go	$<$ 600 cycles d'horloge	lecture seule	GPU
constante	64 ko	$<$ 600 cycles d'horloge	lecture seule	GPU
partagée	16-48 ko	$\approx$ 2-32 cycles d'horloge	lecture/écriture	multiprocesseur
registre	32 bits	$\approx$ 1-22 cycles d'horloge	lecture/écriture	processeur

Tableau 2.1 – Caractéristiques typiques des différentes mémoires d'un GPU moderne.

La mémoire de l'accélérateur est relativement importante en taille (typiquement 3 à 6 Go) mais lente d'accès (jusqu'à 600 cycles d'horloge). Elle est divisée en plusieurs régions spécifiques dont font partie les mémoires **locale** et **globale**. Ces dernières peuvent être accédées en lecture et en écriture par tous les éléments de calcul du GPU. D'une part, la mémoire locale stocke les variables et larges structures de données automatiques qui consomment plus de registres que le nombre disponible. D'autre part, la mémoire globale stocke les données transférées du CPU. En effet, la création ou la suppression de données n'y sont pas permises durant l'exécution parallèle. Elles doivent donc être créées sur le CPU pour être ensuite allouées sur le GPU et copiées dans la mémoire globale.

Les caches **constant** et **texture** de chaque multiprocesseur sont liés aux mémoires constante et texture qui sont physiquement situées dans la mémoire de l'accélérateur. Comme ils tirent avantage de caches matériels du GPU, ils sont accessibles en lecture seule par les processeurs mais sont plus rapides d'accès. La mémoire constante est très limitée en taille (typiquement 64 ko) alors que la taille de la mémoire texture peut être ajustée de façon à occuper la mémoire de l'accélérateur disponible. De plus, le cache texture est optimisé pour la localité spatiale 2D permettant ainsi d'accélérer des accès non consécutifs en mémoire mais proches spatialement.

Contrairement à la mémoire globale, la mémoire **partagée** est locale à chaque multiprocesseur. Elle est accessible en lecture et en écriture par tous les processeurs composant le multiprocesseur où elle est située. Elle est divisée en banques mémoire (16 à 32) de mots de 32 bits pouvant être accédées simultanément. Malgré sa petite taille (typiquement 16 à 48 ko), elle est rapide d'accès tant qu'il n'y a pas de conflits de banques (de 2 à 32 cycles d'horloge). Ces conflits se produisent lorsque les adresses des demandes mémoire se trouvent dans la même banque. Les accès à ces banques sont alors sérialisés. La mémoire partagée étant plus rapide que la mémoire globale, elle doit donc être exploitée autant que possible.

Enfin, chaque multiprocesseur contient entre 8192 et 65536 **registres** de 32 bits. Ce sont les mémoires les plus rapides du GPU (1 cycle d'horloge) mais elles impliquent l'utilisation de la mémoire locale lente lorsqu'elles sont utilisées en trop grand nombre. De plus, les accès peuvent être retardés (jusqu'à 22 cycles d'horloge typiquement) en raison des dépendances lecture-après-écriture et des conflits de banques des registres. De façon générale, les différentes mémoires du

GPU possèdent des caractéristiques spécifiques et doivent être gérées minutieusement de sorte à maximiser la performance des applications parallèles.

Le modèle de programmation CUDA [cud11] est basé sur le concept de noyaux (*kernels*). Ce sont des fonctions écrites en C et exécutées en parallèle par un grand nombre de threads CUDA. Comme l'illustre la Figure 2.3, ces threads sont regroupés en blocs (1D, 2D ou 3D), eux-mêmes rassemblés au sein d'une grille (1D, 2D ou 3D). Cette dernière est lancée sur le GPU et les blocs sont distribués aux multiprocesseurs pour être exécutés indépendamment les uns des autres. Lorsque l'exécution de certains blocs est achevée, d'autres sont lancés sur les multiprocesseurs disponibles. Les tailles de grille (nombre de blocs) et de bloc (nombre de threads) sont configurables. Elles sont limitées respectivement à 65535 blocs par dimension et à 512 ou 1024 threads par bloc.

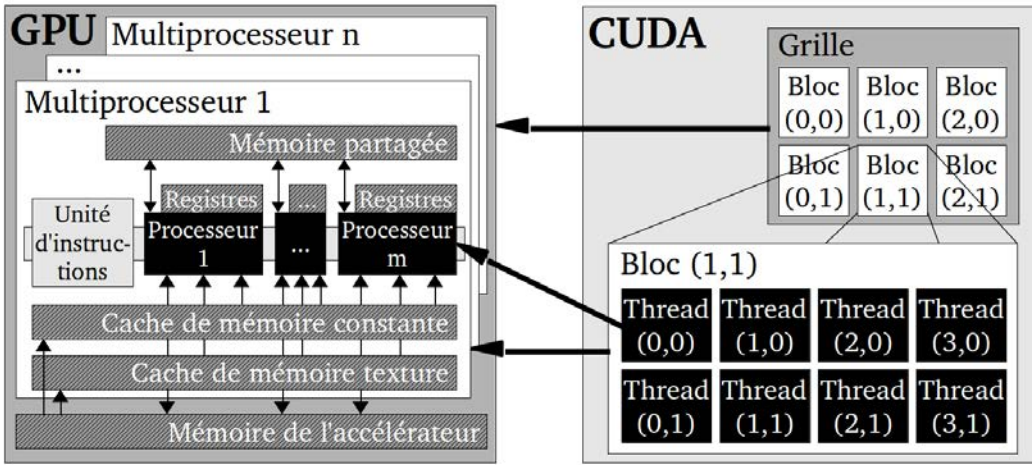


Figure 2.3 – Schéma du modèle de programmation CUDA.

Afin de gérer les centaines de threads des différents blocs, les multiprocesseurs emploient un modèle d'architecture nommé *SIMT* (*Single Instruction, Multiple Thread*) sous un modèle apparenté au parallélisme de données. Pour ce faire, le système rassemble les threads au sein de chaque bloc en groupes (*warps*) de 32 threads qui sont exécutés simultanément durant des cycles d'horloge successifs. Comme le montre la Figure 2.4, chaque warp contient les threads dont les identifiants sont consécutifs et en ordre croissant. Par conséquent, le premier warp regroupe les threads d'identifiants 0 à 31, le second regroupe les threads d'identifiants 32 à 63, etc. Pour maximiser l'efficacité du parallélisme, le nombre de threads par bloc doit donc être un multiple de la taille des warps.

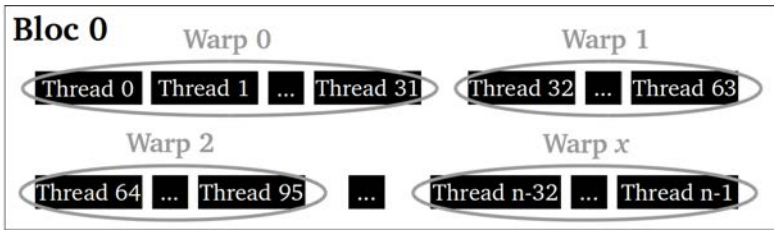


Figure 2.4 – Création des warps.

Les threads composant un warp commencent à la même adresse dans le programme mais sont libres de s'exécuter indépendamment des autres. Un warp effectuant une instruction à la fois, le programme est plus performant quand les 32 threads qui le composent empruntent le même chemin d'exécution. Il est donc important de noter que les instructions de contrôle (*if*, *switch*, *do*, *for*, *while*, etc.) peuvent affecter l'efficacité d'un algorithme. En effet, ces instructions forcent les threads d'un même warp à diverger dans certains cas, c'est-à-dire, à prendre différents chemins ou branches dans le programme. Par conséquent, les chemins d'exécution sont sérialisés, augmentant le nombre total d'instructions effectuées dans le warp. La Figure 2.5 illustre cette divergence de branchement au sein d'un warp.

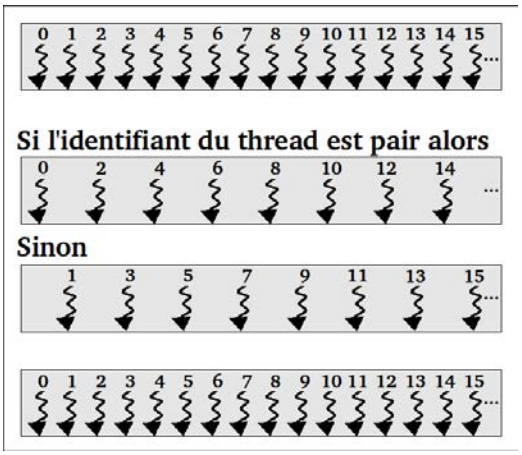


Figure 2.5 – Divergence de branchement au sein d'un warp.

Le nombre de warps pouvant résider sur un multiprocesseur est compris entre 24 et 64 (totalisant de 768 à 2048 threads) selon la capacité du GPU. Plusieurs blocs peuvent donc être traités au même moment au sein d'un multiprocesseur. Ils sont alors appelés blocs actifs. Leur nombre varie entre 0 et 8 et dépend du nombre de registres utilisés et de la quantité de mémoire partagée occupée, ces ressources étant communes à tous les threads les composant. NVIDIA a mis en place un tableur nommé *CUDA Occupancy Calculator* permettant de connaître, entre autres, le nombre de blocs actifs par multiprocesseur, facilitant ainsi le paramétrage des tailles de grille et de bloc. Un aperçu de ce tableur est présenté dans la Figure 2.6. Afin d'obtenir un plus grand nombre de blocs actifs, le nombre de registres utilisés peut être fixé à la compilation mais cela entraîne souvent une perte de performance.

En ce qui concerne la communication entre les différents threads d'un même bloc, un mécanisme de synchronisation est utilisé pour coordonner leurs échanges. Au contraire, les blocs sont indépendants les uns des autres et ne possèdent pas de mécanismes explicites de synchronisation. Cependant, ils peuvent implicitement coordonner leurs exécutions au retour sur le CPU. Par conséquent, si les blocs doivent se synchroniser au sein d'un noyau, ce dernier doit être divisé en plusieurs noyaux. La mémoire partagée étant locale à chaque bloc, elle peut donc être utilisée pour la coopération des threads dans un bloc. En revanche, lorsque des threads appartenant à différents blocs veulent communiquer, ils doivent obligatoirement passer par la mémoire globale plus lente. Toutefois, s'il y a suffisamment d'instructions arithmétiques indépendantes exécutées durant ces accès, une grande partie de sa latence peut être dissimulée par le programmateur de threads. En effet, quand l'instruction exécutée par un warp nécessite

1.) Select Compute Capability (click):	2,0
2.) Enter your resource usage:	
Threads Per Block	128
Registers Per Thread	32
Shared Memory Per Block (bytes)	6890
3.) GPU Occupancy Data is displayed here and in the graphs:	
Active Threads per Multiprocessor	**896**
Active Warps per Multiprocessor	**28**
Active Thread Blocks per Multiprocessor	**7**
Occupancy of each Multiprocessor	**58%**
Physical Limits for GPU Compute Capability:	**2,0**
Threads per Warp	32
Warps per Multiprocessor	48
Threads per Multiprocessor	1536
Thread Blocks per Multiprocessor	8
Total # of 32-bit registers per Multiprocessor	32768
Register allocation unit size	64
Register allocation granularity	warp
Shared Memory per Multiprocessor (bytes)	49152
Shared Memory Allocation unit size	128
Warp allocation granularity (for register allocation)	
Allocation Per Thread Block	
Warps	4
Registers	4096
Shared Memory	6912
Maximum Thread Blocks Per Multiprocessor	Blocks
Limited by Max Warps / Blocks per Multiprocessor	8
Limited by Registers per Multiprocessor	8
Limited by Shared Memory per Multiprocessor	**7**

Figure 2.6 – *CUDA Occupancy Calculator.*

le résultat d'une opération prenant du temps, ce warp est placé dans une zone d'attente et un second warp est lancé. Plus il y a de warps actifs par multiprocesseur, et donc de blocs actifs, plus la latence peut être masquée. Les accès à la mémoire globale nécessitent également d'être au maximum *coalescents* pour permettre des gains de vitesse. Les threads doivent alors accéder à des zones contiguës en mémoire dans l'ordre des indices de threads. L'impact des accès non coalescents peut être réduit grâce au mécanisme de cache inclus dans les cartes graphiques de génération 2.x et plus. Enfin, les retards introduits par les dépendances lecture-après-écriture des registres peuvent être ignorées si les multiprocesseurs contiennent au moins 192 threads actifs pour les masquer.

Les nombreuses spécificités des GPU complexifient grandement le développement d'implémentations parallèles sur ce type d'architecture. Le gain potentiel de performance qu'elles permettent d'obtenir est toutefois d'un intérêt certain pour le domaine de l'optimisation combinatoire et plus particulièrement des métaheuristiques. Différents travaux sur la parallélisation GPU de ces méthodes peuvent d'ailleurs être trouvés dans la littérature, souvent sous forme de premières études exploratoires ou expérimentales. Les sections suivantes présentent une revue de ces travaux en les classant en deux familles telles que définies par Talbi [Tal09] : la Section 2.2 présente les métaheuristiques à solution unique et la Section 2.3, les métaheuristiques à population de solutions. Pour chacune des principales métaheuristiques retrouvées dans la littérature, une description de son fonctionnement et de ses principales caractéristiques est explicité, puis les travaux portant sur sa parallélisation GPU sont détaillés. Dans chaque cas, un

accent particulier est mis sur les différents niveaux de parallélisme et de granularité utilisés, ainsi que sur la gestion des différentes mémoires.

2.2 Métaheuristiques à solution unique

Les métaheuristiques à solution unique font partie d'une famille de méthodes manipulant et transformant une seule solution durant la recherche [Tal09]. Leur phase dominante est l'exploitation qui intensifie la recherche dans des régions locales afin de parcourir les solutions les plus intéressantes. Cette famille comprend, entre autres, la Recherche avec Tabous, le Recuit Simulé et la Recherche Locale Itérée. Osman et Laporte [OL96] soulignent le fait que la Recherche avec Tabous et le Recuit Simulé partagent la capacité de guider une Recherche Locale pour éviter les mauvais optima locaux. De même, la Recherche Locale itérée guide une Recherche Locale sous-jacente en modifiant légèrement une solution à chaque itération pour tenter de la diriger vers l'optimum global. Le mécanisme de Recherche Locale étant une composante essentielle de ce type de métaheuristique, la section suivante décrit tout d'abord les travaux consacrés à la parallélisation GPU de ce mécanisme en particulier. Ensuite, ceux concernant la Recherche avec Tabous et le Recuit Simulé sont détaillés.

2.2.1 Recherche Locale

La Recherche Locale (RL) est une heuristique qui est apparue à la fin des années 1950 avec la création des premiers algorithmes d'échange d'arcs pour le problème du Voyageur de Commerce [Cro58, AL03]. Le fonctionnement de la RL basique, ou méthode de descente (*local search descent method / hill climbing algorithm*), est décrit dans l'Algorithme 2.1. Tout d'abord, la RL débute avec une solution initiale produite aléatoirement ou par une heuristique de construction. Le voisinage de cette solution est ensuite généré et exploré. L'algorithme essaye alors d'améliorer itérativement la solution courante en la remplaçant par un voisin de coût plus petit. La recherche se termine lorsque la solution courante ne possède plus aucun voisin améliorant. Un optimum local est atteint, bien souvent éloigné de l'optimalité [OL96].

```
Générer une solution S
Tant que S n'est pas un optimum local Faire
    Transformer S afin d'obtenir ses voisins V
    Évaluer le voisinage V
    Remplacer S par le voisin choisi par la règle de pivotement
Retourner la meilleure solution
```

Algorithme 2.1 – Recherche Locale

Un concept essentiel associé à la RL est la notion de structure de voisinage [VA95, AL03]. Cette dernière assigne à chaque solution un ensemble de solutions voisines générées par des transformations locales (inversion, transposition, déplacement, etc.). La plupart des structures de voisinage sont basées sur des fonctions d'échange (*k-exchange*) qui modifient un certain nombre d'éléments k de la solution. Lorsque k est petit, les solutions obtenues sont généralement

de moins bonne qualité. L'augmentation de k peut favoriser l'obtention de meilleures solutions mais peut aussi mener à des temps de calcul prohibitifs. Des voisinages plus complexes ont donc été créés afin de permettre une bonne exploration de la recherche en temps raisonnable. Par exemple, le voisinage de profondeur variable (*variable-depth neighborhood*), conçu par Lin et Kernighan [KL70, LK73], est obtenu en effectuant une séquence d'échanges sur la solution courante plutôt qu'un seul échange.

La règle de pivotement est un élément clé de la RL qui dicte le choix de la solution voisine qui remplacera la solution courante [Yan90]. Les plus utilisées sont les règles *best-improvement* et *first-improvement*. Dans le premier cas, tous les voisins de la solution courante sont évalués et celui qui produit la plus grande amélioration est sélectionné. Dans le second cas, le premier mouvement améliorant est accepté et les autres sont abandonnés. D'autres méthodes peuvent être utilisées telles que les règles *random-improvement* qui choisit un voisin aléatoirement ou *least-improvement*, qui produit l'amélioration la plus petite.

Parallélisation GPU des RL

Les travaux de Luong *et al.* [LMT10b, LMT11b] se spécialisent dans la parallélisation sur GPU des RL à large voisinage. Ils sont basés sur le principe que la qualité des solutions obtenues par une RL peut être améliorée en augmentant la taille du voisinage. Cependant, un plus grand nombre de voisins implique également un temps de résolution plus long, d'où la nécessité d'implémenter un algorithme parallèle. De façon générale, les parties de la RL qui sont les plus consommatrices en temps de calcul sont la génération et l'évaluation des solutions. Ces deux phases sont donc déplacées sur le GPU pour être exécutées en parallèle et le reste de l'algorithme demeure sur le CPU. Les données d'entrée du problème sont allouées et copiées dans la mémoire globale du GPU une seule fois au début de l'algorithme puis placées dans la mémoire texture. De leur côté, la structure stockant les coûts des différents voisins et la solution candidate sont copiées dans la mémoire globale du GPU. Le noyau génère ensuite tous les voisins en parallèle, les évalue et copie leur coût dans la structure prévue à cet effet. Cette dernière est enfin transférée sur le CPU afin d'y sélectionner la nouvelle solution courante. Dans cette implémentation, la mémoire partagée est utilisée pour les opérations de réduction qui permettent d'obtenir le coût minimum de tous les voisins.

Lors de l'exécution du noyau, chaque thread est associé à la génération et à l'évaluation d'un voisin. Ce dernier peut être défini par un nombre d'indices variable selon le type de voisinage utilisé. Par exemple, si le voisinage est obtenu en échangeant deux éléments de la solution courante, un voisin comportera deux indices : la position du premier élément et la position du second élément. Cependant, un thread est identifié par un seul indice. Il est donc indispensable de définir une relation entre l'identifiant de ce thread et le voisin généré. Selon le type de problème, une solution peut être représentée de différentes manières. Par exemple, pour les problèmes binaires, une solution correspond à un vecteur de valeurs binaires de taille n. Un voisinage avec distance de Hamming de 2 (modification de 2 bits) a alors une taille égale à $[n \cdot (n-1)]/2$. Soient i et j les indices des deux bits à modifier dans la solution, l'indice $f(i,j)$ du thread correspondant est égal à $i \cdot (n-1) + (j-1) - [i \cdot (i+1)]/2$. De même, à partir de l'indice du thread $f(i,j)$, les indices i et j du premier et deuxième bit à modifier peuvent être calculés. Les calculs des correspondances entre thread et voisin pour différents types de problèmes (binaire,

permutation, représentation vectorielle discrète et vecteur de valeurs réelles) et différents types de voisinage sont décrits dans [LMT11b].

L'algorithme parallèle résultant a été implémenté au sein d'une Recherche avec Tabous et quatre problèmes avec différents encodages ont été testés sur plusieurs configurations GPU. Une accélération maximale de 73,30 avec une distance de Hamming de 2 est obtenue pour le problème du perceptron permuté (binaire) et de 243,00 pour la fonction continue de Weierstrass (vecteur de valeurs réelles). Le problème d'affectation quadratique (permutation), pour $[n \cdot (n-1)]/2$ voisins, permet d'obtenir jusqu'à 18,60 d'accélération. Enfin, le problème du Voyageur de Commerce (permutation) utilise un opérateur d'échange de deux éléments. Plus le nombre de cœurs des GPU augmente, plus l'accélération augmente : jusqu'à 19,70 pour le Tesla M2050 contenant 448 cœurs. Les plus plus grandes instances de ce problème (pr2392, fnl4461 ou rl5915) n'ont cependant pas pu être résolues sur des GPU ayant 240 cœurs ou moins en raison du nombre de registres limité. Aucune de ces études ne fait état de la qualité des solutions obtenues.

Afin de pouvoir résoudre les problèmes de plus grande taille, les auteurs ont intégré à leur algorithme un mécanisme permettant de contrôler le nombre de threads par bloc (*thread control*). En général, les approches trouvées dans la littérature énumèrent et testent les valeurs possibles ce qui est trop coûteux en temps. Le mécanisme créé par les auteurs envoie, quant à lui, des "vagues" de threads pour régler les paramètres durant les premières itérations de la RL. Les différents essais sont alors mesurés pour trouver la meilleure configuration. Avec ce mécanisme, les plus gros problèmes ont pu être exécutés et une accélération maximale de 19,90 est obtenue pour le problème du Voyageur de Commerce.

Luong *et al.* [LLMT10] ont également intégré leur algorithme de RL parallèle au sein d'une Recherche avec Tabous Itérée. Le problème considéré est celui de l'affectation quadratique à 3 dimensions. Le voisinage utilisé échange deux positions dans les deux permutations formant la solution courante. Sa taille est donc égale à $[n \cdot (n-1)]^2/2$. Des accélérations allant jusqu'à 6,10 ont pu être obtenues. Luong [Luo11] explique les différences d'accélération entre les différents problèmes par leur type de fonction d'évaluation. Celles qui nécessitent un grand nombre de calculs sont bien adaptées à l'architecture du GPU alors que celles qui sont davantage liées à la mémoire doivent essayer de tirer au maximum avantage des mémoires petites et rapides du GPU. Les différentes applications des auteurs ont été intégrées au logiciel ParadisEO [MLBT11] dédié au développement de méthodes d'optimisation approchées parallèles hybrides.

Les algorithmes de RL à relances multiples (*multi-start* ou *restarts*) sont aussi étudiés. Ils répètent la RL plusieurs fois à partir de solutions initiales différentes. Luong *et al.* [LMT11a] effectuent les différents relances/RL en parallèle. Chaque RL est associée à un thread afin de minimiser le transfert de données entre le CPU et le GPU. Tout d'abord, les données sont allouées et copiées sur le GPU. Les solutions de chaque RL sont ensuite générées et évaluées en parallèle. Enfin, le processus est répété jusqu'à ce que le critère de fin soit atteint. Afin de tester leur modèle, les algorithmes de descente, de Recherche avec Tabous et de Recuit Simulé ont été implémentés avec relances multiples. Une accélération maximale de 12 est obtenue dans ces travaux. De leur côté, O'Neil *et al.* [OTB11] résolvent le problème du Voyageur de Commerce en stockant la matrice des distances dans la mémoire partagée. Ils affectent plusieurs relances aux différents threads qui utilisent une RL *2-opt*. Une accélération maximale de 61,90 a été rapportée sur un problème de 100 villes.

Les travaux de Schulz [Sch12] sont quant à eux plus techniques et visent à adapter les RL à l'architecture Fermi des GPU NVIDIA. Le problème de tournées de véhicules avec contrainte de capacité est résolu par une RL best-improvement *2-opt* et *3-opt*. Leur premier algorithme parallèle *The Benchmark Version*, considéré comme étant le résultat de plusieurs travaux dans la littérature, est utilisé comme version de base. La solution initiale est tout d'abord créée et évaluée. Les voisins sont ensuite identifiés et les données nécessaires copiées sur le GPU. À chaque itération de la RL, les hiérarchies de segments (parties de la solution résultantes de la suppression d'arcs) sont créées sur le GPU puis les objectifs et contraintes sont évalués. Le meilleur voisin est finalement trouvé et le mouvement est effectué. Pas à pas, les points faibles de l'algorithme parallèle sont identifiés pour en améliorer sa performance. Afin de limiter l'accès à la mémoire locale lente du GPU, les segments sont placés dans les registres et la mémoire partagée est utilisée pour stocker les données souvent accédées. De plus, les opérations arithmétiques coûteuses sont remplacées par des opérations sur les bits en restreignant la taille des parties de la hiérarchie à des puissances de 2. Des tests sont aussi effectués afin de trouver la meilleure taille possible des blocs de threads. L'auteur étudie également les accès mémoires et conclut que la proximité en mémoire des segments de threads voisins est bénéfique mais pas cruciale. Il souligne finalement l'idée que la taille du voisinage est importante car si elle est trop petite, l'implémentation n'est pas performante, et si elle est trop grande, le temps passé ne sera pas utile. La taille du voisinage devrait alors être comprise entre 10^6 et 10^9 selon la technologie GPU utilisée. Dans l'ensemble de ces travaux, aucune information n'est fournie à propos de la qualité des solutions obtenues.

2.2.2 Recherche avec Tabous

La métaheuristique de Recherche avec Tabous (RT) a été initialement proposée par Glover [Glo89, Glo90]. Elle met en œuvre des mécanismes directement inspirés de la mémoire humaine pour tirer bénéfice des erreurs du passé. Son fonctionnement est présenté dans l'Algorithme 2.2. À chaque itération, le voisinage de la solution courante est généré par diverses transformations locales puis évalué. Ces phases peuvent être accélérées en utilisant des listes de candidats qui permettent de réduire la taille du voisinage. Le meilleur voisin est ensuite choisi pour devenir la solution courante même si celui-ci la dégrade. Cette détérioration permet ainsi d'éviter l'emprisonnement dans des optima locaux et de se diriger vers les optima globaux. Enfin, une mémoire à court terme, nommée liste taboue, est mise à jour. Elle contient les mouvements dits tabous, c'est-à-dire, les derniers mouvements qui ont permis de passer des solutions courantes aux meilleurs voisins. Ces mouvements sont temporairement interdits afin d'éviter le retour à des solutions précédentes trop rapidement et d'empêcher les cycles. Cette mémoire est considérée comme une stratégie de diversification qui permet l'exploration de régions non visitées de l'espace de recherche. Cependant, il serait pénalisant d'interdire un mouvement améliorant grandement la solution. Un critère d'aspiration est donc intégré et autorise certains mouvements tabous à être choisis. L'algorithme s'arrête après l'atteinte d'un critère d'arrêt correspondant à la stagnation de la recherche ou à un nombre d'itérations fixé. Une description complète de cette méthode peut être trouvée dans Glover et Laguna [GL97].

```
Initialiser la solution initiale S
Tant que le critère d'arrêt n'est pas atteint Faire
    Transformer S afin d'obtenir ses voisins V
    Évaluer V
    Sélectionner le meilleur voisin v non tabou
    Mettre à jour la liste taboue
    Remplacer la solution courante S par v
Retourner la meilleure solution
```

Algorithme 2.2 – Recherche avec Tabous

Parallélisation GPU de la RT

Dans la plupart des travaux de parallélisation de la RT, soit quelques parties de l'algorithme sont déplacées sur le GPU, soit la totalité de la méthode. Janiak *et al.* [JJL08] parallélisent l'évaluation du voisinage selon le principe qu'elle est la partie la plus consommatrice en temps de calcul pour le problème du Voyageur de Commerce. Ils utilisent le langage de programmation graphique HLSL (*High Level Shader Language*) où les structures de données natives d'un GPU sont les textures [JL07]. Ces dernières peuvent être vues comme des tableaux en deux dimensions situés dans la mémoire vidéo (vRAM) du GPU correspondant à la mémoire de l'accélérateur. Dans les applications graphiques, les textures sont affichées à l'écran alors que pour le GPGPU, elles servent à stocker des données en mémoire. Des transformations sont effectuées sur des textures d'entrée et le résultat est sauvegardé dans une texture de sortie. Au début de l'algorithme, les auteurs ont donc envoyé deux textures sur le GPU qui contiennent les distances et le voisinage. À chaque itération, ce dernier est évalué en parallèle sur le GPU pour obtenir le meilleur voisin et les données sont mises à jour. Les expérimentations portent sur des problèmes allant jusqu'à 100 villes et l'accélération maximale rapportée est de 1,71. Zhang *et al.* [ZLG+09], parallélisent la construction et l'évaluation du voisinage avec le modèle de programmation CUDA. Leur but est de trouver un cycle ordonné dans une base de données dans le domaine de la recherche d'images. Le cycle optimal est obtenu par la minimisation de l'entropie des images effectué par la RT. Chaque bloc CUDA est associé à une donnée du cycle et chaque thread à une dimension particulière de cette donnée. Les fragments de données sont stockés dans la mémoire partagée. La parallélisation de la RT repose sur trois noyaux. Chacun construit le voisinage du cycle courant pour un mouvement précis (échange, déplacement et inversion d'éléments du cycle) puis calcule l'entropie des différents voisins. Celui qui possède l'entropie minimale est ensuite transmis au CPU. L'accélération maximale obtenue par cette implémentation s'élève à 67,00.

Czapinski et Barnes [CB11] parallélisent quant à eux la totalité de la RT pour résoudre le problème d'ordonnancement de type *flow-shop* de permutation. La matrice des temps de traitement est placée dans la mémoire constante et le voisinage des solutions courantes dans la mémoire globale. Plutôt que de stocker les voisins les uns à la suite des autres, les premiers éléments de chaque voisin sont d'abord mémorisés, puis les seconds éléments, etc. Cette manière non intuitive de stockage permet des accès coalescents à la mémoire globale. Au début de l'algorithme, chaque thread applique un mouvement à la solution courante afin de générer un voisin. Ensuite, deux implémentations parallèles sont proposées pour la phase d'évaluation du voisi-

nage. Dans la première, nommée *"séquentielle"*, chaque thread évalue une solution en parallèle. La matrice des temps de fin des tâches ne peut être contenue dans la mémoire partagée mais peut être évaluée dans le sens des lignes ou des colonnes. Une seule ligne ou une seule colonne est alors stockée dans la mémoire partagée à la fois. La deuxième implémentation parallèle, nommée *"parallèle"*, évalue les diagonales de cette matrice en parallèle. Des accélérations maximales allant jusqu'à 48,69 sont obtenues avec la version *"parallèle"* et jusqu'à 89,01 pour la version *"séquentielle"*. La version *"parallèle"* est plus rapide que la version *"séquentielle"* pour les plus petits ensembles de données, mais plus lente pour les grands. Les auteurs l'expliquent par le fait que la version *"séquentielle"* nécessite plus de permutations ce qui permet de mieux utiliser les multiprocesseurs. Les phases de sélection de la meilleure solution et de mise à jour de la liste taboue sont également effectuées en parallèle sur le GPU avec un nombre de threads total égal à la taille du voisinage. Pour la première phase, les valeurs supérieures ou égales à celle de la meilleure solution connue sont modifiées par les threads. Ainsi, elles ne sont pas choisies lors de la sélection de la valeur minimale. Les valeurs sont ensuite groupées et réparties sur les différents blocs CUDA. Chaque groupe trouve l'indice de sa valeur minimale et l'inscrit dans un tableau. Si le tableau a plus d'un élément, le processus est réitéré avec comme valeurs initiales ce nouveau tableau. Lors de la seconde phase, les éléments de la liste taboue sont mis à jour par les threads en parallèle. Pour les petits ensembles de données, ces deux phases sont plus rapides lorsqu'elles ne sont pas parallélisées malgré les transferts fréquents de données.

Enfin, Bozejko *et al.* [BUW10] intègrent la RT au sein d'un algorithme hybride parallèle nommé *meta²heuristics* (*meta-square-heuristics*). Celui-ci est utilisé pour résoudre le problème d'ordonnancement d'ateliers flexibles. Il est composé de deux modules : la sélection de machines exécutée de façon séquentielle sur le CPU et la programmation d'opérations effectuée en parallèle sur GPU. Chaque module utilisant une métaheuristique spécifique, deux versions de l'algorithme en découlent. La première, *TSBM²h*, utilise la méthode de RT pour le premier module, alors que la seconde, *PBM²h*, emploie une métaheuristique à population de solutions. Le second module utilise, quant à lui, soit un algorithme d'insertion constructif (*INSA : INSertion Algorithm*), soit un algorithme de RT (*TSAB : Tabu Search Algorithm with Backtracking*). Tout d'abord, les algorithmes TSBM²h et PBM²h génèrent sur le CPU un voisinage ou une population. Des séquences d'opérations sont calculées en parallèle sur le GPU ainsi que les coûts d'affectation obtenus avec les algorithmes INSA ou TSAB. Le minimum est renvoyé au CPU afin de mettre à jour les différentes structures de données. Dans le cas de l'algorithme PBM²h, des affectations de machines sont ensuite trouvées sur le CPU puis les machines libres sont insérées aléatoirement les positions vides sur le GPU. Des accélérations allant jusqu'à 54,75 sont obtenues avec l'algorithme TSBM²h et jusqu'à 20,51 avec l'algorithme PBM²h.

2.2.3 Recuit Simulé

Le Recuit Simulé (RS) est une métaheuristique inspirée de la technique du recuit en métallurgie. Elle est née des travaux de Metropolis *et al.* [MRR+53] mais fut utilisée pour la première fois dans la résolution de problèmes d'optimisation combinatoire par Kirkpatrick *et al.* [KGV83]. Son fonctionnement est décrit dans l'Algorithme 2.3. Tout d'abord, la température T de l'algorithme est initialisée avec une valeur élevée. Une solution initiale S est ensuite transformée en S' par une modification élémentaire (permutation, translation, etc.).

Cette transformation se traduit par une variation d'énergie ΔE correspondant à la fonction objectif du problème. La règle de Metropolis permet alors d'accepter la solution S' avec une certaine probabilité. Cette règle autorise la métaheuristique à dégrader la solution courante afin de permettre à la recherche de s'extraire des optima locaux. Si $\Delta E \leq 0$, S' devient la nouvelle solution courante S. Sinon, la probabilité que la solution soit acceptée est de $e^{\frac{-\Delta E}{T}}$. La température T permet ainsi de contrôler l'acceptation des dégradations. Quand T a une valeur élevée, une grande proportion des dégradations sont acceptées alors que quand $T = 0$, elles sont toutes refusées. Ce processus est répété jusqu'à l'équilibre thermodynamique ou jusqu'à l'atteinte d'un critère d'arrêt comme un nombre d'itérations fixé. La température T est ensuite abaissée et le processus redémarre à partir de S. Le système est dit figé quand la température est nulle ou quand plus aucun mouvement n'est accepté. L'algorithme s'arrête alors et retourne la meilleure solution trouvée depuis le début de l'algorithme.

```
Initialiser la solution S et la température T
Tant que le système n'est pas figé Faire
    Tant que l'équilibre thermodynamique n'est pas atteint Faire
        Répéter
            Modifier de façon élémentaire S en S'  // variation d'énergie ΔE
            Si ΔE ≤ 0 Alors // règle d'acceptation de Metropolis
                S = S'  // acceptation de la solution
            Sinon
                Choisir aléatoirement un nombre réel R ∈ [0; 1]
                Si R ≤ e^(-ΔE/T) Alors
                    S = S'  // acceptation de la solution
                Sinon
                    Refuser la solution
        Jusqu'à ce que la solution soit acceptée
    Si le système n'est pas figé Alors
        Diminuer lentement T
Retourner la meilleure solution
```

Algorithme 2.3 – Recuit Simulé

Parallélisation GPU du RS

Han *et al.* [HCRK11] proposent une implémentation GPU d'un algorithme de *floorplanning* basé sur le RS pour l'automatisation de la conception électronique. La version séquentielle de base débute avec un plan initial représenté par un arbre. À plusieurs reprises, un mouvement aléatoire lui est appliqué et le modifie selon l'acceptation du mouvement, créant ainsi une longue chaîne de dépendance. Cette dernière doit être cassée pour adapter efficacement l'algorithme au GPU tout en préservant la qualité de solution. L'algorithme parallèle explore donc l'espace de recherche différemment en appliquant les mouvements en parallèle. Tout d'abord, le CPU crée et copie l'arbre ainsi que les données nécessaires sur le GPU. Un nombre de blocs CUDA égal à la taille du plan sont ensuite lancés sur les multiprocesseurs et chaque bloc copie l'arbre dans sa mémoire partagée. Un seul thread par bloc est en charge de lui appliquer un mouvement et de l'évaluer. Enfin, les données nécessaires à l'acceptation du meilleur mouvement sont ramenées

sur le CPU. Avec cette méthode, seules de faibles accélérations sont obtenues en raison du trop grand nombre de copies entre les différentes mémoires. Pour y remédier, les données rarement accédées ne sont plus conservées dans la mémoire partagée selon le principe que le temps de copie est plus long que le temps d'accès à la mémoire globale. Plusieurs threads au lieu d'un seul sont également utilisés pour les copies en mémoire. Enfin, les accès à la mémoire globale sont effectués de façon coalescente. Une accélération maximale de 17,00 est obtenue mais une dégradation de la qualité de solution est constatée. Avec cette méthode, le GPU effectue le même nombre de mouvements que le CPU mais le type d'exploration est différent. Le CPU parcourt l'espace de recherche en profondeur alors que le GPU l'explore en largeur. Deux paramètres additionnels sont donc intégrés à l'algorithme pour choisir le nombre de mouvements effectués en parallèle (largeur) et le nombre de mouvements consécutifs effectués par chaque thread (profondeur). L'arbre est déplacé sur le GPU pour éviter les transferts entre les mémoires du GPU et du CPU. Les plans intermédiaires sont ensuite copiés de la mémoire partagée à la mémoire globale de sorte à permettre à un thread de commencer avec un plan calculé par un autre thread. Quand la profondeur est augmentée, la qualité de solution est améliorée mais l'accélération diminue. Les auteurs en concluent qu'un gain d'accélération vient généralement avec une dégradation de la qualité des solutions. Plusieurs techniques sont utilisées afin d'améliorer cette dernière comme l'utilisation d'une température différente pour chaque thread. Une accélération maximale de 498,40 est obtenue avec dégradation de la qualité de solution et de 188,75 avec une qualité de solution similaire ou meilleure.

Choong *et al.* [CBZ10] ont choisi l'approche de déplacer la totalité du RS sur le GPU. Les blocs exécutent alors le même algorithme sur des sous-ensembles différents dont les données sont stockées dans la mémoire partagée et les threads calculent les mouvements en parallèle. L'accélération maximale rapportée par cette approche est de 25,31. Li *et al.* [LTLL10] ont plutôt parallélisé deux phases du RS sur le GPU. La première génère les nouvelles solutions en parallèle par différents threads à l'aide des méthodes d'échange de deux nœuds et de 2-opt. La seconde applique la règle de Metropolis et effectue une réduction parallèle afin d'obtenir la nouvelle solution destinée à remplacer la solution courante. Dans ce cas, des accélérations allant de 2,52 jusqu'à 6,44 ont été obtenues. Stivala *et al.* [SSW10] ont implémenté un algorithme nommé *SA Tableau Search* basé sur la méthode du RS. Cet algorithme est utilisé pour la recherche de structures de protéines dans des bases de données. La méthode du RS résout alors le problème de l'extraction de sous-séquences les plus similaires possibles. Deux niveaux de parallélisation sont appliqués à l'algorithme. Les différentes relances du RS sont tout d'abord initialisées de façon aléatoire et exécutées en parallèle. Celles-ci permettent de comparer deux structures entre elles. Ensuite, plusieurs comparaisons avec les multiples structures de la base de données sont également effectuées en parallèle en exploitant l'indépendance des comparaisons par paires. Un bloc CUDA est alors en charge d'une comparaison et chacun de ses threads effectue une relance du RS. Le noyau a été implémenté de deux manières différentes. Dans la première, toutes les structures de données sont stockées dans la mémoire globale. Dans la seconde, celles de la base de données sont placées dans la mémoire globale et celles de la requête dans la mémoire constante. Chaque bloc copie également en parallèle les structures nécessaires dans la mémoire partagée pour y accéder plus rapidement. Cette seconde implémentation est plus rapide mais ne permet pas l'exécution des plus grosses structures, contrairement à la première. L'accélération maximale obtenue par cette implémentation s'élève à 34,00.

De façon générale, les travaux de parallélisation des métaheuristiques à solution unique présentent des techniques variées afin de tirer profit des architectures GPU et donnent des accélérations très variables allant de 1,71 à 498,40. De plus, une dégradation de la qualité des solutions est généralement associée à la réduction des temps de calcul. La section suivante décrit les principaux travaux basés sur les métaheuristiques à population de solutions.

2.3 Métaheuristiques à population de solutions

Les métaheuristiques à population de solutions appartiennent à une famille de méthodes faisant évoluer un ensemble de solutions [Tal09]. La phase dominante de ces métaheuristiques est l'exploration. Elle privilégie la diversification de la recherche en la réorientant périodiquement vers des régions peu visitées de l'espace des solutions. Cette famille regroupe, entre autres, les Algorithmes Évolutionnaires, l'Optimisation par Essaims Particulaires et l'Optimisation par Colonie de Fourmis. Les sections suivantes décrivent ces trois métaheuristiques ainsi que les travaux portant sur leur parallélisation GPU.

2.3.1 Algorithmes Évolutionnaires

Les Algorithmes Évolutionnaires (AE) [BFM97] sont inspirés de la théorie de l'évolution des espèces de Darwin. Selon elle, les êtres vivants diffèrent les uns des autres et leurs caractéristiques sont plus ou moins adéquates à leur environnement. Les individus qui peuvent se reproduire transmettent à leur descendance des caractéristiques utiles qui leur ont permis de rester en vie dans le passé. Par conséquent, seuls les individus les mieux adaptés à leur environnement survivent au fur et à mesure des générations. Plusieurs familles d'algorithmes descendent des AE, les plus connues étant les Algorithmes Génétiques (AG) proposés par J. H. Holland [Hol75]. L'Algorithme 2.4 décrit le fonctionnement basique d'un AE. En premier lieu, une population initiale d'individus est générée le plus souvent aléatoirement. Une valeur de performance nommée *fitness* est attachée à chaque individu. Elle dépend de la fonction objectif et mesure le degré d'adaptation de chaque individu face au but visé. Durant un certain nombre de générations, l'AE fait évoluer la population de façon à améliorer la fitness de ses membres. Pour ce faire, différents opérateurs sont appliqués à ces derniers pendant le processus évolutionnaire. Le premier, nommé sélection, choisit les individus qui vont participer à la phase de reproduction. Les parents sont le plus souvent sélectionnés par rapport à leur fitness ce qui favorise la survie des meilleurs individus et l'intensification de la recherche vers les meilleures solutions. Différents opérateurs de sélection peuvent être appliqués comme la sélection par tournoi ou la sélection proportionnelle. L'opérateur de croisement génère ensuite les descendants des individus sélectionnés à partir de leurs gènes. L'opérateur de mutation modifie généralement aléatoirement un certain nombre d'individus. Enfin, la fitness des individus est évaluée afin d'appliquer l'opérateur de remplacement qui détermine quels individus doivent disparaître de la population.

D'autres familles d'algorithmes descendent des AE. Les algorithmes mémétiques [MN92] sont des AE hybridés avec une RL. Les AE/AG cellulaires (AEc/AGc) [AD08] sont des algorithmes structurés où les individus ne peuvent interagir qu'avec leurs voisins (nord, sud, est

et ouest) et ont été conçus pour des machines massivement parallèles. Les Stratégies d'Évolution (SE) [Rec65] manipulent des vecteurs de valeurs réelles à l'aide d'opérateurs de mutation et de sélection. La Programmation Évolutionnaire (PE) [FOW66] n'utilise pas d'opérateur de croisement mais privilégie des opérateurs de remplacement stochastiques. Les algorithmes co-évolutionnaires [Bar96] gèrent deux populations d'individus ou plus. Dans ce contexte, la fitness dépend de l'individu en question mais aussi des individus des autres populations. Enfin, l'Évolution Différentielle (ED) [SP97] est conçue pour les problèmes d'optimisation continue. Elle est inspirée des AG et des SE combinés avec une technique géométrique de recherche.

```
Générer une population initiale
Évaluer l'adaptation de chaque individu de la population
Tant que le nombre de générations n'est pas atteint Faire
    Appliquer l'opérateur de sélection
    Appliquer l'opérateur de croisement sur les individus sélectionnés
    Appliquer l'opérateur de mutation sur les individus selon une probabilité
    Évaluer l'adaptation de chaque individu de la population
    Appliquer l'opérateur de remplacement
Retourner le meilleur individu
```

Algorithme 2.4 – Algorithme Évolutionnaire

Parallélisation GPU des AE

Les AE ont été les premières métaheuristiques à être implémentées sur GPU. Dans les travaux les plus anciens, l'accessibilité aux performances des GPU était limitée par la nécessité d'utiliser des langages de programmation graphique complexes pour les scientifiques non-initiés. Ces travaux antérieurs à l'apparition du modèle de programmation CUDA ont été réalisés par Yu *et al.* [YCP05], Fok *et al.* [FWW07] et Wong et Wong [WW09]. Ces auteurs répartissent les génomes composant les chromosomes dans différentes textures. Chaque point de la texture au format RGBA contient quatre données de type réel (*float*) associées chacune à un génome. Si les chromosomes sont stockés les uns à la suite des autres dans une texture, l'augmentation du nombre de génomes par chromosome nécessitera une réorganisation de la texture. Les quatre premiers génomes de chaque chromosome sont donc stockés dans la première texture, les quatre suivants dans la deuxième, etc. Lorsque le nombre de génomes augmente, il suffit alors d'ajouter une texture qui contient les nouveaux génomes des chromosomes. Cependant, quand les populations contiennent un trop grand nombre d'individus, la quantité de mémoire peut devenir insuffisante. Fok *et al.* [FWW07] ont choisi la PE plutôt qu'un AG en suivant le principe qu'elle ne possède pas de phase de croisement et est ainsi mieux adaptée aux GPU. Ils privilégient la parallélisation des phases d'évaluation et de mutation en raison de leur exécution sur des données indépendantes et les déportent donc sur le GPU. La phase de compétition demeure quant à elle sur le CPU car le gain de performance associé à la parallélisation ne parvient pas à compenser les temps nécessaires au transfert des données. Comme les GPU ne possédaient pas à l'époque de bibliothèque permettant de générer les nombres aléatoires nécessaires aux AE, ceux-ci étaient produits et exploités de différentes façons. Par exemple, le CPU pouvait les générer à chaque itération et les stocker sous forme de textures sur le GPU. Cependant, ces

opérations impliquent des temps de communication prohibitifs entre le CPU et le GPU. Wong et Wong [WW09] ont donc créé un nouvel opérateur génétique de sélection afin de réduire le nombre de transferts de nombres aléatoires. Cet opérateur est un compromis entre la sélection globale et la sélection locale. La première donne les meilleurs résultats mais est trop consommatrice en nombres aléatoires alors que la seconde n'en utilise aucun mais impose des limitations pouvant amener à une convergence lente. Quant à eux, Yu *et al.* [YCP05] ont implémenté un générateur congruentiel linéaire pour générer les nombres aléatoires.

L'apparition du modèle de programmation CUDA a entraîné une recrudescence des travaux sur les AE parallèles sur GPU. Deux stratégies de parallélisation se sont alors distinguées : la déportation sur GPU des fonctions les plus coûteuses en temps de calcul et la déportation de la totalité de l'algorithme.

Partant du principe que la fonction généralement la plus coûteuse en temps de calcul est l'évaluation des fitness, Wahib *et al.* [WMMA11] l'ont déplacé au sein d'un noyau. Les instructions conditionnelles qui provoquent une divergence de threads ont été transformées de façon à employer les opérations sur les bits. L'occupation a également été maximisée en trouvant un compromis entre le nombre de threads par bloc, le nombre de registres utilisés et la quantité de mémoire partagée occupée. Luong *et al.* [LMT10a] ont modifié le modèle en îles existant afin d'apporter une approche générique d'AE parallèles sur GPU. Dans celle-ci, le CPU gère la totalité du processus évolutionnaire de chaque île puis envoie les individus sur le GPU par la mémoire globale. Le GPU les évalue ensuite en parallèle en associant chaque thread à un individu, puis les renvoie au CPU. Ce schéma est relativement simple à implémenter, mais la copie fréquente des données entre le CPU et le GPU peut diminuer sa performance. Sharma et Collet [SC10], Leung *et al.* [LLTS12] et Ben-Shalom *et al.* [BSARK12] déportent également la fonction d'évaluation sur le GPU. Dans un contexte d'hybridation entre un AE et une RL, Luong *et al.* [LMT10c] parallélisent la génération et l'évaluation des voisins. Les données d'entrée sont copiées seulement une seule fois afin de minimiser les transferts de données entre le CPU et le GPU. Les données qui ne sont pas modifiées durant l'exécution du noyau sont placées dans la mémoire texture et chaque transformation locale est associée à un identifiant unique de thread. Les solutions voisines sont alors générées et évaluées en parallèle puis récupérées sur le CPU afin d'exécuter l'AE. Maitre *et al.* [MKQ+12] déportent l'évaluation et la RL sur le GPU. Ils n'utilisent que la mémoire globale selon le principe que les données des problèmes de plus grande taille ne peuvent être contenues dans la mémoire partagée. De plus, ils associent chaque thread à un individu et assignent à chaque bloc le nombre maximum de threads autorisé sur un multiprocesseur en fonction des limitations techniques du GPU. Selon le type de problème et les différentes utilisations des mémoires, les stratégies basées sur la déportation des fonctions les plus coûteuses mènent à des accélérations maximales variant entre 6,50 et 132,00.

La seconde stratégie de parallélisation des AE consiste à déporter la totalité ou une grande partie de l'algorithme sur le GPU. Dans la plupart des travaux, une population initiale est générée où chaque individu est géré par un thread CUDA. Cette population peut aussi être structurée en plusieurs sous-populations ou îles, chacune étant associée à un bloc. Luong *et al.* [LMT10a], Pospichal *et al.* [PJS10, PSJ10] et Feng *et al.* [FZY10] appliquent cette stratégie. Le nombre d'individus par île est alors limité au nombre maximum de threads par bloc. La population est générée sur le CPU et transférée dans la mémoire globale du GPU. Ensuite, le processus évolutionnaire est entièrement exécuté en parallèle afin de minimiser les copies

entre le CPU et le GPU. Enfin, les communications entre les îles sont effectuées par la mémoire globale. Lorsque la mémoire partagée stocke les données relatives aux populations, le temps d'exécution de l'algorithme est réduit mais la résolution des gros problèmes est pénalisée. Zhu [Zhu11], Ramírez-Chavez *et al.* [RCCRT11] et Pinel *et al.* [PDB12] utilisent également cette stratégie et assignent chaque thread à un individu de la population.

Dans un grand nombre de travaux appliquant cette stratégie de parallélisation, les opérateurs classiques sont modifiés afin d'obtenir une accélération maximale tout en conservant la qualité des solutions. Munawar *et al.* [MWMA09] transforment l'opérateur de RL qu'ils considèrent comme la phase la plus coûteuse et difficile à implémenter de l'algorithme. En effet, l'algorithme original peut se terminer avant le nombre maximum d'itérations et impliquer une divergence de threads. Un nombre d'itérations variable est donc fixé durant lequel les threads ne peuvent pas quitter la RL. Par conséquent, l'algorithme est plus efficace puisque les threads d'un même warp suivent tous le même branchement. Les auteurs utilisent également les masques et les opérations logiques sur les bits dans la plupart des opérateurs afin d'éviter les instructions conditionnelles et la divergence des threads, tout comme Wahib *et al.* [WMMA11] et Jaros et Pospichal [JP12]. Dans leurs travaux, Munawar *et al.* stockent les enfants dans la mémoire partagée pour les petits problèmes et dans la mémoire globale pour les plus gros. Jaros et Pospichal stockent, quant à eux, les paramètres dans la mémoire constante et utilisent la mémoire partagée pour sauvegarder les données temporaires. De leur côté, Soca *et al.* [SBPE10] gèrent la divergence des threads d'une autre manière. En effet, les opérateurs de croisement et de mutation de base sont probabilistes et ne sont donc appliqués que sur une partie des individus. Les threads, correspondant aux individus, ne suivent alors pas tous le même branchement au sein de leur bloc. Les auteurs effectuent donc le choix d'application des opérateurs au niveau du bloc. Ainsi, tous les threads du bloc appliquent l'opérateur ou non. Enfin, Arora *et al.* [ATD10] modifient la mutation binaire pour qu'elle soit mieux adaptée aux GPU. Dans sa version de base, elle doit modifier chaque bit du chromosome avec une certaine probabilité ce qui requiert beaucoup de nombres aléatoires. S'inspirant de la mutation réelle qui modifie une seule variable avec une certaine probabilité, l'implémentation proposée remplace un seul bit aléatoirement.

Oiso *et al.* [OMYO11] constatent que les approches qui assignent chaque individu à un thread doivent généralement employer une population plus grande que dans les AG traditionnels pour être efficaces. Ils allouent alors chaque individu à un bloc et ses gènes aux threads du bloc. Par exemple, lors de la phase d'évaluation, chaque thread charge les variables nécessaires et calcule une partie de la fonction d'évaluation qui est stockée dans la mémoire partagée. Ensuite, les threads se synchronisent puis effectuent une réduction parallèle pour obtenir la valeur de la fonction d'évaluation. De même, Huang *et al.* [HHL12] traitent le problème d'ordonnancement de type *flow-shop* en associant chaque chromosome à un bloc pour que ses données puissent être contenues dans la mémoire partagée. Chaque thread calcule la date de fin de chaque tâche et la fonction de fitness de son chromosome associé. L'étape de mutation est remplacée par une étape d'immigration où de nouveaux membres sont générés aléatoirement et intégrés à la population. Dans les algorithmes proposés par Krömer *et al.* [KPSA11] et Fujimoto et Tsutsui [FT10], chaque solution est également traitée par un bloc. Fabris et Krohling [FK12] implémentent une version parallèle d'un algorithme ED co-évolutionnaire utilisant deux populations d'individus. La fitness de chaque individu est évaluée par une compétition avec tous les membres de l'autre population. Si la taille de chaque population est égale à n, alors il faut n^2 appels de la fonction

objectif pour chaque population et chaque génération. Lors du calcul de la fitness, chaque bloc est associé à un individu et les threads effectuent les calculs de fitness associés à tous les individus de la seconde population. Selon le type de problème et les différentes utilisations de la mémoire, les implémentations basées sur la déportation de la totalité de l'AE sur le GPU fournissent des accélérations maximales variant entre 7,80 et 2074,00.

Certains travaux utilisent plusieurs GPU dans la parallélisation des AE. Vidal et Alba [VA10] proposent deux implémentations d'un AGc qui s'exécutent sur un nombre différent de GPU. La première en utilise un seul et stocke la population entière dans la mémoire globale. Chaque individu est initialisé et évalué en parallèle par un thread puis un noyau identifie ses voisins. Les opérateurs génétiques sont ensuite appliqués en parallèle et les threads sont synchronisés. Enfin, les individus sont remplacés par leur enfant selon la condition définie dans l'opérateur de remplacement. La seconde implémentation est une extension de la première mais utilise plusieurs GPU. Chaque thread CPU est associé à un GPU. La population est décomposée en sous-populations affectées à chaque GPU et stockées dans la mémoire globale. Les frontières, qui contiennent les voisins nécessaires à chaque sous-population pour appliquer les opérateurs, sont calculées et copiées sur les GPU. Après chaque phase de remplacement, les frontières des sous-populations sont envoyées sur le CPU. Ce dernier les redistribue ensuite aux GPU spécifiques. Les expérimentations ont montré que la qualité des solutions trouvées est proche de l'optimum. Des accélérations maximales allant jusqu'à 771,00 ont été obtenues par l'implémentation utilisant plusieurs GPU. Cependant, une accélération maximale de 0,917 a été obtenue entre cette version et celle s'exécutant sur un seul GPU. Utiliser plusieurs GPU n'est donc pas efficace dans ce cas précis car la taille de la population est trop petite et les échanges de données avec le CPU sont trop fréquents.

Les travaux portant sur les AE sont les plus nombreux et possèdent l'historique le plus long dans la littérature sur la parallélisation GPU des métaheurstiques. Par conséquent, ils ont permis non seulement d'établir les bases du domaine, mais aussi de mettre en évidence différentes techniques permettant d'exploiter les mémoires rapides du GPU. Résolvant le problème du Voyageur de Commerce, l'implémentation de Chen *et al.* [CDJN11] nécessite une structure de données contenant les distances entre les villes. Comme cette structure est de taille trop importante pour résider dans la mémoire partagée, un tableau de coordonnées est donc stocké et les distances sont recalculées à chaque itération de l'algorithme. Pour optimiser les modèles de canaux ioniques, Ben-Shalom *et al.* [BSARK12] décomposent la procédure d'intégration numérique en étapes de façon à ce que les threads ne requièrent pas l'historique complet de la solution. Ainsi, à chaque itération, le thread reçoit un ensemble de données de la mémoire globale suffisamment petit pour être stocké dans les registres. Tsutsui et Fujimoto [TF09] n'utilisent pas d'entiers mais un tableau de caractères non signés pour contenir un grand nombre d'individus dans la mémoire partagée. Des entiers courts non signés permettent également de stocker plus de données dans la mémoire constante. Pedemonte *et al.* [PAL11] étudient d'ailleurs l'impact de l'espace mémoire occupé par la population d'un AG avec représentation binaire sur la performance de l'algorithme, les capacités de certains types de mémoire du GPU étant limitées. Ils analysent les différences de performance entre une implémentation utilisant un type de données booléen et une implémentation regroupant les bits dans un type de données non booléen. La première est facile à implémenter mais gaspille de la mémoire. En effet, la taille d'un booléen est comprise entre 1 et 4 octets alors qu'une donnée binaire ne nécessite qu'un bit. La seconde

implémentation utilise pleinement la mémoire mais nécessite des opérations bit à bit. Elle mène à une réduction du temps d'exécution GPU puisque les mémoires les plus rapides sont aussi celles les plus petites. Oiso *et al.* [OMYO11], Luong *et al.* [LMT10a] et Pospichal *et al.* [PJS10] se sont intéressés au tri des individus par rapport à leur fitness. Les méthodes séquentielles sont généralement basées sur le tri rapide qui est difficile à paralléliser sur GPU. Par conséquent, ces auteurs utilisent le tri bitonique qui a été déclaré comme l'un des tris les plus rapides sur GPU pour un petit nombre d'éléments [GZ06]. Il est donc particulièrement adapté au modèle en îles où les sous-populations ou îles sont de taille relativement petite. Cependant, ce tri nécessite un nombre d'éléments qui est une puissance de deux. Il est alors nécessaire d'ajouter des éléments fictifs.

2.3.2 Optimisation par Essaims Particulaires

L'Optimisation par Essaims Particulaires (OEP) a été inventée par Kennedy et Eberhart [KE95] en 1995. Cette métaheuristique est basée sur les comportements collectifs de déplacements d'animaux tels que les volées d'oiseaux, les bancs de poissons ou encore les essaims d'abeilles. Ces animaux progressent simultanément de façon assez complexe sans vision globale de leur environnement. Ils ne possèdent que des informations locales comme la position et la vitesse de leurs voisins. Par exemple, un banc de poissons est capable d'éviter un prédateur en se divisant brusquement en deux groupes et en se reformant tout en maintenant la cohésion du banc [DPST03]. Dans un algorithme OEP, ces animaux sont représentés par des particules et correspondent à des points de l'espace de recherche uni- ou multi-dimensionnel. Cette métaheuristique est présentée dans l'Algorithme 2.5. Tout d'abord, une population de particules définies sous forme de vecteurs est initialisée de façon aléatoire. Chaque particule i se trouve à une position $\vec{x_i}$ (x_{id} étant la position dans la dimension d) et possède une vélocité $\vec{v_i}$ correspondant à un vecteur de changement de position. Elle garde en mémoire la meilleure position $\vec{p_i}$ qu'elle a visitée, ainsi que celle de ses proches voisins $\vec{p_g}$. À chaque itération, les particules se déplacent en fonction de $\vec{x_i}$, $\vec{v_i}$, $\vec{p_i}$ et $\vec{p_g}$ puis leur qualité, aussi nommée *fitness*, est évaluée. Pour une particule i, la fitness correspond à la valeur de la fonction à optimiser F dans la position $\vec{x_i}$: $F(\vec{x_i})$. Les meilleures positions $\vec{p_i}$ et $\vec{p_g}$ pour chaque particule sont ensuite actualisées. Enfin, les vélocités sont mises à jour en tenant compte de l'importance relative de l'expérience individuelle et celle de la communication sociale. Une description complète de cette méthode peut être trouvée dans Clerc [Cle10].

Parallélisation GPU des algorithmes OEP

Les algorithmes OEP parallèles peuvent être de type synchrone ou asynchrone. Dans le premier cas, les positions $\vec{x_i}$, $\vec{p_i}$ et $\vec{p_g}$ d'une particule i sont actualisées lorsque la totalité des particules ont été évaluées. Dans le second cas, ces mises à jours sont autorisées juste après l'évaluation de i. Mussi *et al.* [MDC11] ont tout d'abord étudié les algorithmes OEP synchrones et proposent deux approches utilisant une topologie en anneau fixe. La première est nommée *SyncPSO* et déporte la totalité de l'algorithme sur le GPU. Cette approche se caractérise par une absence d'accès à la mémoire globale sauf au début et à la fin de l'algorithme. Elle utilise un ou plusieurs essaims qui correspondent aux blocs CUDA. Les particules de l'essaim,

Pour chaque particule i **Faire**
  Initialiser la position $\vec{x_i}$ et la vélocité $\vec{v_i}$
Tant que le critère d'arrêt n'est pas atteint **Faire**
  Pour chaque particule i **Faire**
    Si $F(\vec{x_i}) > F(\vec{p_i})$ **Alors**
      Pour chaque dimension d **Faire**
        $p_{id} = x_{id}$ *// mise à jour de la meilleure position de la particule i*
    Si $F(\vec{x_i}) > F(\vec{p_g})$ **Alors**
      Pour chaque dimension d **Faire**
        $p_{gd} = x_{id}$ *// mise à jour de la meilleure position globale*
    Pour chaque dimension d **Faire**
      Mettre à jour la vélocité v_{id} et la position x_{id} *// déplacement des particules*
Retourner la meilleure position

Algorithme 2.5 – Optimisation par Essaims Particulaires

associées aux threads, mémorisent leurs données personnelles dans les registres et communiquent entre elles par la mémoire partagée. Elles mettent à jour leur position, calculent leur fitness et actualisent leur meilleure position individuelle. Une réduction parallèle sur le vecteur de fitness est ensuite effectuée pour en obtenir la valeur minimale. Cette étape implique deux phases de synchronisation des threads. Enfin, le thread d'indice 0 met à jour la meilleure solution globale avec la valeur minimale du vecteur. Une fois toutes les générations effectuées, les données sont transférées vers la mémoire globale. Les auteurs constatent que cette stratégie apparaît comme étant efficace au niveau algorithmique mais impose certaines limitations. Quand un trop grand nombre de particules sont simulées, le nombre de registres disponibles n'est plus suffisant pour contenir les données. La mémoire locale lente est donc utilisée. De plus, lorsqu'un seul essaim est simulé, la puissance de calcul du GPU n'est pas totalement exploitée car seul un multiprocesseur est utilisé parmi tous ceux disponibles.

La seconde approche de parallélisation proposée par les auteurs est nommée *RingPSO*. Elle a été conçue pour palier aux limitations de l'approche *SyncPSO*. Contrairement à cette dernière, les phases principales de l'algorithme sont parallélisées au sein de trois noyaux qui utilisent des configurations de blocs et de threads différentes. Le travail peut donc être mieux réparti sur tous les multiprocesseurs disponibles. Les données principales sont stockées dans la mémoire globale du GPU afin de partager des informations entre les noyaux. Elles sont organisées de façon à éviter les accès non coalescents. Dans le premier noyau, chaque bloc met à jour les données de sa particule avec un nombre de threads égal à la dimension du problème. De même, dans le second noyau, chaque bloc évalue la fitness d'une particule en utilisant un thread par coordonnée de position. Le dernier noyau, quant à lui, met à jour les meilleures fitness en associant chaque essaim à un bloc et chaque thread à une particule de l'essaim. Une deuxième version de cette approche a été réalisée en fusionnant les noyaux de mises à jour afin de réduire le nombre de synchronisations. Mussi *et al.* [MCC+10, MCD09] ont également utilisé cette stratégie pour la détection des panneaux de signalisation basée sur leur forme et leur couleur. Solomon *et al.* [STT11] proposent une approche similaire puisqu'ils décomposent la totalité de l'algorithme en plusieurs noyaux et modifient les configurations des blocs et des threads de chacun. Cependant, la matrice d'entrée est placée dans la mémoire texture pour accélérer l'algorithme. Certains noyaux assignent les threads aux dimensions de

la particule alors que d'autres les associent aux particules. Dans ce dernier cas, les threads composant un essaim peuvent être regroupés dans le même bloc. Zhou et Tan [ZT09, ZT10] et Wang *et al.* [WWW11] créent également plusieurs noyaux mais utilisent toujours un nombre de threads égal au nombre de particules. Selon le type de problème et les différentes utilisations des mémoires, les accélérations maximales varient entre 11,43 et 44,00 pour les versions synchrones des algorithmes OEP parallèles.

Mussi *et al.* [MNC11] ont ensuite étudié les algorithmes OEP parallèles asynchrones. Ils ont proposé une approche utilisant un seul noyau qui alloue un bloc par particule avec un thread par dimension. Le nombre de blocs pouvant être lancés simultanément sur un GPU étant limité, le nombre de particules utilisé par l'algorithme demeure inférieur au nombre maximum de blocs. De plus, les données des particules sont stockées dans la mémoire partagée. Les auteurs comparent leur algorithme asynchrone avec l'approche *RingPSO* sur plusieurs fonctions d'optimisation. Les deux versions obtiennent la même qualité de solution mais la version asynchrone est plus rapide. Une accélération maximale d'environ 270,00 est obtenue pour la fonction de Rastrigin. Cette dernière est résolue plus rapidement que les autres de par son utilisation des fonctions mathématiques complexes qui peuvent être remplacées par les fonctions rapides du GPU.

Bastos-Filho *et al.* [CBFN11] analysent quant à eux l'impact de différentes topologies sur un algorithme parallèle de type synchrone. Ce dernier est divisé en plusieurs noyaux pour créer des barrières de synchronisation implicites. Parmi les noyaux produits, seul celui recherchant la meilleure position $\vec{p_g}$ des voisins de chaque particule i diffère selon les topologies. Les auteurs étudient donc ce noyau en particulier et définissent un thread par particule. Dans la topologie en étoile, toutes les particules peuvent communiquer entre elles. Par conséquent, la meilleure du voisinage est également la meilleure de l'essaim. Une réduction parallèle sur l'essaim est donc utilisée pour trouver $\vec{p_g}$. La topologie en rayon est semblable mais les particules sont d'abord comparées au foyer de l'essaim (la particule d'indice 0) avant la réduction parallèle. Dans la topologie en anneau, les particules ne possèdent que deux voisins qui sont associés à des éléments contigus d'un tableau. Ils sont donc placés dans les éléments précédant et suivant celui de la particule, le meilleur étant l'un des deux. La topologie de von Neumann est similaire mais les particules possèdent quatre voisins. Le tableau qui les stocke devient donc une grille. Enfin, la topologie à quatre *clusters* est un mélange de celles en étoile et en anneau. Chaque particule peut communiquer avec toutes celles composant son cluster/essaim comme dans la topologie en étoile. Une réduction parallèle est donc utilisée pour trouver le meilleur voisin de l'essaim. Ensuite, les quatre particules servant de passerelle entre les différents clusters/essaims échangent leurs informations comme dans la topologie en anneau. La version asynchrone de l'OEP est également analysée. L'algorithme est totalement déporté sur le GPU dans un seul noyau. Il autorise la modification de $\vec{p_g}$ dès que les particules trouvent une meilleure position. Les phases de recherche du meilleur voisin sont similaires à celles de l'algorithme synchrone. Il n'y a cependant pas de garantie que la meilleure information soit considérée. La qualité des solutions et l'accélération obtenues dépendent du type de problème, de la topologie et du type d'algorithme (synchrone ou asynchrone). Par exemple, en utilisant les topologies à quatre *clusters* et en rayon pour la résolution de la fonction de Rosenbrock, la version synchrone produit des solutions de meilleure qualité que la version asynchrone. Avec les topologies en étoile, en anneau et de von Neumann, les résultats sont assez similaires entre les deux types

d'algorithmes. L'utilisation de la version asynchrone est donc privilégiée puisqu'elle est plus rapide.

Laguna-Sánchez *et al.* [LSOCCC+10] remplacent les différentes étapes de l'algorithme séquentiel par des noyaux afin que l'algorithme parallèle ait la même structure. Les trois stratégies de parallélisation employées par les auteurs sont celles développées pour les AEs par Cantú-Paz [CP98] et reprises par Belal et El-Ghazawi [BEG04] dans le cadre des algorithmes OEP. Dans chacune d'entre elles, un thread est créé par particule. La première utilise la stratégie de parallélisation globale où seul le calcul de la fonction objectif est réalisé en parallèle. La seconde utilise cette même stratégie mais réalise également la mise à jour des positions en parallèle. Cependant, la phase de comparaison des individus reste effectuée de façon séquentielle par le CPU. Les transferts CPU-GPU entre chaque noyau pénalisent donc ces deux stratégies globales. De plus, la deuxième stratégie ayant un temps d'exécution plus court que la première, les auteurs constatent que la performance des GPU est améliorée quand la quantité de travail distribuée aux threads augmente. La dernière approche est un modèle hybride utilisant les stratégies à grain épais et à grain fin. L'initialisation reste sur le CPU mais tous les autres calculs sont effectués en parallèle sur le GPU au sein d'un seul noyau.

Enfin, Cagnoni *et al.* [CBM12] ont constaté que les implémentations parallèles GPU de la littérature sont bien souvent comparées à des implémentations CPU n'utilisant pas toutes les ressources possibles (multicœur, SIMD) pour être performantes. S'éloignant du modèle de programmation CUDA utilisé dans leurs travaux antérieurs [MNC11], ils utilisent OpenCL pour compiler le code parallèle des GPU mais également celui des CPU multicœur. Les versions GPU et multicœur CPU sont comparées sur des fonctions mathématiques. Les versions GPU obtiennent des accélérations maximales allant de 1,00 à 6,00, montrant ainsi qu'elles sont plus rapides que les versions CPU. Cependant, les auteurs en concluent que les accélérations obtenues ne sont pas aussi grandes que lorsque la version CPU n'est pas parallélisée ni optimisée.

2.3.3 Optimisation par Colonie de Fourmis

Les algorithmes d'Optimisation par Colonie de Fourmis (OCF) sont inspirés du comportement collectif des fourmis qui résolvent naturellement des problèmes complexes comme le choix du plus court chemin de leur nid à une source de nourriture, sans vision globale du trajet. Pour ce faire, les fourmis communiquent entre elles par une substance nommée phéromone. Elles en déposent au sol et forment des pistes odorantes pouvant être suivies par les autres fourmis de la colonie. Le premier algorithme OCF, nommé *Ant System* (AS), a été initialement proposé par Dorigo et al. [DMC91, DMC96] et conçu pour résoudre le problème du Voyageur de Commerce. Cependant, il peut être appliqué à de nombreux problèmes d'optimisation combinatoire dans lesquels une heuristique de construction peut être définie. Les fourmis réalisent alors des marches aléatoires sur un graphe complet $G = (N, L)$ nommé graphe de construction [DS04]. N représente l'ensemble des nœuds du graphe où chacun est associé à un composant de la solution et L désigne l'ensemble des liens connectant les différents nœuds. Le fonctionnement du AS est décrit dans l'Algorithme 2.6. Pendant un certain nombre de cycles ou itérations, toutes les fourmis construisent une solution en se déplaçant dans le graphe. Elles sont placées aléatoirement sur un nœud de départ et avancent vers un autre nœud selon une règle de transition d'état. Cette dernière est basée sur l'heuristique définie pour le problème et la quantité de phéromone

déposée par les fourmis. Une fois toutes les solutions construites, chaque fourmi dépose une quantité de phéromone proportionnelle à la qualité de sa solution et la meilleure est mémorisée. Un mécanisme d'évaporation se produit ensuite afin d'effacer les traces de phéromone les plus insignifiantes.

```
Initialiser les pistes de phéromone
Tant que le critère de fin n'est pas atteint Faire
    Placer aléatoirement les fourmis sur chaque nœud
    Construire les solutions des fourmis selon une règle de transition d'état
    Mettre à jour les pistes de phéromone par dépôt des fourmis et évaporation
    Mémoriser la meilleure solution construite
Retourner la meilleure solution
```

Algorithme 2.6 – Optimisation par Colonie de Fourmis

La plupart des algorithmes OCF existants sont basés sur le AS et lui apportent certaines modifications. Le *MAX-MIN Ant System* (MMAS), proposé par Stützle et Hoos [SH00], cherche à tenir compte des informations récoltées pour exploiter les meilleures solutions trouvées pendant la recherche. Il apporte trois améliorations majeures au AS. Premièrement, seule la meilleure fourmi de l'itération courante ou de l'exécution totale est autorisée à mettre à jour les traces de phéromone. Deuxièmement, pour éviter une stagnation prématurée de la recherche, les valeurs de phéromone sont bornées entre une valeur maximale τ_{max} et une valeur minimale τ_{min}. τ_{max} est d'ailleurs utilisé pour initialiser les traces de phéromone au début de l'algorithme. Enfin, le MMAS incorpore également un mécanise de renforcement (*trail-smoothing mechanism*) ajustant l'intensité des traces de phéromone lorsque l'algorithme est proche de la convergence. Son but est de faciliter l'exploration des solutions en augmentant la probabilité de sélectionner les arêtes ayant une faible quantité de phéromone. Dans la méthode du AS avec stratégie élitiste [DMC96], la phéromone déposée sur les arêtes qui appartiennent à la meilleure solution est davantage renforcée comparativement aux autres. L'algorithme du AS_{rank} [BHS97] combine le AS avec élitisme et un classement des fourmis par ordre croissant de longueur de tournées. Seules les premières fourmis sont ainsi autorisées à mettre à jour les traces de phéromone de façon proportionnelle à leur rang. L'algorithme du *Ant Colony System* (ACS) [DG97], quant à lui, ajoute une règle de mise à jour locale de phéromone. Cette dernière est appliquée pendant la construction des solutions et permet un changement dynamique de l'attraction des arêtes. Enfin, Tsuitsui [Tsu06] propose une variante nommée *cunning Ant System* (*c*AS) où chaque fourmi génère une solution en empruntant une partie des solutions qui ont été construites dans les itérations précédentes.

Parallélisation GPU des algorithmes OCF

Li *et al.* [LHPQ09] implémentent un algorithme MMAS parallèle pour résoudre le problème du Voyageur de Commerce. Les étapes du MMAS sont réalisées dans plusieurs noyaux avec différentes configurations de blocs et de threads. La construction des tournées par les fourmis est décomposée en deux noyaux. Le premier choisit les $n-1$ prochaines villes à visiter où n est le nombre total de villes. Pour ce faire, chaque fourmi doit réaliser un grand nombre de

calculs. Elle est donc représentée par un bloc dont les threads effectuent ces calculs en parallèle. Le second noyau récupère la dernière ville qui n'a pas encore été visitée de la tournée, sans calculer la règle de transition d'état. Ensuite, la longueur des tournées construites est évaluée afin de trouver la plus courte par une réduction parallèle et d'actualiser la meilleure tournée globale. Enfin, chaque bloc est chargé de la mise à jour d'une ligne de la matrice de phéromone. Pour les noyaux qui demandent beaucoup d'accès mémoire, toutes les données telles que la distance, la phéromone et les listes des villes autorisées sont transférées dans la mémoire partagée. Des expérimentations ont été effectuées sur trois problèmes allant de 51 à 226 villes et une accélération maximale de 11 a été obtenue avec une qualité de solution similaire à celle de l'algorithme CPU. Weiss [Wei11] transforme également chaque phase de l'algorithme en noyau avec différentes configurations de threads et de blocs.

Cecilia *et al.* [CGU+11, CGN+12, CNA+12] parallélisent quant à eux deux phases de l'algorithme du AS qui sont la construction des tournées et la gestion de la phéromone. Un premier noyau effectue les calculs répétés des heuristiques. Les fonctions mathématiques spécialisées pour les GPU sont utilisées pour accélérer l'exécution de l'algorithme. Concernant le noyau de construction des tournées en particulier, les auteurs expliquent que l'approche traditionnelle où chaque fourmi est identifiée par un thread CUDA n'est pas bien adaptée aux GPU en raison du peu de threads utilisés. Dans ce noyau, chaque fourmi est donc associée à un bloc et les threads aux villes. Chaque thread charge la valeur heuristique associée à sa ville et génère un nombre aléatoire. Il vérifie ensuite si sa ville est contenue dans la liste taboue de sa fourmi. Cette liste contient les villes qu'elle a visitée dans l'ordre chronologique. Une divergence de threads est alors impliquée car seuls les threads dont la ville n'a pas déjà été visitée effectuent le calcul de règle de transition. Afin d'éviter cette divergence, la liste taboue est représentée par un entier par ville/thread stocké dans un registre : 0 si la ville a déjà été visitée, 1 sinon. Ainsi, chaque thread multiplie cette valeur avec sa valeur heuristique et le nombre aléatoire. Le résultat est stocké dans la mémoire partagée et une réduction parallèle est effectuée pour obtenir la prochaine ville à visiter. Cependant, cette implémentation n'est valable que lorsque le nombre de villes est inférieur au nombre de threads maximum par bloc. Pour les plus gros problèmes, les villes sont découpées en groupes distribués aux threads. Chaque groupe sélectionne de façon stochastique une ville non visitée puis la meilleure est choisie. La liste taboue est dans ce cas contenue dans un registre 32-bits avec un bit pour chaque ville. Concernant la phase de mise à jour de la phéromone, l'approche typique consiste à affecter un thread par ville de la tournée de chaque fourmi. Cependant, l'utilisation des instructions atomiques est essentielle puisque plusieurs fourmis peuvent avoir visité le même arc. Afin d'éviter ces instructions non supportées par tous les types de GPU, une nouvelle approche est proposée où chaque thread met à jour indépendamment une entrée de la matrice de phéromone. Afin d'accélérer l'algorithme, les tournées sont chargées dans la mémoire partagée. Quand elles ne peuvent plus être contenues dans cette mémoire, les tournées sont divisées en groupes chargés un par un. Des problèmes tests allant jusqu'à 2392 villes ont été résolus. La qualité des solutions obtenue est similaire ou légèrement meilleure à celle obtenue par la version séquentielle et chaque phase parallèle permet d'obtenir des accélérations de 21 et 20.

Certains auteurs intègrent une RL à leur algorithme OCF pour améliorer la qualité des solutions obtenues. Tsutsui et Fujimoto [TF11] combinent ainsi l'algorithme cAS avec une RT puis avec un algorithme 2-opt itéré. Le problème résolu est celui de l'affectation quadratique

et possède une taille du voisinage égale à $[n \cdot (n-1)]/2$ où n est la taille du problème. Toutes les données de l'algorithme sont stockées dans la mémoire globale du GPU et les matrices accessibles en lecture seule dans des textures. Chaque étape de l'algorithme séquentiel est transformée en noyau. Le premier construit les m solutions avec m blocs de 1 thread. Dans le second, la RL est appliquée aux m solutions avec m blocs dont les threads calculent les coûts des différents mouvements. Selon le type de mouvement, leurs coûts peuvent être calculés en $O(1)$ ou en $O(n)$, ce qui engendre une divergence des threads au sein d'un même warp. Les indices des threads sont alors gérés pour que chaque warp n'effectue que des calculs en $O(1)$ ou $O(n)$. Le troisième et dernier noyau consiste à mettre à jour les traces de phéromones. Il est composé de quatre sous-noyaux : initialisation des traces avec n blocs de n threads, évaporation avec n blocs de n threads, dépôt avec m blocs de 1 thread et ajustement selon les bornes minimale et maximale avec n blocs de n threads. Zhu et Curry [ZC09] intègrent également une RL (*Pattern Search*) en s'inspirant de leurs travaux sur les AEs [Zhu11]. Chaque thread correspond à une fourmi qui génère une nouvelle solution basée sur un vecteur de probabilités puis effectue la RL. Selon le type de problème et les différentes approches de parallélisation utilisées, les accélérations maximales varient entre 6,50 et 132,00 pour les algorithmes OCF décrits dans cette section.

Comme les métaheuristiques à solution unique, la littérature montre que les métaheuristiques à population de solutions peuvent exploiter efficacement le potentiel des architectures GPU. En effet, des accélérations maximales allant jusqu'à 2074,00 sont rapportées pour certains types de problèmes. De façon générale, la revue des travaux de parallélisation présentée dans cette section montre que l'émergence récente des GPU en tant que plateformes de calcul haute performance a suscité un intérêt certain dans le domaine des métaheuristiques. En effet, une classification des publications selon les années, illustrée à la Figure 2.7, met en évidence non seulement la jeunesse de ce domaine mais aussi la quantité non négligeable de travaux réalisés durant les toutes dernières années. Cependant, ces travaux sont essentiellement spécifiques à une famille ou à une métaheuristique en particulier et une vision globale et fondamentale demeure manquante. Certains auteurs ont néanmoins proposé des classifications ou taxonomies pour les métaheuristiques parallèles qui prennent en considération les architectures traditionnelles de type CPU. Ces classifications représentant un point de départ pertinent pour une étude globale des métaheuristiques parallèles sur GPU, elles sont décrites dans la prochaine section.

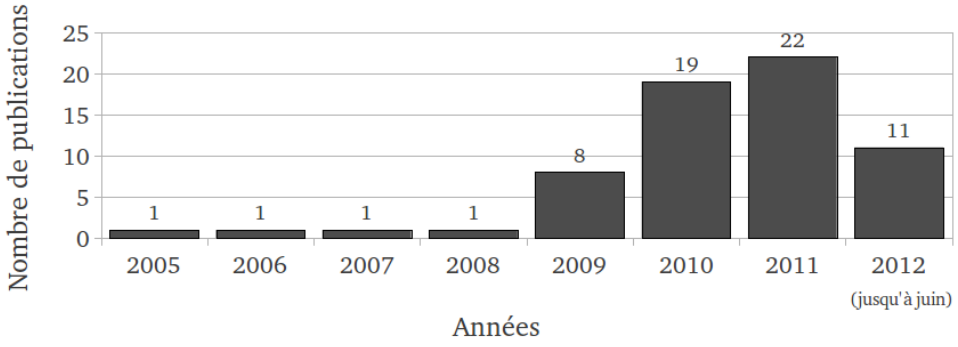


Figure 2.7 – Nombre de publications sur la parallélisation des métaheuristiques sur GPU.

2.4 Classification des métaheuristiques parallèles

La classification proposée par Crainic et Toulouse [CT10] fait partie des rares travaux apportant une vue globale et fondamentale aux métaheuristiques parallèles. Ces auteurs ont analysé les différentes stratégies et approches proposées dans la littérature pour en faire ressortir les points communs. La classification résultante est construite selon trois dimensions décrites dans la Figure 2.8 : cardinalité du contrôle de la recherche (*Search Control Cardinality*), contrôle de la recherche et communication (*Search Control and Communication*) et différentiation de la recherche (*Search Differentiation*).

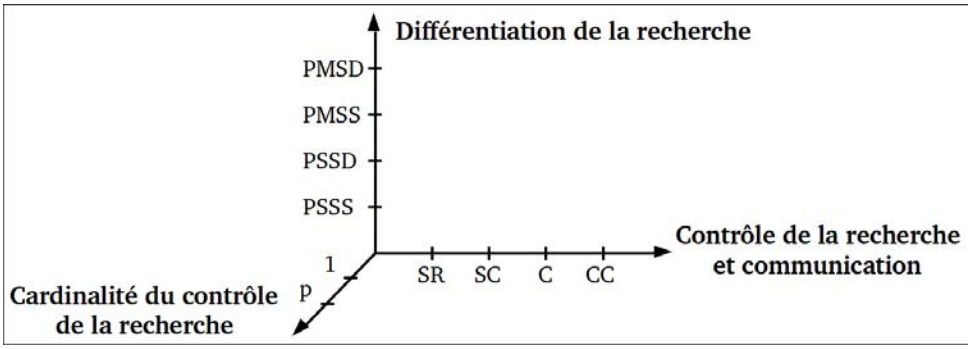


Figure 2.8 – Trois dimensions de la classification de Crainic [Cra05].

La première dimension, nommée cardinalité du contrôle de la recherche, spécifie si la recherche globale est contrôlée par un processus (*1-control, 1C*) ou par plusieurs processus (*p-control*, pC) pouvant communiquer entre eux. Dans le cas du $1C$, un processus maître exécute l'algorithme mais délègue à différents processus esclaves une partie de son travail, le plus souvent les calculs numériques coûteux en temps. Dans le cas du pC, la recherche globale est divisée afin que chaque processus soit en charge de sa propre recherche.

La seconde dimension, nommée contrôle de la recherche et communication, précise la manière dont l'information entre les processus est échangée. Les communications peuvent être de type synchrone ou asynchrone. Dans le premier cas, tous les processus doivent se coordonner afin de communiquer entre eux à des moments précis de l'algorithme (nombre d'itérations, intervalle dans le temps, etc.) déterminés par le maître. Deux classes sont définies : Synchronisation Rigide (SR) et Synchronisation de Connaissances (SC). La classe SR permet l'échange de peu ou pas d'informations et est idéalement complétée par l'approche $1C$ de la première dimension. Le processus maître possède l'information et communique avec les processus esclaves mais ces derniers ne peuvent pas échanger de connaissances entre eux. Dans l'approche pC de la première dimension, plusieurs trajectoires de recherche indépendantes sont effectuées sans qu'elles ne puissent communiquer entre elles. La classe SC amène un plus grand niveau de communication. Dans l'approche $1C$, le processus maître possède toujours l'information mais délègue une plus grosse quantité de travail nécessitant généralement des mémoires locales. Dans l'approche pC, un processus d'échange d'informations intensif est présent entre les trajectoires de recherche.

Dans le cas où les communications sont asynchrones, chaque processus est en charge de sa propre recherche sur une partie ou la totalité du domaine et initie les communications avec les autres processeurs selon sa propre logique et son statut. La recherche globale est terminée

lorsque tous les processus ont fini leur exécution. Deux classes sont définies : Collégiale (C) et Connaissances Collégiales (CC). La classe C consiste en une communication simple. Quand un processus trouve une solution améliorante, il l'envoie à plusieurs processus voisins ou à tous. Le message envoyé correspond alors toujours au message reçu, contrairement à la classe CC. Dans cette dernière, les communications sont analysées afin de mieux guider la trajectoire globale de la recherche. Le message reçu est donc plus riche en informations.

La troisième dimension, nommée différentiation de la recherche, est basée sur le fait que plus d'une méthode ou variante peut être intégrée à une métaheuristique. Les solutions initiales peuvent être différentes, ainsi que les stratégies employées (paramètres, etc.). Quatre cas sont considérés : Points/Populations initiaux Semblables, Stratégies de recherche Semblables (PSSS), Points/Populations initiaux Semblables, Stratégies de recherche Différentes (PSSD), Points/Populations initiaux Multiples, Stratégies de recherche Semblables (PMSS) et Points/Populations initiaux Multiples, Stratégies de recherche Différentes (PMSD).

À partir de cette classification et des différentes méthodes de parallélisation existantes dans la littérature, Crainic et Toulouse identifient plusieurs stratégies parallèles : $1C$ de bas niveau, décomposition du domaine et multiples recherches. Ces différentes stratégies sont décrites dans les prochaines sections. Pour chacune d'entre elles, des liens avec d'autres classifications de métaheuristiques parallèles sont effectuées, notamment celles de Verhoeven et Aarts [VA95] pour les algorithmes de RL et de Pedemonte *et al.* [PNC11] pour l'OCF.

2.4.1 Stratégies $1C$ de bas niveau

Les stratégies $1C$ de bas niveau exploitent le parallélisme intrinsèque des métaheuristiques, c'est-à-dire, les boucles intérieures de calcul. Leur but est d'accélérer l'algorithme séquentiel sans en modifier la logique et le comportement et non pas d'effectuer une meilleure exploration de la recherche. Typiquement, ces stratégies sont de type $1C/SR/PSSS$ où l'exploration débute avec une solution ou une population initiale. Le processus maître décompose alors les calculs impliqués dans les boucles intérieures et les distribuent aux processus esclaves qui les effectuent de façon parallèle et sans communication. Dans les métaheuristiques à solution unique, l'évaluation du voisinage de la solution courante est souvent effectuée en parallèle alors que dans celles à population de solutions, c'est bien souvent la fonction d'évaluation des individus qui est parallélisée.

Dans le cadre des métaheuristiques à solution unique, la classification des RL de Verhoeven et Aarts [VA95], qui a largement inspirée celle de Cung *et al.* [CMRC02], distingue deux types d'approches illustrées à la Figure 2.9. Les approches ajustées (*tailored approaches*) sont spécifiques à un problème et consistent généralement à paralléliser une partie précise de l'algorithme. Les approches générales, quant à elles, peuvent être appliquées à différents problèmes. Les stratégies $1C$ de bas niveau sont des approches générales caractérisées par un parallélisme marche-simple (*single-walk*), étape-simple (*single-step*). Une seule exécution de l'algorithme est effectuée dans laquelle une ou plusieurs étapes consécutives sont parallélisées. Les différents voisins de la solution courante sont généralement évalués en parallèle et une seule transformation est appliquée à la solution courante.

Dans le cadre des métaheuristiques à population de solutions, Cantù-Paz [CP98] a proposé une classification pour les AGs. Selon lui, une stratégie $1C$ de bas niveau correspond à

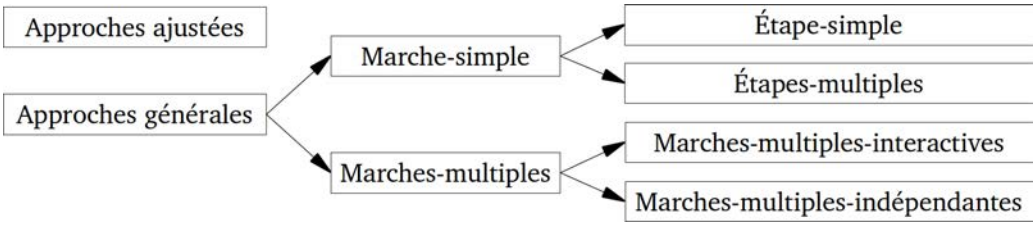


Figure 2.9 – Classification des algorithmes de Recherche Locale parallèles proposée par Verhoeven et Aarts [VA95].

une parallélisation globale nommée aussi parallélisation maître-esclaves. Elle consiste à n'avoir qu'une seule population exécutant le même algorithme. Seules certaines parties de ce dernier sont parallélisées afin de ne pas modifier son comportement par rapport à l'algorithme séquentiel. Un processus maître sépare les tâches à effectuer entre les différents processus esclaves afin qu'ils les exécutent en parallèle. La partie la plus souvent parallélisée est l'évaluation des fitness des individus car elles sont indépendantes les unes des autres et car la communication entre les différents processus n'y est pas nécessaire. Les opérateurs génétiques peuvent aussi être parallélisés mais les coûts de communications impliqués par rapport à la simplicité de cette phase peuvent pénaliser la performance de l'algorithme.

Pour les algorithmes OCF, une stratégie $1C$ de bas niveau correspond à l'approche des fourmis parallèles qui vise à exécuter la phase de construction des tournées des fourmis sur plusieurs unités de calculs. Les travaux liés à cette approche ont été initiés par Bullnheimer *et al.* [BKS97] qui ont proposé deux stratégies de parallélisation de l'algorithme OCF basique sur une architecture à passage de messages et à mémoire distribuée. La première est une stratégie synchrone de bas niveau qui a pour objectif d'accélérer les calculs en distribuant les fourmis aux processeurs d'une façon maître-esclaves. À chaque itération, le maître distribue la structure de phéromone aux esclaves qui calculent leurs tournées en parallèle puis qui les renvoient au maître. Selon la classification de Pedemonte *et al.* [PNC11] illustrée à la Figure 2.10, cette stratégie correspond aux modèles maître-esclaves à grain fin et à gros grain. Dans le modèle à grain fin, les esclaves effectuent des tâches de granularité minimum et communiquent fréquemment avec le maître contrairement au gros grain. Le temps nécessaire pour ces communications et synchronisations globales implique alors des coûts considérables. La seconde stratégie proposée par Bullnheimer *et al.* vise à les réduire en laissant l'algorithme effectuer un certain nombre d'itérations sans échanger d'informations. Les auteurs en concluent que cette stratégie partiellement asynchrone est préférable du fait de la réduction considérable des coûts de communication. Les travaux de Talbi *et al.* [TRFR01], Randall et Lewis [RL02], Islam *et al.* [ITT03], Craus et Rudeanu [CR04], Stützle [Stü98b] et Doerner *et al.* [DHBL06] sont basés sur une approche de parallélisation similaire et une architecture à mémoire distribuée. Delisle *et al.* [DKGG01, DGK+05b] ont implémenté cette approche sur des architectures à mémoire partagée telles que les ordinateurs SMP et les processeurs multicœur. Ils ont également comparé les performances entre les deux types d'architectures [DGK+05a].

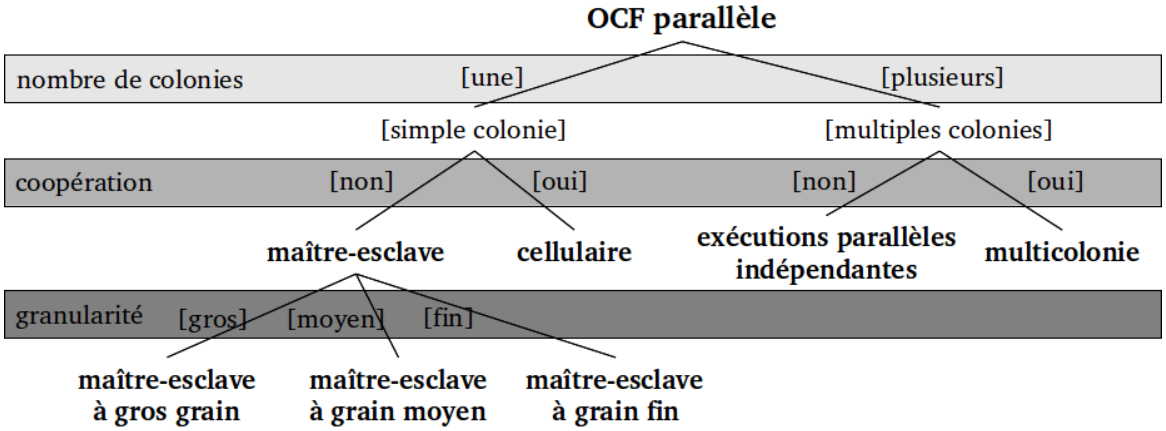


Figure 2.10 – Classification des algorithmes d'Optimisation par Colonie de Fourmis parallèles proposée par Pedemonte *et al.* [PNC11].

2.4.2 Stratégies basées sur une décomposition du domaine

Les stratégies basées sur une décomposition du domaine sont principalement de types $1C/SC/PMSS$ ou $1C/SC/PMSD$. Contrairement aux stratégies $1C$ de bas niveau, l'ensemble de solutions visitées est différent de l'algorithme séquentiel. Le processus maître est en charge de partitionner l'espace de recherche en sous-ensembles. Les esclaves explorent ensuite leur partition en parallèle : chaque sous-problème est résolu par une métaheuristique appliquée à chaque sous-ensemble. Le maître synchronise leur travail, collecte les solutions partielles et les assemble pour former la nouvelle solution courante. Les partitions peuvent alors être modifiées au cours de la recherche à des intervalles réguliers. La recherche est ensuite redémarrée à partir des nouvelles partitions.

Selon les classifications de Verhoeven et Aarts et de Cung *et al.*, ce type de stratégie correspond à une approche globale caractérisée par un parallélisme marche-simple, étapes-multiples (*multiple-step*). Un seul processus parcourt l'espace de recherche dans lequel plusieurs étapes consécutives sont parallélisées. Un certain nombre d'échanges simultanés sont alors effectués sur différentes parties de la solution courante. Dans la même optique, Johnson et McGeoch [JM97] définissent deux stratégies de parallélisation pour les algorithmes de RL k-opt appliqués au problème du Voyageur de Commerce. La première utilise un partitionnement géométrique pour diviser l'ensemble des villes en groupes envoyés à différents processeurs. Pour chaque groupe, un algorithme de construction produit une sous-tournée améliorée par une procédure de RL. Les sous-tournées optimales sont ensuite réunies pour former une tournée contenant toutes les villes. Comme ce partitionnement possède le désavantage d'isoler les sous-groupes et de ne pas reconnecter les sous-tournées intelligemment, la seconde stratégie favorise le partitionnement qui divise les tournées en solutions partielles contenant une partie des arcs de la solution courante.

En ce qui concerne la classification de Pedemonte *et al.*, cette stratégie correspond au modèle maître-esclaves à grain moyen. Le domaine du problème est décomposé et chaque esclave résout un sous-problème indépendamment. Le maître gère quant à lui l'information globale du problème et construit la solution complète à partir des solutions partielles des esclaves.

2.4.3 Stratégies reposant sur de multiples recherches

Les stratégies reposant sur de multiples recherches réalisent plusieurs marches en parallèle dans l'espace des solutions. Leur but est d'accélérer les métaheuristiques mais aussi d'améliorer la qualité des solutions obtenues par une exploration de l'espace de recherche plus approfondie.

Les stratégies reposant sur de multiples recherches indépendantes sont principalement de types $pC/SR/PMSS$ ou $pC/SR/PMSD$. Elles apportent généralement des performances intéressantes tout en étant simples à implémenter puisqu'aucun échange d'information n'est effectué. Elles ont basées sur l'exécution de plusieurs recherches parallèles indépendantes sur la totalité de l'espace des solutions. Parmi tous les résultats des recherches, la meilleure solution est ensuite récupérée.

Selon la classification de Verhoeven et Aarts, ce type de parallélisme est nommé marches-multiples-indépendantes (*multiple independent walks*) et selon celle de Cung *et al.*, threads de recherche indépendants (*independent search threads*). Pour la Recherche Locale Itérée (RLI), Stützle [Stü98a] utilise une approche à base de population afin d'éviter la stagnation de la recherche et propose une stratégie nommée "pas-d'interaction" (*no interaction*). Elle utilise une population de p membres et autorise un temps d'exécution maximum t_{max}. La RLI est alors appliquée à chaque membre de façon indépendante durant un temps $\frac{t_{max}}{p}$.

Pour les algorithmes OCF, cette stratégie a été introduite par Stützle [Stü98b] avec les approches par colonies multiples. Ces dernières sont nommées *runs* indépendants parallèles par Pedemonte *et al.* Cette stratégie vise à exécuter des colonies de fourmis entières sur les unités de calcul disponibles et est basée sur une architecture à passage de messages et à mémoire distribuée. Plusieurs copies indépendantes du même algorithme y sont exécutées en parallèle, chacune représentant une colonie de fourmis.

Les stratégies reposant sur de multiples recherches coopératives sont en général de types pC/SC, pC/C ou pC/CC avec comme différentiation de la recherche $PSSD$, $PMSS$ ou $PMSD$. Elles intègrent des mécanismes permettant de partager de l'information sur les diverses explorations effectuées durant la recherche de façon à améliorer la qualité des solutions. L'information échangée concerne bien souvent les "bonnes" solutions (meilleure solution courante, meilleure solution globale, informations de contexte, etc.). Les processeurs peuvent échanger de l'information directement entre eux ou indirectement par l'intermédiaire de structures de données globales selon un graphe de communication : chaque nœud représente un processus et les arcs, les paires de processus pouvant communiquer directement entre eux. De façon générale, l'information ne doit pas être échangée trop fréquemment afin d'éviter des coûts de communication trop importants ainsi que la convergence prématurée vers les optima locaux.

Verhoeven et Aarts nomment ce type de parallélisation marches-multiples-interactives (*multiple interacting walks*) et Cung *et al.*, threads de recherche coopératifs (*cooperative search threads*). Pour la RLI, Stützle [Stü98a] utilise une approche à base de population nommée "remplacement-de-la-plus-mauvaise-solution" (*replace-worst*) qui fonctionne de la même manière que l'approche "pas-d'interaction". Cependant, elle remplace la plus mauvaise solution par la meilleure à des intervalles réguliers. Martin et Otto [MO96] ont, quant à eux, proposé une implémentation dans laquelle plusieurs solutions sont calculées simultanément sur différents processeurs et la meilleure solution remplace (rarement) toutes les solutions à des intervalles irreguliers. Hong *et al.* [HKM97] ont conçu un algorithme de RLI parallèle où une population

de taille p doit effectuer un total de m itérations, soit i itérations par solution. Les solutions peuvent ensuite interagir entre elles afin d'échanger de l'information. Selon la valeur de i, la stratégie varie. Si $i = \infty$, elle repose sur de multiples recherches coopératives avec p exécutions indépendantes de m/p itérations chacune. Sinon, elle repose sur de multiples recherches indépendantes. Comme le résument Hoos et Stützle [HS04], les travaux de parallélisation de la RLI pour le problème du Voyageur de Commerce suivent principalement une approche à base de population, reposant sur de multiples recherches. L'interaction entre ses membres doit être limitée ou inexistante afin de faciliter la parallélisation de la RLI. Ils en concluent même que les variantes sans interactions trouvent souvent la même qualité de solution que les variantes avec interactions.

Selon la classification de Cantù-Paz, cette stratégie équivaut aux parallélisations à grain épais (ou modèle en îles) et à grain fin. Celle à grain épais consiste à diviser la population en plusieurs sous-populations d'assez grande taille. Chaque sous-population exécute son propre AG et échange de temps en temps ses solutions avec les autres (processus de migration). Dans la parallélisation à grain fin, la taille de ces dernières des sous-populations doit être la plus petite possible. Le cas idéal correspond à un individu par élément de calcul ce qui implique une communication très fréquente. Cette parallélisation est particulièrement adaptée aux architectures massivement parallèles.

Pour les algorithmes OCF, ce principe a été introduit par Middendorf *et al.* [MRS02] qui proposent quatre stratégies d'échange d'informations entre les colonies de fourmis : échange de la meilleure solution globale, échange circulaire des meilleures solutions locales, des immigrés ou des meilleures solutions en plus des immigrés. Il est montré qu'il peut être avantageux pour les colonies d'éviter de communiquer trop d'informations trop souvent. Renoncant à l'idée de partager l'information de phéromone, ils ont fondé leur stratégie sur le transfert d'une solution unique à chaque étape d'échange. Selon la classification de Pedemonte *et al.*, cette stratégie correspond au modèle multi-colonie où plusieurs colonies explorent l'espace de recherche en utilisant leurs propres matrices de phéromone et en échangeant périodiquement des informations. Selon ce même auteur, les stratégies reposant sur de multiples recherches coopératives peuvent également correspondre au modèle cellulaire où une seule colonie est structurée en petits voisinages se chevauchant pour permettre une diffusion des informations. Chacun possède sa propre matrice de phéromone qui est mise à jour en ne considérant que les solutions construites par les fourmis dans son voisinage. Chu *et al.* [CTZ05], Manfrin *et al.* [MBSD06], Ellabib *et al.* [ECB07] et Alba *et al.* [ALO07] ont également proposé différentes stratégies d'échange d'informations. Beaucoup de paramètres ont été étudiés comme la topologie des liens entre les processeurs ou la nature et la fréquence des échange d'information. Ces stratégies ont été implémentées sur des architectures à mémoire distribuée. D'autre part, Delisle *et al.* [DGK09] ont adapté certaines d'entre elles sur architectures à mémoire partagée.

2.5 Objectifs de la recherche

Trois constats importants peuvent être retenus de l'ensemble des travaux présentés dans ce chapitre. Premièrement, l'utilisation du parallélisme massif sur GPU permet d'améliorer la performance des métaheuristiques. Le temps de résolution de ces méthodes peut ainsi être ré-

duit jusqu'à 2000 fois. Il devient néanmoins important de délivrer des méthodes d'optimisation qui sont compétitives autant au niveau de la qualité des solutions obtenues qu'à celui du temps de calcul. Les travaux recensés dans la littérature ont montré non seulement que cet objectif était souvent difficile à atteindre, mais aussi que les caractéristiques et facteurs de performance essentiels de ces méthodes étaient encore mal compris. Il reste donc encore beaucoup de travail conceptuel, technique et comparatif à accomplir dans le but d'exploiter efficacement cette architecture abordable et massivement parallèle pour l'optimisation combinatoire.

Deuxièmement, la plupart des travaux existants traitent d'une métaheuristique en particulier sans apporter une vision globale et fondamentale des métaheuristiques parallèles sur GPU. En effet, les classifications existantes proposent une telle vision pour les architectures traditionnelles de calcul, mais ne tiennent pas compte des caractéristiques spécifiques des GPU tels que les différentes catégories de mémoire et les granularités de parallélisation multiples. Comme ces caractéristiques sont déterminantes sur la performance et même sur la faisabilité d'une métaheuristique implémentée sur GPU, il devient nécessaire de proposer des modèles qui les intègrent.

Troisièmement, les travaux de parallélisation sur GPU existants présentent souvent des points faibles qui laissent des questions importantes sans réponse. Tout d'abord, les accélérations sont couramment fournies avec une évaluation inexistante ou inappropriée de la qualité des solutions ainsi que des paramètres irréalistes ou peu communs. De plus, les méthodes sont généralement simplifiées pour s'adapter aux GPU et deviennent donc non compétitives avec la littérature. Les stratégies sont également expérimentées sur de petites instances de problèmes sans dépasser les limites associées aux mémoires des GPU actuels. L'influence de l'utilisation des différentes mémoires du GPU sur les performances des métaheuristiques n'est d'ailleurs pas encore bien comprise. Enfin, un manque de détails d'implémentation empêche souvent la reproductibilité des stratégies de parallélisation.

Le premier objectif de cette thèse est de proposer une taxonomie originale des métaheuristiques parallèles implémentées sur des architectures basées sur les GPU. Elle vise à formaliser et à structurer les différentes approches de parallélisation possibles en fonction des caractéristiques fondamentales des GPU. Elle fournira ainsi un cadre méthodologique général cohérent facilitant l'implémentation des métaheuristiques sur ces architectures.

Le second objectif de cette thèse vise à valider cette taxonomie dans un contexte appliqué de conception et de développement de méthodes d'optimisation parallèles compétitives. Plusieurs stratégies de parallélisation originales spécifiquement adaptées à l'architecture GPU sont proposées à partir des composantes clés de cette taxonomie. Afin de montrer leur validité et leur généricité, elles sont développées dans des contextes de résolution par métaheuristiques à solution unique, métaheuristiques à population de solutions et méthodes de Recherche Locale. Plus spécifiquement, plusieurs implémentations performantes basées sur les métaheuristiques d'Optimisation par Colonie de Fourmis et de Recherche Locale Itérée sont proposées. Le choix de ces deux méthodes particulières a été motivé par le fait que leur parallélisation GPU est moins étudiée que celle des autres méthodes dans la littérature et qu'elles permettent de couvrir des caractéristiques communes à un grand nombre de méthodes. La performance des implémentations proposées permettant d'établir la validité des stratégies de parallélisation et des points de repère fournis par la taxonomie, une étude expérimentale minutieuse mettra constamment en opposition la réduction du temps de calcul, la qualité des solutions trouvées, le respect du

comportement de la méthode séquentielle originale et les limitations technologiques de l'architecture GPU.

Les objectifs de recherche de cette thèse ont été définis de sorte à apporter une contribution dans le domaine des métaheuristiques parallèles autant au niveau conceptuel que technique. Les prochains chapitres visent à répondre à chacun de ces objectifs tout en décrivant en détail le cheminement parcouru.

Chapitre 3

Taxonomie des métaheuristiques parallèles sur GPU

Les taxonomies et classifications existantes des métaheuristiques parallèles apportent une bonne vue d'ensemble des approches de parallélisation à un niveau d'abstraction relativement élevé. Ce dernier n'est toutefois pas assez précis pour caractériser efficacement les approches de parallélisation basées sur les architectures GPU. En effet, il n'intègre pas leurs caractéristiques spécifiques comme les différents types de mémoire et les granularités de parallélisation multiples qu'il est possible d'exploiter. Par exemple, la première dimension de la classification de Crainic et Toulouse [CT10] spécifie le nombre de processus contrôlant la recherche. Cependant, cette information n'est pas suffisante en raison de l'organisation hiérarchique des éléments de calcul pouvant être associés à l'exécution d'une tâche sur un GPU. La première dimension "élément de calcul" de la taxonomie proposée dans ce chapitre permet donc de préciser quelles composantes de la métaheuristique sont associées aux différents niveaux de parallélisme : GPU/grille, multiprocesseur/bloc et processeur/thread. Cette dimension, présentée dans la première section, peut donc compléter celle de Crainic et Toulouse. Une autre caractéristique importante de ce type d'architecture est la diversité des structures de mémoire qui peuvent avoir une grande influence sur l'efficacité d'une implémentation. La seconde dimension proposée, nommée "mémoire", spécifie donc le type de mémoire du GPU utilisée pour stocker les différentes données nécessaires au fonctionnement de la métaheuristique. Elle est détaillée dans la deuxième section. À partir de ces dimensions, plusieurs stratégies de parallélisation sont ensuite formalisées et mises en relation avec les différentes approches retrouvées dans la littérature.

3.1 Première dimension : élément de calcul

Une notion importante dans le domaine des métaheuristiques parallèles est celle d'élément de calcul (*processing element*). Le travail de la métaheuristique est réparti sur plusieurs de ces éléments fonctionnant en parallèle. Dans le contexte de la parallélisation CPU, ce terme peut être associé à un CPU monocœur ou à un cœur d'un processeur multicœur. La parallélisation GPU est plus complexe car cette architecture est composée de plusieurs multiprocesseurs qui fonctionnent en parallèle et qui incluent différents processeurs. La notion d'élément de calcul

peut donc être associée à un processeur ou à un multiprocesseur. Au niveau de la conception et de la programmation, les processeurs exécutent des threads et les multiprocesseurs exécutent des blocs. Deux niveaux de parallélisation peuvent donc être observés. De plus, lorsque plusieurs GPU sont regroupés au sein d'un cluster, un troisième niveau de parallélisation est constaté et les GPU peuvent alors exécuter des grilles spécifiques. Ces trois niveaux sont explicités dans la Figure 3.1 : niveau I - GPU, niveau II - bloc et niveau III - thread. Ainsi, les différents threads CUDA évoluent simultanément sur les processeurs et sont regroupés en blocs CUDA exécutés en parallèle sur les multiprocesseurs. Lorsqu'un cluster de plusieurs GPU est utilisé, des processus CPU sont créés et lancent leur grille de blocs sur leur GPU associé.

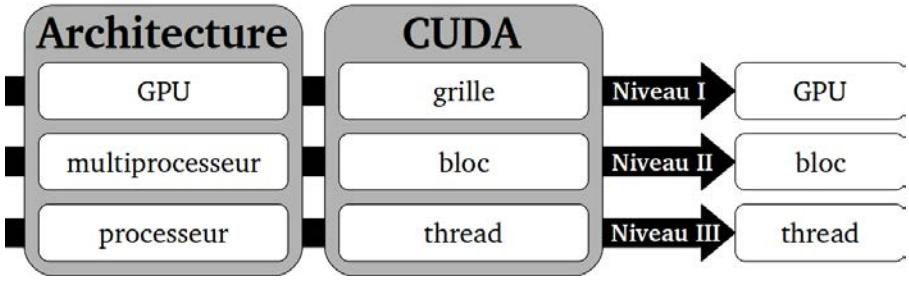


Figure 3.1 – Niveaux de parallélisation du GPU et du modèle de programmation CUDA.

Les métaheuristiques parallèles doivent s'adapter à cette architecture pour maximiser l'utilisation de sa puissance de calcul. L'étude de ces méthodes a permis de montrer qu'elles pouvaient également être parallélisées selon trois niveaux décrits dans la Figure 3.2. Les AE consistent en une population ou plusieurs sous-populations d'individus (ou chromosomes) qui contiennent des gènes (ou variables). Les algorithmes OEP manipulent un ou plusieurs essaims de particules qui peuvent être multi-dimensionnelles. Les algorithmes OCF gèrent une ou plusieurs colonies de fourmis où chaque fourmi doit construire sa tournée en utilisant une règle de transition d'état. Les métaheuristiques à solution unique (RS, RT et RLI) manipulent une solution et ses différents voisins. Toutefois, ces méthodes peuvent également employer une stratégie qui repose sur de multiples recherches. Ainsi, une population de solutions est générée où chacune d'entre elles est améliorée par une RL en parallèle. Les trois niveaux de parallélisation des métaheuristiques sont donc résumés en : niveau 1 - population, niveau 2 - solution et niveau 3 - élément.

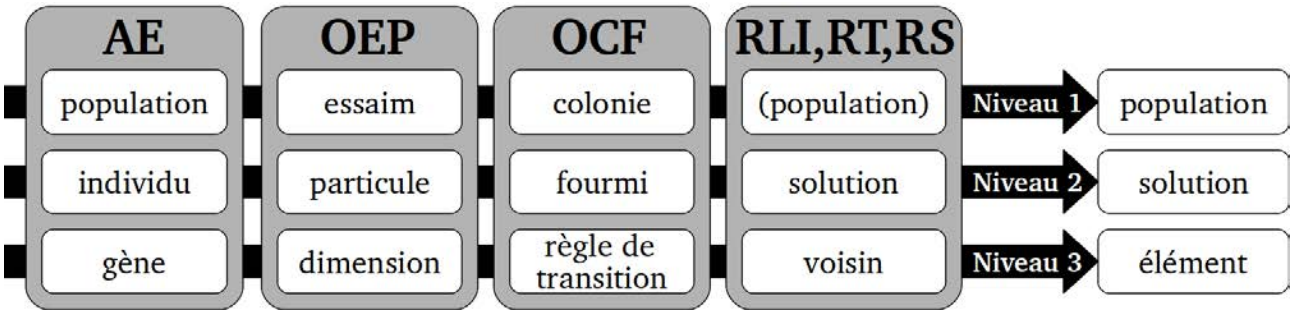


Figure 3.2 – Niveaux de parallélisation des métaheuristiques.

À partir des niveaux de parallélisation du GPU et des métaheuristiques, différentes combinaisons peuvent être obtenues dont les plus courantes sont présentées dans la Figure 3.3. Une population est associée à un certain nombre de GPU ou de blocs. Une solution peut, quant à elle, être assignée à chacun des trois niveaux de parallélisation du GPU : GPU(s), bloc(s) ou thread(s). Enfin, les éléments sont associés aux blocs ou aux threads. Plusieurs stratégies de parallélisation GPU peuvent ainsi naître de ces combinaisons. Par exemple, une approche possible consisterait à assigner la population au GPU, les solutions aux blocs et les éléments aux threads. La taxonomie de ces différentes stratégies de parallélisation permet ainsi d'enrichir la classification de Crainic et Toulouse. Pour rappel, elle compte trois dimensions : la première spécifie le nombre de processus contrôlant la recherche, la seconde précise la manière dont l'information est échangée entre les processus et la dernière est spécialisée dans la différentiation de la recherche. La première dimension peut donc être complétée par la dimension proposée "élément de calcul".

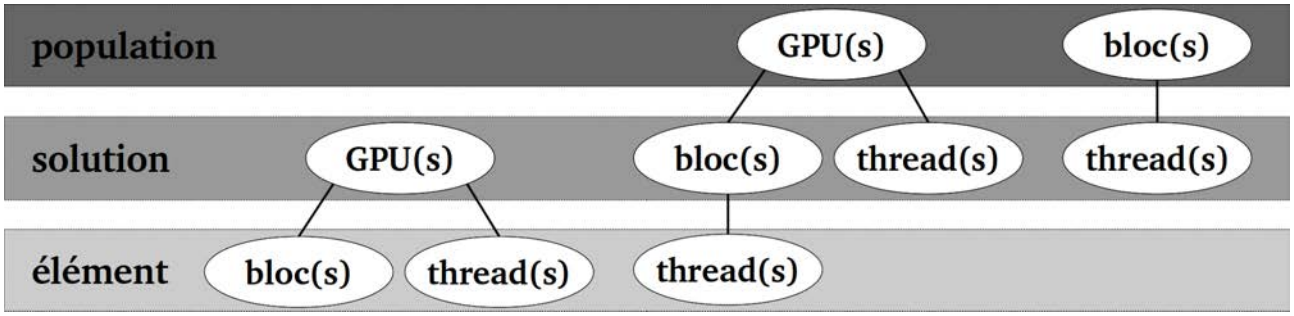


Figure 3.3 – Taxonomie des métaheuristiques parallèles sur GPU.

Pour chaque niveau de parallélisation des métaheuristiques, les différentes combinaisons qui peuvent être obtenues sont décrites tout en faisant un lien avec les trois stratégies de parallélisation de Crainic et Toulouse. Par souci de simplicité, la stratégie 1C de bas niveau est renommée "stratégie 1C-BN", celle basée sur une décomposition du domaine est renommée "stratégie DD" et celle reposant sur de multiples recherches est renommée "stratégie MR".

Niveau 1 - population

Ce niveau donne naissance à trois combinaisons : pop_{gpu}, pop_{bl} et pop_{blS}.

***pop*$_{gpu}$** associe une population (ou sous-population) à un GPU. Le nombre de populations utilisées dans l'algorithme est donc limité par le nombre de GPU disponibles. Dans la stratégie 1C-BN, le GPU décompose les calculs et les distribue aux esclaves (blocs et/ou threads). Dans la stratégie DD, le GPU partitionne l'espace de recherche en sous-ensembles et les esclaves (blocs et/ou threads) résolvent les sous-problèmes en parallèle. La stratégie MR effectue de multiples recherches dans l'espace des solutions. Elle divise donc bien souvent la population en sous-populations pouvant chacune être dirigée par un GPU. La combinaison pop_{gpuS} n'est pas mentionnée car elle consiste à répartir la population sur plusieurs GPU, ce qui est équivalent à la combinaison pop_{gpu} avec une sous-population par GPU.

***pop**$_{bl}$* assigne une population à un bloc. Néanmoins, utiliser une seule population/bloc ne permet pas d'exploiter au maximum les capacités du GPU qui peut exécuter un grand nombre de blocs à la fois (typiquement 65535 blocs pour une grille 1D). Il est donc préférable de diviser la population en un grand nombre de groupes. Le nombre de sous-populations est alors limité au nombre maximum de blocs qui peuvent être lancés. Lorsque la population est assignée au bloc, une solution ne peut plus être associée qu'à un ou plusieurs threads qui correspondent aux esclaves des stratégies 1C-BN et DD ou aux différentes recherches dans la stratégie MR.

***pop**$_{blS}$* associe une population à plusieurs blocs. Elle permet à chaque population d'obtenir plus de ressources (nombre de threads, quantité de mémoire partagée, etc.). Cependant, plusieurs populations doivent être gérées sur le GPU sinon la combinaison employée est plutôt pop_{gpu}.

Niveau 2 - solution

Ce niveau donne naissance à six combinaisons : sol_{gpu}, sol_{gpuS}, sol_{bl}, sol_{blS}, sol_{th} et sol_{thS}.

***sol**$_{gpu}$* associe une solution à un GPU. Elle est surtout utilisée par les métaheuristiques à solution unique où les différents voisins sont évalués par les blocs ou par les threads. Les métaheuristiques à population de solutions contiennent bien souvent au moins une dizaine d'individus/solutions ce qui signifie qu'un cluster d'une dizaine de GPU, non accessible à tous les scientifiques, serait nécessaire. Lorsque plusieurs solutions sont calculées, leur nombre est donc limité par le nombre de GPU disponibles.

***sol**$_{gpuS}$* permet de diviser les éléments sur plusieurs GPU. Cette combinaison peut être bénéfique pour les métaheuristiques à grand voisinage qui nécessitent plus de ressources, par exemple.

***sol**$_{bl}$* assigne la solution à un bloc. Les éléments sont donc calculés par les threads et la population est gérée par le GPU. Les blocs correspondent alors aux esclaves des stratégies 1C-BN et DD et aux recherches de la stratégie MR. Cependant, le nombre de solutions est limité au nombre maximum de blocs autorisés par grille.

***sol**$_{blS}$* associe la solution à plusieurs blocs. Elle permet donc d'obtenir plus de ressources par solution.

***sol**$_{th}$* assigne la solution à un thread. Les éléments de chacun sont donc calculés "de façon séquentielle" puisque cette combinaison est semblable aux stratégies de parallélisation CPU où chaque solution est associée à un processeur ou à un cœur. Les threads sont donc les esclaves des stratégies 1C-BN et DD et les recherches de la stratégie MR. Cette combinaison est efficace si l'algorithme contient un grand nombre de solutions afin de tirer profit des capacités du GPU qui peut exécuter des milliers de threads en parallèle.

***sol**$_{thS}$* assigne la solution à plusieurs threads. Cependant, plusieurs populations doivent être gérées par le bloc sinon la combinaison utilisée est sol_{bl}.

Niveau 3 - élément

Ce niveau donne naissance à quatre combinaisons : elt_{bl}, elt_{blS}, elt_{th} et elt_{thS}.

$\boldsymbol{elt_{bl}}$ associe un élément à un bloc. Cette combinaison peut être utilisée si un quatrième niveau de parallélisation peut être défini au sein de la métaheuristique. Dans ce cas, le calcul de l'élément est réparti entre les différents threads. Un seul thread peut également s'occuper du calcul mais disposer de toutes les ressources du bloc disponibles (mémoire partagée, etc.) pour lui seul. Le produit du nombre d'éléments et du nombre de solutions doit être inférieur au nombre maximum de blocs ou bien chaque bloc doit travailler sur plusieurs éléments à la fois.

$\boldsymbol{elt_{blS}}$ assigne le calcul d'un élément à plusieurs blocs.

$\boldsymbol{elt_{th}}$ associe chaque élément à un thread. Si la solution est assignée au bloc, le nombre d'éléments doit être inférieur au nombre maximum de threads autorisés dans un bloc ou bien chaque thread doit gérer plusieurs éléments à la fois.

$\boldsymbol{elt_{thS}}$ associe chaque élément à plusieurs threads. Cependant, plusieurs éléments doivent être calculés par bloc sinon la combinaison utilisée est elt_{bl}.

3.1.1 Stratégies de parallélisation

À partir de ces combinaisons, un grand nombre de stratégies de type $pop_x sol_y elt_z$ peuvent être créées. Afin d'illustrer cette taxonomie, les Tableaux 3.1 et 3.2 regroupent les différentes stratégies de parallélisation sur GPU employées dans la littérature. Certains travaux ont été écartés de cette taxonomie car la stratégie utilisée n'était pas assez précise.

Référence	Méthode	Stratégie	Description
[LMT10b, LMT11b]	RL / RT	$sol_{gpu}elt_{th}$	
[LLMT10]	RL / RT itérée		
[LTLL10]	RS		
[CB11]	RT		
[LMT10c]	AE + RL	$pop_{gpu}elt_{th}$	AE sur CPU, RL sur GPU
[LMT11a]	RL / RT / RS	$pop_{gpu}sol_{th}$	multiples relances
[OTB11]	RL		multiples relances
[ZC09]	OCF + RL		
[Zhu11]	SE + RL		
[MKQ+12]	AE / AG (+ RL)		
[LMT10a]	AE		
[RCCRT11]	ED		
[PDB12]	AGc		
[SBPE10]	AGc		
[VA10]	AGc		une seule population
[VA10]	AGc		une sous-population par GPU
[CBFN11]	OEP		noyau recherchant les meilleurs voisins
[WWW11]	OEP		
[ZT09]	OEP		
[LSOCCC+10]	OEP		

Tableau 3.1 – Classification des stratégies de parallélisation GPU utilisées dans la littérature selon la dimension "élément de calcul" (partie 1).

Référence	Méthode	Stratégie	Description
[SSW10]	RS	$pop_{gpu}sol_{bl}elt_{th}$	multiples relances
[CBZ10]	RS		les blocs exécutent le même algorithme sur des sous-ensembles différents
[HHL12]	AG		
[FT10]	AG + RL		
[KPSA11]	AG / ED		
[MNC11]	OEP asynchrone		
[MWMA09]	AGc + RL	$pop_{bl}sol_{th}$	une sous-population par bloc
[FZY10]	AG		une sous-population par bloc
[TF09]	AG		une sous-population par bloc
[PJS10]	AG		une sous-population par bloc
[CDJN11]	AG		un seul bloc utilisé
[MDC11]	OEP		un essaim associé à un bloc ($SyncPSO$)

Tableau 3.2 – Classification des stratégies de parallélisation GPU utilisées dans la littérature selon la dimension "élément de calcul" (partie 2).

Malgré le grand nombre de stratégies de parallélisation GPU pouvant être définies, seules cinq ont été utilisées dans la littérature. La plus employée, $pop_{gpu}sol_{th}$, est celle qui est semblable aux stratégies de parallélisation CPU. Le GPU gère la population et les esclaves/threads, les solutions. Cette stratégie est également employée pour une implémentation multi-GPU où chaque sous-population est gérée par un GPU. Cependant, lorsque plusieurs sous-populations évoluent en parallèle, il est rare de posséder un cluster dont le nombre de GPU est supérieur ou égal au nombre de sous-populations. Dans ce cas, c'est donc bien souvent la stratégie $pop_{bl}sol_{th}$ qui est employée. Les populations sont alors structurées au sein des blocs et possèdent leurs propres ressources. La stratégie $pop_{gpu}sol_{bl}elt_{th}$ est également très employée puisqu'elle permet d'obtenir trois niveaux de parallélisation distincts. La stratégie $sol_{gpu}elt_{th}$ associe la solution unique au GPU afin que les différents threads génèrent et évaluent les voisins dans le cadre des RL et des métaheuristiques RT et RS. Une dernière stratégie $pop_{gpu}elt_{th}$ est employée lorsque un AE est hybridé avec une RL mais où seule cette dernière est parallélisée. Le CPU dirige alors la population d'individus et le GPU gère la création et l'évaluation des voisins de ces différents individus.

3.1.2 Stratégies de parallélisation hybrides

La taxonomie proposée dans la Section 3.1.1 permet de classifier les travaux de la littérature qui utilisent un seul noyau ou la même approche de parallélisation pour tous les noyaux. Cependant, certains auteurs décomposent leurs algorithmes en plusieurs noyaux et ajustent les configurations gpu/bloc/thread afin de mieux exploiter le parallélisme du GPU. Chaque noyau suit ainsi une stratégie de parallélisation différente des autres qui est adaptée à son fonctionnement. Les Tableaux 3.3 et 3.4 décrivent les différentes stratégies de parallélisation hybrides décrites dans la littérature. Certains travaux ne sont pas mentionnés car la stratégie utilisée n'était pas assez précise.

Lorsque la stratégie employée n'a pas un lien direct avec les niveaux population, solution et élément de la métaheuristique, des correspondances entre ces niveaux de parallélisation et

ceux du noyau sont effectuées. Par exemple, le noyau de dépôt de phéromone de Li *et al.* [LHPQ09] consiste à mettre à jour une matrice de taille $n \cdot n$ où n blocs sont utilisés, chacun mettant à jour une ligne. La matrice correspond alors à la population et est associée au GPU. Les lignes associées aux blocs représentent les solutions et les différents éléments de chaque ligne sont affectés aux threads. La stratégie employée pour ce noyau est donc de type $pop_{gpu}sol_{bl}elt_{th}$.

Référence	Méthode	Stratégie	Description
[LHPQ09]	OCF	$pop_{gpu}sol_{bl}elt_{th}$	construction des tournées
		$pop_{gpu}sol_{bl}elt_{th}$	dépôt de phéromone (matrice $n \cdot n$) : n blocs, chacun mettant à jour une ligne
		$pop_{bl}elt_{th}$	mise à jour de la matrice de phéromone (un seul bloc)
[Wei11]	OCF	$pop_{gpu}elt_{th}$	initialisation, dépôt et évaporation des traces de phéromone : un thread par trace
		$pop_{gpu}sol_{bl}elt_{th}$	calcul des valeurs heuristiques : un bloc par nœud, un thread par probabilité
		$pop_{gpu}sol_{th}$	construction des solutions : une fourmi associée à un thread
		$pop_{gpu}sol_{bl}elt_{th}$	évaluation des solutions : une fourmi associée à un bloc
[CGN+12]	OCF	$pop_{gpu}sol_{bl}elt_{th}$	construction des tournées
		$pop_{gpu}elt_{th}$	mise à jour de la phéromone : chaque thread met à jour une entrée de la matrice
[TF11]	OCF + RL	$pop_{gpu}sol_{bl}elt_{th}$	construction des tournées : un bloc de un thread par solution
		$pop_{gpu}sol_{bl}elt_{th}$	RL
		$pop_{gpu}sol_{bl}elt_{th}$	initialisation des traces de phéromone : n blocs de n threads
		$pop_{gpu}sol_{bl}elt_{th}$	évaporation de la phéromone : n blocs de n threads
		$pop_{gpu}sol_{bl}elt_{th}$	dépôt de phéromone : m blocs de 1 thread
		$pop_{gpu}sol_{bl}elt_{th}$	ajustement selon les bornes : n blocs de n threads
[MDC11]	OEP	$pop_{gpu}sol_{bl}elt_{th}$	mise à jour des données des particules + évaluation des fitness (*RingPSO*) : une particule par bloc
		$pop_{bl}sol_{th}$	mise à jour des fitness (*RingPSO*) : un essaim par bloc
[STT11]	OEP	$pop_{gpu}elt_{th}$	initialisation des particules + mise à jour de leurs données : un thread par dimension de chaque particule
		$pop_{gpu}sol_{th}$	mise à jour des fitness : une particule associée à un thread
		$pop_{bl}sol_{th}$	mise à jour des meilleures valeurs + détermination des meilleures et plus mauvaises particules : un essaim par bloc, un thread par particule
		$pop_{gpu}elt_{th}$	remplacement des mauvaises particules : un thread par dimension de chaque particule
[OMYO11]	AG	$pop_{gpu}sol_{th}$	sélection par tournoi : nombre de threads = taille de la population
		$pop_{gpu}sol_{bl}elt_{th}$	croisement : un croisement de deux individus par bloc
		$pop_{gpu}sol_{bl}elt_{th}$	reste de l'algorithme : un individu par bloc

Tableau 3.3 – Classification des stratégies de parallélisation GPU hybrides utilisées dans la littérature selon la dimension "élément de calcul" (partie 1).

Référence	Méthode	Stratégie	Description
[PAL11]	AG	$pop_{gpu}sol_{th}$	tournoi binaire + mutation
		$pop_{gpu}sol_{bl}elt_{th}$	croisement : nombre de blocs égal à la moitié de la taille de population
		$pop_{gpu}sol_{bl}elt_{th}$	évaluation : nombre de blocs égal à la taille de population
[ATD10]	AG	$pop_{bl}elt_{th}$	initialisation des individus : chaque bloc contient une partie de la population, les threads initialisant les gènes
		$pop_{bl}sol_{th}$	meilleur individu : assumant que la taille minimum d'une population est 32, le nombre de blocs est égal à la taille de la population divisée par 32 et le nombre de threads est égal à 32
		$pop_{gpu}sol_{bl}elt_{th}$	croisement binaire : nombre de blocs égal à la moitié de la taille de population
		$pop_{bl}elt_{th}$	croisement réel : chaque bloc gère une partie de la population et une partie des variables, chaque thread est associé au croisement d'un gène entre deux individus
		$pop_{gpu}sol_{th}$	mutation binaire : chaque thread remplace un seul gène
[FK12]	ED co-évolutionnaire	$pop_{gpu}sol_{bl}elt_{th}$	évaluation des deux populations : chaque bloc est associé au même individu dans chaque population et les threads calculent les fitness associées à tous les autres individus
		$pop_{gpu}sol_{bl}elt_{th}$	mise à jour des fitness : semblable mais une seule population
		$pop_{bl}sol_{th}$	croisement + mutation : un seul bloc lancé
		$pop_{gpu}sol_{bl}elt_{th}$	évaluation + choix des nouveaux individus

Tableau 3.4 – Classification des stratégies de parallélisation GPU hybrides utilisées dans la littérature selon la dimension "élément de calcul" (partie 2).

Malgré le grand nombre de stratégies de parallélisation GPU existantes, les mêmes sont bien souvent employées. La stratégie la plus utilisée dans ces différents noyaux est celle de type $pop_{gpu}sol_{bl}elt_{th}$ qui permet de tirer profit au maximum des capacités du GPU et de ses trois niveaux de parallélisation. Les noyaux qui emploient la stratégie $pop_{gpu}elt_{th}$ bénéficient également des trois niveaux de parallélisation de façon implicite. En effet, les éléments des solutions sont toujours associés aux threads mais les solutions n'ont pas besoin d'être structurées en blocs. Certains noyaux emploient l'approche de parallélisation CPU qui associe la solution aux threads et qui correspond à stratégie GPU $pop_{gpu}sol_{th}$. La population peut également être structurée en sous-populations attribuées aux différents blocs où les threads gèrent une solution avec la stratégie $pop_{bl}sol_{th}$ ou un élément avec la stratégie $pop_{bl}elt_{th}$.

La performance de la stratégie de parallélisation utilisée dépend du problème à résoudre et de la métaheuristique choisie mais également du type de mémoire utilisé, très important dans la programmation GPU. La taxonomie peut donc intégrer une seconde dimension qui précise l'utilisation des différentes mémoires dans les stratégies.

3.2 Deuxième dimension : mémoire

Cinq types de mémoires de taille, de type d'accès et de vitesse d'accès différents peuvent être retrouvés dans l'architecture GPU comme le montre la Figure 3.4 : mémoire globale, mémoire partagée, mémoire constante, mémoire texture et registres. La métaheuristique, quant à elle, peut nécessiter généralement quatre types de données qui doivent être stockés judicieusement dans ces différentes mémoires.

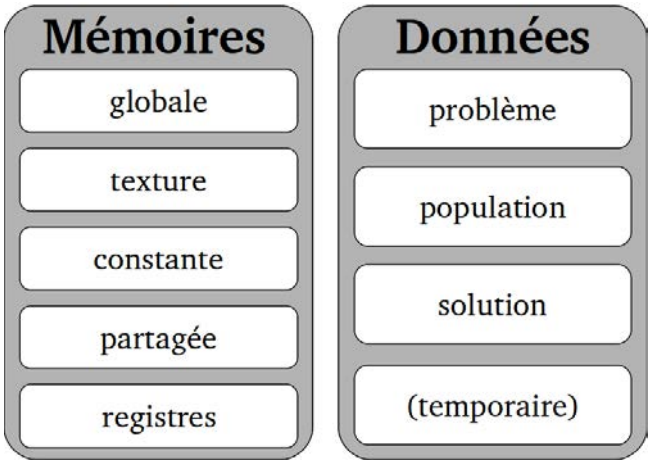


Figure 3.4 – Types de mémoires GPU et types de données des métaheuristiques.

Les données du problème ne sont habituellement pas modifiées durant l'exécution de l'algorithme. Les paramètres et les structures de données dont la taille varie selon celle du problème font partie de ce type de données. Par exemple, pour le problème du sac à dos, les objets, leurs poids et la capacité du sac sont nécessaires. Pour le problème du Voyageur de Commerce, la matrice des distances est une structure de donnée dont la taille augmente selon celle du problème et les listes de candidats, selon le paramètre indiquant le nombre de candidats.

Les données de la population, autres que celles du problème, sont celles qui sont partagées parmi toutes les solutions mais qui n'en caractérisent pas une en particulier. Elles peuvent permettre aux solutions de la population de communiquer entre elles. La meilleure solution de l'itération et/ou la meilleure solution globale font partie de ce type de données. La matrice de phéromone pour la métaheuristique OCF est également une structure de données de la population mais dont la taille est dépendante de celle du problème.

Les données de la solution comprennent la solution elle-même et toutes les structures indispensables à son calcul. Les listes taboues et de probabilités font partie de ce type de données pour la métaheuristique OCF ainsi que les meilleures positions de la particule et de ses voisins pour les algorithmes OEP.

Les données temporaires sont nécessaires au bon déroulement de la métaheuristique. Elles ne sont cependant pas considérées afin de ne pas alourdir la taxonomie, celles-ci relevant davantage du niveau technique que de la stratégie de parallélisation.

Malgré le fait que cette dimension puisse apporter une grande quantité d'informations utiles pour caractériser les approches de parallélisation, elle peut facilement complexifier la

taxonomie et la définition des stratégies pour deux raisons. Premièrement, elle implique un large choix de combinaisons possibles si les implémentations parallèles utilisent plusieurs noyaux caractérisés par des stratégies différentes. Deuxièmement, pour un type de données spécifique (problème, population ou solution), plusieurs mémoires GPU peuvent être employées.

Afin de remédier à la première contrainte, les stratégies qui étaient auparavant de type $pop_x sol_y elt_z$ spécifieront les mémoires utilisées en exposant pour ne pas complexifier les stratégies déjà existantes : $pb^a pop_x^b sol_y^c elt_z$ où a, b et c sont les types de mémoire utilisés. Dans ce cas, le terme pb doit être ajouté pour préciser le type de données du problème. Afin de simplifier la définition de la stratégie, le terme pb est supprimé et les types de mémoire employés par les données du problème sont spécifiées en exposant de elt : $pop_x^b sol_y^c elt_z^a$. Ces différents exposants correspondent alors à gl pour la mémoire globale, tex pour la mémoire texture, cst pour la mémoire constante, par pour la mémoire partagée et reg pour les registres. Afin de remédier à la seconde contrainte, chaque type de données n'est pas pris en compte dans le détail mais dans sa globalité. Ainsi, le nombre de mémoires utilisées est limité aux deux plus importantes de chaque type de données. Par exemple, si les données du problème sont stockées principalement dans les mémoires texture et constante mais rarement dans la mémoire globale, la stratégie est de type $pop_x^b sol_y^c elt_z^{tex,cst}$. De plus, les données résidant dans la mémoire constante, texture ou partagée ont forcément été stockées dans la mémoire globale avant d'être chargées. Dans ce cas, la mémoire globale n'a pas besoin d'être précisée dans le type de la stratégie.

Afin d'illustrer ces explications, les Tableaux 3.5 et 3.6 présentent une classification des stratégies employées dans la littérature en ajoutant la dimension "mémoire". Certains travaux ont été écartés car les éléments fournis pour spécifier les différents types de mémoire n'étaient pas assez explicités.

Référence	Stratégie	Description
[LMT11b]	$sol_{gpu}^{tex,gl} elt_{th}^{tex}$	solution stockée en tex et fitness des voisins en gl (par utilisée pour trouver la meilleure fitness)
[LMT11a]	$pop_{gpu} sol_{th}^{gl} elt^{tex}$	solutions et structures des solutions en gl, données d'entrée en tex
[OTB11]	$pop_{gpu} sol_{th}^{gl} elt^{par}$	données d'entrée en par et solutions en gl
[CB11]	$pop^{par} sol_{gpu}^{gl} elt_{th}^{cst}$	en par, évaluation de la matrice des temps de fin des tâches (une ligne/colonne/diagonale stockée à la fois)
[Zhu11]	$pop_{gpu}^{tex,par} sol_{th}$	indices des parents sélectionnés pour le croisement en par et leurs données en tex
[WWW11]	$pop_{gpu} sol_{th}^{gl,par}$	données des particules en gl et transférées en par pour les calculs
[MDC11]	$pop_{bl}^{par,gl} sol_{th}^{reg,gl}$	$SyncPSO$ - données des particules en reg, tableaux stockant ces données en gl, meilleure solution globale en gl et particules communiquant avec par
[MNC11]	$pop_{gpu} sol_{bl}^{par,gl} elt_{th}$	données des particules en par et meilleure fitness de chaque particule en gl
[MKQ+12]	$pop_{gpu}^{gl} sol_{th}^{gl} elt^{gl}$	seule gl est utilisée car il n'y a aucune garantie que les gros problèmes puissent être contenus dans par

Tableau 3.5 – Classification des stratégies de parallélisation GPU selon les dimensions "élément de calcul" et "mémoire" (partie 1).

Référence	Stratégie	Description
[MWMA09]	$pop_{bl}sol_{th}^{gl}$	individus dans gl - enfants produits stockés dans par mais
	$pop_{bl}sol_{th}^{gl,par}$	pour les plus gros problèmes, stockés dans gl
[PAL11]	$pop_{gpu}sol_{th}^{gl}$	tournoi binaire + mutation
	$pop_{gpu}sol_{bl}^{gl}elt_{th}$	croisement
	$pop_{gpu}sol_{bl}^{gl,par}elt_{th}$	évaluation : fitness partielles en par pour effectuer une réduction
[TF11]	$pop_{gpu}^{gl}sol_{bl}^{gl}elt_{th}^{tex}$	données stockées en gl et matrices accessibles en lecture seule en tex
[STT11]	$pop_{gpu}sol^{gl}elt_{th}^{tex}$	initialisation des particules + mise à jour de leurs données
	$pop_{gpu}sol_{th}^{gl,par}elt^{tex}$	mise à jour des fitness : utilisation de par ou de gl selon la
	$pop_{gpu}sol_{th}^{gl}elt^{tex}$	taille des instances
	$pop_{bl}sol_{th}^{gl}elt^{tex}$	mise à jour des meilleures valeurs
	$pop_{bl}sol_{th}^{gl,par}elt^{tex}$	détermination des meilleures et plus mauvaises particules
	$pop_{gpu}sol^{gl}elt_{th}^{tex}$	remplacement des mauvaises particules

Tableau 3.6 – Classification des stratégies de parallélisation GPU selon les dimensions "élément de calcul" et "mémoire" (partie 2).

Comme le montrent ces deux tableaux et la revue de littérature du Chapitre 2, chaque mémoire a un rôle particulier à jouer dans l'implémentation des métaheuristiques. Les registres, qui sont les mémoires les plus rapides du GPU, sont souvent utilisés pour stocker des parties de solutions afin d'y accéder plus rapidement. La mémoire texture contient généralement les structures de données de grande taille qui ne sont pas modifiées lors de l'exécution des noyaux. Le plus souvent, ce sont donc les données du problème qui sont conservées dans ce type de mémoire, comme la solution à partir de laquelle le voisinage est calculé dans le cas des RL à solution unique. La mémoire constante est également utilisée pour stocker les données non modifiées durant les phases parallélisées comme les paramètres et les données d'entrée du problème. Cependant, cette mémoire de petite taille implique une limitation en ce qui concerne la taille des problèmes pouvant être contenus et résolus.

Cette limitation survient également lorsque les données du problème et les solutions sont stockées dans la mémoire partagée. Dans ce cas, certains auteurs sauvegardent les données du problème dans la mémoire partagée pour les petites instances et dans la mémoire globale pour les plus grosses, ce qui implique l'utilisation de deux stratégies différentes. La mémoire partagée est également employée pour effectuer certains calculs et réductions parallèles. Les données sont alors chargées dans cette mémoire temporairement pour accélérer ces phases. Les solutions et les structures relatives aux solutions sont bien souvent stockées dans la mémoire globale car elles leur assurent un espace mémoire, peu importe la taille du problème. Certains auteurs en quête de simplicité conservent d'ailleurs la totalité de leurs données dans la mémoire globale. Enfin, les données peu souvent accédées doivent être conservées dans la mémoire globale plutôt que d'être chargées dans la mémoire partagée. En effet, le temps de les copier est souvent plus long que celui d'y accéder directement dans la mémoire globale.

3.3 Conclusion

Dans ce chapitre, les différents niveaux de parallélisation de l'architecture GPU et des métaheuristiques ont été mis en évidence. Des liens entre ces différents niveaux ont pu être mis en place, donnant ainsi naissance à plusieurs combinaisons possibles. Ces dernières ont ensuite été classifiées au sein d'une première dimension de façon à pouvoir compléter les classifications existantes. Partant du principe que les stratégies de parallélisation pouvaient être grandement influencées par le type de mémoire GPU utilisé, une seconde dimension a été ajoutée à la taxonomie. Elle permet, pour chaque type de données nécessaires aux métaheuristiques, de spécifier la ou les mémoires employées. Ces deux dimensions peuvent être vues comme une boîte à outils qui permet de créer une multitude de stratégies de parallélisation des métaheuristiques. Les subtilités des approches de parallélisation GPU recensées dans la littérature ont donc pu être classifiées à l'intérieur de ce cadre de référence.

Afin d'illustrer cette taxonomie et d'en valider certaines parties dans un contexte appliqué de conception et de développement de méthodes d'optimisation parallèles compétitives, elle est utilisée pour définir plusieurs implémentations parallèles de métaheuristiques sur GPU. Ces dernières sont également évaluées et comparées de sorte à mettre en évidence les liens unissant les configurations utilisées et la performance obtenue. Le Chapitre 4 est dédié à la parallélisation d'une métaheuristique basée sur une population de solutions : l'Optimisation par Colonie de Fourmis. Le Chapitre 5 est quant à lui dédié à la parallélisation d'une métaheuristique à solution unique intégrant une Recherche Locale : la Recherche Locale Itérée.

Chapitre 4

Stratégies de parallélisation GPU pour l'Optimisation par Colonie de Fourmis

Ce chapitre est consacré à la parallélisation GPU des métaheuristiques à population de solutions et, plus particulièrement, l'Optimisation par Colonie de Fourmis (OCF). À partir du cadre de référence fourni par la taxonomie du Chapitre 3, plusieurs stratégies adaptées à l'environnement de calcul GPU sont proposées et évaluées expérimentalement. D'importantes questions algorithmiques, conceptuelles et techniques sont également abordées dans ce contexte.

Ce chapitre est organisé comme suit. Le fonctionnement général de l'OCF est tout d'abord explicité puis l'algorithme spécifique *MAX-MIN Ant System* (MMAS) est détaillé. Plusieurs implémentations parallèles du MMAS sur GPU sont ensuite proposées, évaluées et comparées autant au niveau de la qualité des solutions que de la réduction du temps de calcul. Elles se distinguent par leur type d'approche ($FOURMI$ et $COLONIE$) et par leur association aux éléments de calcul (processeur ou multiprocesseur). L'ensemble de ces contributions fait partie des premiers travaux dans ce domaine et ont été publiés dans la revue internationale *Journal of Parallel and Distributed Computing* [DDGK13].

L'approche $FOURMI$ utilise une seule colonie où chaque fourmi construit sa solution en parallèle. D'après la taxonomie proposée dans le Chapitre 3, les stratégies de parallélisation proposées sont de type $pop_{gpu}sol_{th}$, $pop_{gpu}sol_{bl}^{par}elt_{th}$ et $pop_{gpu}sol_{bl}^{gl}elt_{th}$. Dans le premier cas, la notion d'élément de calcul est associée au processeur et chaque fourmi correspond à un thread. Les travaux préliminaires basés sur cette première stratégie ont fait l'objet d'une publication sous forme de résumé dans les actes de la conférence internationale *3rd International Conference on Metaheuristics and Nature Inspired Computing (META'10)* [DDK10]. Une étude plus détaillée analysant l'influence de la répartition des processus de recherche a été publiée dans les actes de la conférence internationale *The 2010 International Conference on Parallel and Distributed Processing Techniques and Applications - PDPTA'10* [DDGK10]. Dans les deux autres cas, la notion d'élément de calcul est associée au multiprocesseur. Chaque fourmi correspond donc à un bloc et la construction de sa solution est effectuée en parallèle par les threads. La stratégie de type $pop_{gpu}sol_{bl}^{par}elt_{th}$ stocke les données de chaque fourmi dans la mémoire partagée alors que celle de type $pop_{gpu}sol_{bl}^{gl}elt_{th}$ les conserve dans la mémoire globale. Les travaux préliminaires sur l'approche $FOURMI$ et ses trois stratégies ont fait l'objet d'une publication dans les actes de la conférence nationale *Rencontres francophones du Parallélisme (RenPar'20)* [DDK11].

L'approche *COLONIE* utilise plusieurs colonies évoluant en parallèle. Les stratégies proposées sont alors de type $pop_{bl}sol_{th}$, $pop_{gpu}sol_{th}$, $pop_{gpu}sol_{bl}^{par}elt_{th}$ et $pop_{gpu}sol_{bl}^{gl}elt_{th}$. Dans le premier cas, chaque colonie est associée à un bloc et les fourmis, aux threads. Dans les autres cas, chaque colonie est affectée à un GPU et les fourmis appliquent une stratégie de l'approche *FOURMI*. Une première étude sur cette approche a fait l'objet d'une publication sous forme de résumé dans la conférence nationale *13e congrès annuel de la Société française de Recherche Opérationnelle et d'Aide à la Décision* [DDK12a].

Le premier algorithme OCF a été conçu pour résoudre le problème du Voyageur de Commerce. Ce dernier est un problème académique phare dans le domaine de la recherche opérationnelle. Il a fait l'objet de nombreux travaux dont les résultats peuvent être utilisées pour éprouver les approches proposées. Par conséquent, il est choisi comme cadre de référence et sa description est fournie dans la prochaine section.

4.1 Problème du Voyageur de Commerce

Le problème du Voyageur de Commerce (VC) est schématisé dans la Figure 4.1. Il a inspiré les études de mathématiciens, d'informaticiens, de chimistes, de physiciens et de psychologues [App06]. Il peut être défini par un graphe pondéré complet orienté $G = (V, A, d)$. $V = \{1, 2, ..., n\}$ représente l'ensemble des sommets et $A = \{(i, j)|(i, j) \in V \times V\}$, l'ensemble des arcs ou des arêtes reliant les sommets. $d : A \to \mathbb{N}$ est une fonction assignant un poids ou une distance (entier positif) d_{ij} à chaque arc $(i, j) \in V$. L'objectif de ce problème est de trouver une tournée (ou cycle hamiltonien) de distance minimale dans G passant par tous les sommets une seule fois et retournant au sommet de départ. Son interprétation la plus connue est celle du représentant qui recherche la plus courte tournée entre n clients situés dans des villes différentes.

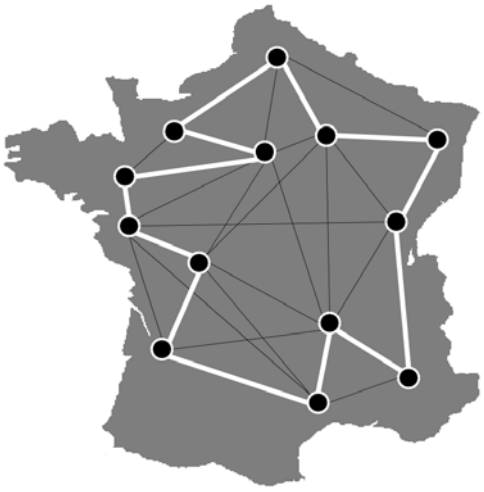


Figure 4.1 – Problème du Voyageur de Commerce.

Deux cas de problèmes du VC sont présentés dans la Figure 4.2 : symétrique et asymétrique. Dans le premier cas, il n'existe qu'un seul chemin entre deux villes adjacentes, c'est-à-dire, $d_{ij} = d_{ji}$ pour $i, j \in V$. Dans le second, il est possible d'avoir deux arcs de poids ou de distance

différente entre deux villes adjacentes. Un problème symétrique satisfait l'inégalité triangulaire, c'est-à-dire que le chemin direct est toujours le plus court, si $d_{ij} <= d_{ik} + d_{kj},\ \forall i, j, k \in V$ [Rei94]. Dans cette thèse, seuls les problèmes symétriques sont considérés.

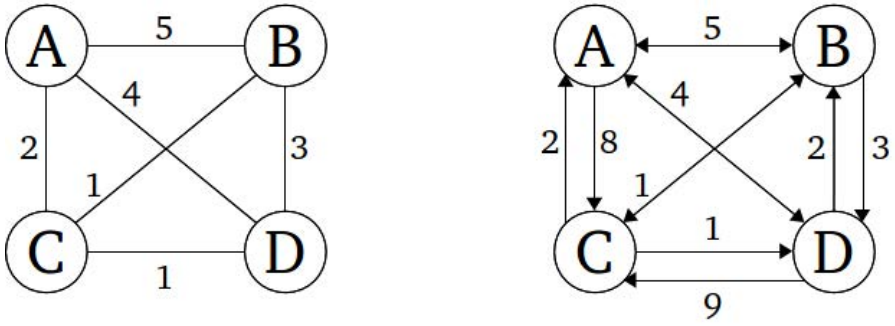


Figure 4.2 – Problème du VC symétrique (à gauche) et asymétrique (à droite).

Le problème du VC est un problème $\mathcal{NP}$-*difficile* dont le temps de résolution croît exponentiellement par rapport à sa taille. Le nombre de tournées possibles pour un problème de n villes est $(n-1)!$ pour le cas asymétrique et $\frac{(n-1)!}{2}$ pour le cas symétrique [Gre08]. Par exemple, pour un problème symétrique de 5 villes, seules 12 possibilités sont examinées alors que pour une instance de 3038 villes, le nombre de tournées possibles est $1,179738539e+9259$. Cependant, quelques cas spéciaux peuvent être résolus en temps polynomial comme ceux où $d_{ij} = 0,\ \forall i \geq j$ [Lap92]. Le problème du VC est utilisé dans de nombreux domaines : logistique, génétique, industrie, télécommunications, neuroscience, etc. Il est à la base de plusieurs applications de routage de véhicules mais peut également être appliqué à la planification du personnel, à l'ordonnancement de la production, ou bien encore, à la cristallographie aux rayons X. Une description complète du problème peut être trouvée dans Applegate [App06], Reinelt [Rei94] et Greco [Gre08].

4.2 Optimisation par Colonie de Fourmis

Les algorithmes d'Optimisation par Colonie de Fourmis (OCF) sont des méthodes à base de population de solutions inspirées du comportement collectif des fourmis qui résolvent naturellement des problèmes complexes comme le choix du plus court chemin de leur nid à une source de nourriture. De leurs observations, Dorigo et al. [DMC91] et Colorni et al. [CDM91] ont remarqué qu'une fourmi seule détient des capacités cognitives relativement simples et limitées contrairement à la colonie qui possède un comportement hautement structuré. Cette structuration émerge des interactions au sein même de la colonie où les fourmis communiquent indirectement entre elles grâce à une substance chimique odorante appelée phéromone.

Le comportement des fourmis est illustré dans la Figure 4.3. Tout d'abord, les fourmis se déplacent aléatoirement de leur nid à une source de nourriture tout en déposant de la phéromone sur le chemin emprunté (a). Elles tiennent ensuite compte des traces de phéromone rencontrées sur leur parcours pour guider leurs déplacements. Lorsqu'un obstacle se place sur leur trajet (b), elles peuvent le contourner en passant par deux chemins. Selon la quantité de phéromone déposée, chaque fourmi reçoit un stimulus plus ou moins fort qui lui permet de choisir un de ces

deux chemins. Celles qui prennent le chemin le plus court peuvent alors effectuer un plus grand nombre de déplacements que les autres dans un intervalle de temps donné. Par conséquent, la quantité de phéromone déposée sur ce chemin augmente plus vite que sur l'autre. Les fourmis sont donc davantage attirées par celui-ci et le renforcent à leur tour (c). Enfin, l'évaporation des pistes de phéromone permet la disparition progressive du chemin le moins emprunté.

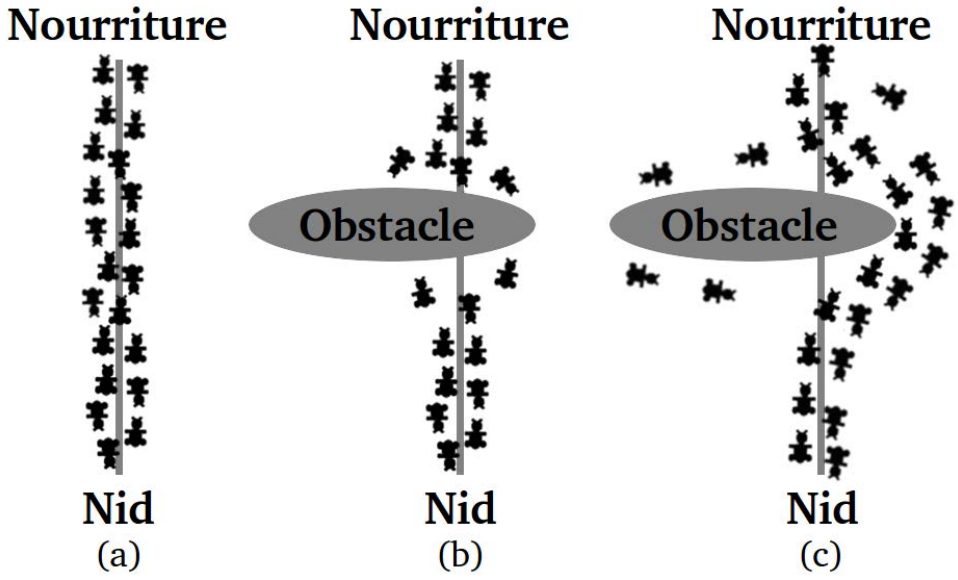


Figure 4.3 – Déplacement des fourmis.

Ainsi, en communiquant indirectement entre elles grâce à des modifications dynamiques de leur environnement, les fourmis peuvent construire une solution à un problème en s'appuyant sur leur expérience collective. Les algorithmes OCF utilisent des fourmis artificielles qui, à la différence des vraies, possèdent une mémoire, ne sont pas complètement aveugles et vivent dans un environnement où le temps est discret. Ces colonies artificielles peuvent donc être utilisées comme un outil d'optimisation applicable à des problèmes complexes. La prochaine section décrit l'algorithme OCF *MAX-MIN Ant System* qui est reconnu comme étant l'un des algorithmes OCF les plus performants [DS04].

4.3 MAX-MIN Ant System

L'algorithme *MAX-MIN Ant System* (MMAS) a été proposé par Stützle et Hoos [SH00]. Il est directement basé sur le premier algorithme OCF *Ant System* (AS) proposé par Dorigo et al. [DMC91, DMC96]. L'idée principale du MMAS consiste à effectuer une plus forte exploitation des meilleures solutions que le AS tout en évitant la stagnation rapide de la recherche. Son fonctionnement est décrit dans l'Algorithme 4.1.

Pendant un certain nombre d'itérations ou de cycles ($nbCycles$), les m fourmis de la colonie effectuent leurs tournées. Chaque fourmi k possède sa propre liste d'éléments tabous $tabou_k$ contenant les différentes villes dans l'ordre visité. Cette liste permet de calculer la longueur de la tournée effectuée et d'éviter la sélection d'une ville ayant déjà été visitée. Deux matrices de

```
Initialiser les paramètres de la colonie
Initialiser la matrice de phéromone τ avec tau_max pour chaque paire de villes
Pour t = 1 à nbCycles Faire
    Pour k = 1 à m Faire
        Placer la fourmi k sur une ville i choisie aléatoirement
        Tant que toutes les villes ne sont pas visitées Faire
            Déplacer la fourmi k sur la ville j selon la règle de transition d'état (4.1)
        Calculer la longueur L_k de la tournée T_k produite par la fourmi k
    Pour k = 1 à m Faire
        Si L_k < L_it Alors
            Mettre à jour la meilleure tournée de l'itération T_it par T_k
    Si L_it < L_gl Alors
        Mettre à jour la meilleure tournée T_gl par T_it
    Évaporer τ selon la formule (4.2)
    Mettre à jour τ avec chaque paire de villes de T_it ou T_gl selon la formule (4.3)
    Contrôler τ_min < τ < τ_max
    Mettre à jour τ avec le mécanisme de renforcement (4.5)
Retourner la tournée la plus courte T_gl
```

Algorithme 4.1 – Algorithme du MAX-MIN Ant System.

taille $n \cdot n$ sont utilisées afin de stocker les différentes distances entre les villes et la quantité de phéromone entre celles-ci. Chaque fourmi k est placée aléatoirement sur une ville de départ et se déplace d'une ville i à une ville j selon la règle de transition d'état suivante :

$$p_{ij}^k = \begin{cases} \frac{(\tau_{ij})^\alpha \cdot (\eta_{ij})^\beta}{\sum_{l \notin tabou_k} (\tau_{il})^\alpha \cdot (\eta_{il})^\beta} \; si \; j \notin tabou_k \\ 0 \; sinon \end{cases} \tag{4.1}$$

La fourmi k ne peut se déplacer que sur une ville n'ayant pas été visitée, c'est-à-dire, n'appartenant pas à sa liste $tabou_k$. Ce déplacement dépend de la quantité de phéromone τ_{ij} déposée entre i et j et de la visibilité η_{ij} entre ces deux villes. La visibilité correspond à l'inverse de la distance d_{ij} : $\eta_{ij} = \frac{1}{d_{ij}}$. Les paramètres α et β contrôlent l'importance relative de la piste de phéromone par rapport à celle de la visibilité. Avec $\alpha = 0$, seule la visibilité est prise en compte : plus une ville est proche, plus la probabilité de la choisir est grande. Avec $\beta = 0$, seules les pistes de phéromone jouent un rôle : plus l'intensité de la piste de phéromone est grande, plus ce trajet aura de chance d'être choisi. L'efficacité du MMAS dépend généralement d'un bon compromis entre ces deux paramètres.

Une fois les différentes probabilités de se déplacer vers les villes non visitées calculées, j est sélectionnée par une méthode de sélection proportionnelle (*proportionate selection*). Une de ces méthodes les plus populaires est celle de la roulette (*Roulette Wheel Selection*). La roulette est découpée en plusieurs parties où chacune correspond à une ville. La taille de chaque partie est proportionnelle à la probabilité de choisir la ville associée. Un point est alors tiré aléatoirement sur la roulette. La ville j correspondant à la partie sélectionnée par le point aléatoire est choisie comme prochaine ville à visiter.

Lorsque toutes les tournées sont complétées, l'algorithme compare leurs longueurs afin de mémoriser la meilleure solution de l'itération courante. Cette dernière est aussi comparée à la meilleure solution trouvée depuis le début de l'algorithme de sorte à conserver la plus courte. Toutes les pistes de phéromone sont ensuite évaporées selon la formule suivante :

$$\tau_{ij} = (1 - \rho) \cdot \tau_{ij} \tag{4.2}$$

où ρ est un paramètre contrôlant la quantité de phéromone évaporée.

Dans l'algorithme de base AS, toutes les fourmis déposent de la phéromone sur leur chemin. Dans le MMAS, seule la meilleure fourmi de l'itération courante k_{it} ou de l'exécution complète k_{gl} est autorisée à mettre à jour les traces de phéromone après chaque itération. Elle dépose alors une quantité $\Delta\tau_{ij}$ de phéromone entre les villes i et j proportionnelle à la distance totale L parcourue lors de la construction de la tournée :

$$\Delta\tau_{ij} = \begin{cases} \frac{1}{L} \textit{ si la tournée de la meilleure fourmi passe par } (i,j) \\ 0 \textit{ sinon} \end{cases} \tag{4.3}$$

Ainsi, les éléments qui apparaissent fréquemment dans les meilleures solutions sont plus renforcés. Quand la meilleure solution de l'exécution complète est utilisée, la recherche se dirige trop vite vers celle-ci et l'exploration de nouvelles solutions devient limitée. Le MMAS se concentre alors sur l'utilisation des meilleures solutions des itérations car elles peuvent être considérablement différentes les unes des autres. Il est également possible de mélanger les stratégies en utilisant la meilleure solution de l'itération par défaut et la meilleure solution de l'exécution complète à un nombre fixe d'itérations.

Afin d'éviter la stagnation prématurée de la recherche, la quantité de phéromone déposée sur chaque arête est bornée par des valeurs minimale τ_{min} et maximale τ_{max} de phéromone. En limitant la quantité de phéromone déposée ou évaporée, la différence entre les traces ne peut pas être trop extrême. Le calcul des probabilités, qui dépend de la quantité de phéromone déposée, est donc influencé de façon à ne pas toujours choisir la même solution et à explorer de nouvelles solutions. À la fin de chaque itération, la quantité de phéromone déposée sur chaque arête doit respecter les limites fixées. Ainsi, si elle est inférieure à τ_{min} ou supérieure à τ_{max}, elle est réinitialisée à τ_{min}, respectivement à τ_{max}. La borne minimale permet de laisser la possibilité à un chemin d'être exploré et la borne maximale évite qu'il ne soit trop renforcé par rapport aux autres. Leur valeur est mise à jour après chaque cycle. τ_{max} est également utilisée pour initialiser les traces de phéromone au début de l'algorithme ce qui permet une exploration plus importante des solutions pendant les premières itérations.

Enfin, un mécanise de renforcement (*trail-smoothing mechanism*) est incorporé au MMAS. Il permet d'ajuster l'intensité des traces de phéromone lorsque l'algorithme s'approche de la convergence. Son but est de faciliter l'exploration des solutions en augmentant la probabilité de sélectionner les arêtes ayant une faible quantité de phéromone. En effet, la stagnation de la recherche peut être indiquée par le facteur de λ-branchement moyen (*mean λ-branching factor*). Ce dernier correspond au nombre de liens partant de la ville i possédant des valeurs de phéromone τ_{ij} plus grandes que :

$$\lambda \cdot (tmax_i - tmin_i) + tmin_i \tag{4.4}$$

où $tmax_i$ est la valeur maximale de phéromone sur les arêtes partant de i et $tmin_i$ la valeur minimale. En prenant $\lambda = 0,05$, les auteurs montrent que si le facteur de $0,05$-branchement moyen est proche de 1, seulement quelques arêtes (bien souvent une seule) partant d'un sommet i auront une grande probabilité d'être sélectionnées et pratiquement aucune autre solution ne sera explorée. Le mécanisme de renforcement alors ajouté au MMAS augmente l'intensité des pistes de phéromone proportionnellement à la différence entre la borne maximale τ_{max} et la quantité de phéromone τ_{ij} :

$$\tau_{ij} = \tau_{ij} + \gamma \cdot (\tau_{max} - \tau_{ij}) \quad avec \quad 0 < \gamma \leq 1 \tag{4.5}$$

Lorsque le nombre de villes des problèmes étudiés augmente, le temps d'exécution de l'algorithme est de plus en plus long. Le MMAS peut donc inclure des listes de candidats qui permettent de réduire ce temps mais aussi d'orienter l'algorithme vers les arêtes les plus courtes, et donc, potentiellement vers les chemins les plus courts. Ces listes contiennent, pour chaque ville, les cl villes les plus proches ordonnées par distance croissante par rapport à cette ville. À chaque cycle, toutes les villes ne sont plus considérées mais seulement les villes candidates. En revanche, si toutes les villes de la liste ont déjà été visitées, les villes restantes sont considérées et celle dont la valeur $[\tau_{ij}(t)]^\alpha \cdot [\eta_{ij}(t)]^\beta$ est la plus grande sera choisie.

La nature concurrente de la construction des tournées et de la recherche globale dans l'espace des solutions permet à la métaheuristique OCF d'être un bon candidat pour la parallélisation. Cependant, ce potentiel apporte également des défis importants concernant principalement la gestion de la phéromone et la taille des structures de données devant être maintenues. La prochaine section décrit les différentes stratégies de parallélisation GPU proposées pour cette métaheuristique.

4.4 Stratégies de parallélisation GPU

Pour chaque type d'approche $FOURMI$ et $COLONIE$, plusieurs implémentations parallèles du MMAS sur GPU qui diffèrent par leur traitement des deux dimensions de la taxonomie, c'est-à-dire la définition d'élément de calcul et l'utilisation des mémoires, sont proposées.

La Section 4.4.1 est consacrée à l'approche $FOURMI$. Cette dernière utilise une seule colonie où chaque fourmi construit sa solution en parallèle. La première stratégie proposée associe chaque fourmi à un thread et la seconde, à un bloc. Dans cette dernière, les données des fourmis peuvent être stockées dans la mémoire partagée ou globale.

La Section 4.4.2 est consacrée à l'approche $COLONIE$. Cette dernière utilise plusieurs colonies évoluant en parallèle. La première stratégie proposée associe chaque colonie à un bloc et la seconde, à un GPU. Dans cette dernière, les solutions peuvent être assignées aux threads ou aux blocs.

4.4.1 Stratégies de parallélisation pour l'approche $FOURMI$

La plus grande partie du temps de calcul de l'algorithme MMAS est consacrée à la construction des tournées. Comme chaque fourmi peut construire sa tournée indépendamment des

autres, cette phase représente un bon candidat pour la parallélisation. Elle a donc été déplacée du CPU au GPU. La plupart des problèmes rencontrés durant le processus de parallélisation sont reliés à la gestion des mémoires. Plus particulièrement, les transferts de données entre le CPU et le GPU ainsi que les accès à la mémoire globale du GPU requièrent des temps prohibitifs. De façon générale, ces accès peuvent être réduits en utilisant la mémoire partagée rapide du GPU. Cependant, cette dernière est de petite taille et locale à chaque multiprocesseur. Dans le cas des algorithmes OCF, les trois structures de données centrales sont la matrice de phéromone, la matrice de distance et les listes de candidats. Elles sont nécessaires à toutes les fourmis de la colonie pour calculer la règle de transition d'état mais leur taille conséquente (allant de $O(n \cdot cl)$ à $O(n^2)$) ne permet pas de les contenir dans la mémoire partagée. Elles sont donc conservées dans la mémoire globale du GPU. D'autre part, comme elles ne sont pas modifiées durant la phase de construction des tournées, il est possible de profiter du cache texture en lecture seule pour réduire ces temps d'accès mémoire. Cette stratégie de parallélisation générale est résumée dans la Figure 4.4 et décrite dans l'Algorithme 4.2. Tout d'abord, le CPU initialise les différentes structures de données, alloue l'espace mémoire nécessaire sur le GPU et y copie les données. Le noyau qui construit les tournées des fourmis en parallèle est exécuté. Le CPU récupère ensuite les différentes tournées du GPU, met à jour la matrice de phéromone puis la recharge dans la mémoire du GPU. Enfin, le CPU mémorise la tournée la plus courte et un autre cycle peut débuter.

```
Initialiser les paramètres de la colonie et la matrice de phéromone τ
Copier la matrice de distance, τ et les listes de candidats dans la mémoire texture du GPU
Copier les structures de données de la colonie dans la mémoire globale du GPU
Initialiser le nombre de blocs nbbl et de threads par bloc nbth
Pour t = 1 à nbCycles Faire
    Noyau avec nbbl blocs de nbth threads
        Récupérer l'identifiant id de la fourmi
        Construction de la tournée de la fourmi id
        Calculer la longueur L_id de la tournée T_id produite par la fourmi id
    Récupérer du GPU sur le CPU les tournées T construites et leurs longueurs L
    Pour k = 1 à m Faire
        Mettre à jour τ pour chaque arête (i, j) de la tournée effectuée par la fourmi k
    Évaporer τ
    Pour k = 1 à m Faire
        Si L_k < L_g Alors
            Mettre à jour la meilleure tournée T_gl par T_k
    Copier τ dans la mémoire texture du GPU
Retourner la tournée la plus courte T_gl
```

Algorithme 4.2 – Pseudo-code de l'approche générale $FOURMI$.

À partir de cette structure de parallélisation générale, deux stratégies spécifiques de parallélisation de la phase de construction des tournées sont proposées. Elles diffèrent principalement sur leur définition d'élément de calcul. Sur une architecture CPU conventionnelle, ce concept est habituellement associé à un processeur monocœur ou à un des cœurs d'un processeur multicœur.

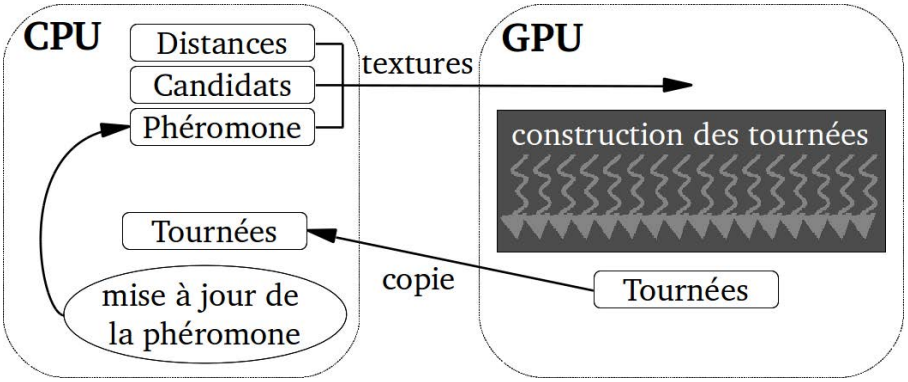


Figure 4.4 – Stratégie de parallélisation générale de l'approche $FOURMI$.

Sur une architecture GPU, comme le montrent les travaux de parallélisation GPU présentés dans le Chapitre 2, un des choix les plus employés est d'associer ce concept à un processeur. Dans ce cas, la première stratégie définie consiste à assigner chaque fourmi à un thread CUDA. Cette première stratégie est nommée $FOURMI_{thread}$ et présentée en Figure 4.5. Dans le noyau décrit dans l'Algorithme 4.2, l'identifiant id de la fourmi correspond donc à l'identifiant du thread. Chaque thread calcule alors la règle de transition d'état de sa fourmi de façon SIMD.

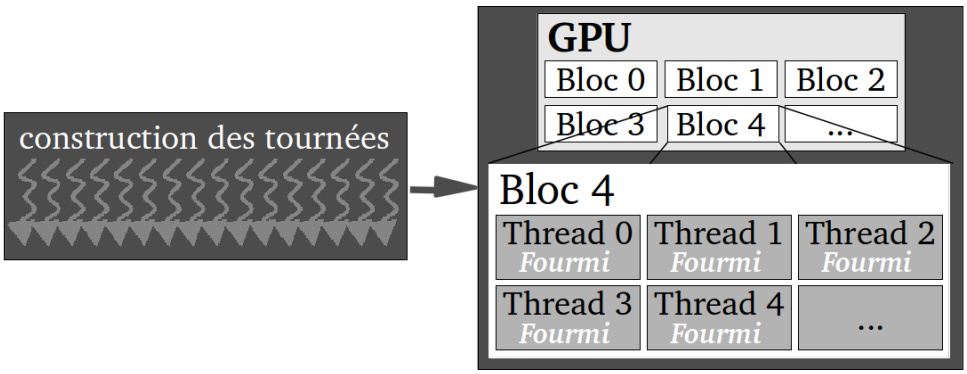


Figure 4.5 – Stratégie de parallélisation $FOURMI_{thread}$.

Cette stratégie a l'avantage d'autoriser l'exécution d'un grand nombre de fourmis sur chaque multiprocesseur mais l'inconvénient de limiter l'utilisation de la mémoire partagée rapide du GPU. En effet, chaque fourmi a besoin de ses propres structures de données, principalement les tournées et les tableaux de probabilités (de taille $O(n)$), pour calculer efficacement la règle de transition d'état nécessaire à la construction d'une solution. De simples calculs montrent qu'utiliser la mémoire partagée pour ces structures forcerait l'algorithme à utiliser un très petit nombre de fourmis sur chaque multiprocesseur et cette restriction s'intensifierait linéairement avec la taille du problème. Des optimisations de code pourraient augmenter ce nombre d'un facteur constant mais ne suffirait pas à contourner les limitations matérielles. Cependant, ces structures de données peuvent être stockées dans la mémoire globale et accédées en mode lecture et écriture durant la phase de construction des tournées. D'après la taxonomie proposée dans le Chapitre 3, la stratégie $FOURMI_{thread}$ est donc de type $pop_{gpu}^{tex} sol_{th}^{gl} elt^{tex,reg}$. La colonie est déportée sur le GPU (pop_{gpu}) et les fourmis sont associées aux threads (sol_{th}). Les données du

problème sont les paramètres de l'OCF, la matrice de distance et les listes de candidats. Les paramètres sont stockés dans les registres (elt^{reg}) et les deux autres structures dans la mémoire texture (elt^{tex}). Les données principales de la colonie concernent la matrice de phéromone également placée dans la mémoire texture (pop^{tex}). Les données des fourmis sont, quant à elles, stockées dans la mémoire globale (sol^{gl}).

La seconde stratégie est basée sur l'association de l'élément de calcul à un multiprocesseur. Dans ce cas, chaque fourmi est associée à un bloc CUDA et le parallélisme est préservé pour la construction des tournées. Cette seconde stratégie est nommée $FOURMI_{bloc}$ et présentée en Figure 4.6. Dans le noyau décrit dans l'Algorithme 4.2, l'identifiant id de la fourmi correspond donc à l'identifiant du bloc. Un seul thread par bloc est en charge de la construction des tournées des fourmis, mais un niveau de parallélisme supplémentaire peut être exploité pendant le calcul de la règle de transition d'état. En effet, chaque fourmi évalue les villes candidates avant de sélectionner celle qu'elle ajoutera à la solution courante. Ces évaluations pouvant être effectuées en parallèle, elles sont assignées aux threads restants du blocs.

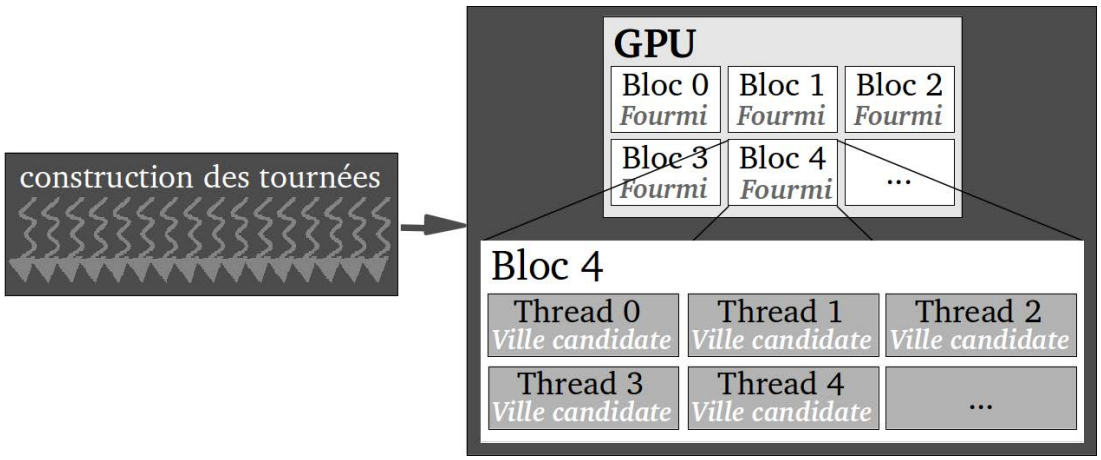


Figure 4.6 – Stratégie de parallélisation $FOURMI_{bloc}$.

Cette stratégie est donc de type $pop^{tex}_{gpu} sol_{bl} elt^{tex,reg}_{th}$. Elle permet d'utiliser les trois niveaux de parallélisation du GPU. La colonie est déportée sur le GPU (pop_{gpu}), les fourmis sont associées aux blocs (sol_{bl}) et leurs éléments aux threads (elt_{th}). Comme pour la stratégie $FOURMI_{thread}$, les données du problème sont placées dans la mémoire texture et dans les registres ($elt^{tex,reg}$) et les données de la colonie dans la mémoire texture (pop^{tex}). Si l'idée de la première stratégie était suivie, les structures de données de la fourmi serait conservées dans la mémoire globale. Cependant, puisqu'une seule fourmi est assignée à un bloc, il devient possible de tirer avantage de la mémoire partagée pour des problèmes plus gros que quelques dizaines de villes. Les données nécessaires pour construire la règle de transition d'état sont donc stockées dans cette mémoire rapide et accessible à tous les threads participant aux calculs. Dans le but d'évaluer les bénéfices et les limites de l'utilisation de la mémoire partagée dans ce contexte, deux variantes de la stratégie $FOURMI_{bloc}$ sont distinguées : $FOURMI^{globale}_{bloc}$ et $FOURMI^{partagée}_{bloc}$. Dans le premier cas, la stratégie est de type $pop^{tex}_{gpu} sol^{gl}_{bl} elt^{tex,reg}_{th}$ car les données des fourmis sont conservées dans la mémoire globale (sol^{gl}). Dans le second cas, la stratégie est de type $pop^{tex}_{gpu} sol^{par}_{bl} elt^{tex,reg}_{th}$ car les données des fourmis sont stockées dans la mémoire partagée (sol^{par}).

4.4.2 Stratégies de parallélisation pour l'approche $COLONIE$

L'approche $COLONIE$ consiste à distribuer plusieurs colonies sur les éléments de calcul afin qu'elles évoluent en parallèle. Dans ce cas, deux configurations matérielles possibles sont considérées : un simple GPU et plusieurs GPU. Les différentes colonies peuvent donc être associées aux blocs ou aux GPU. La première stratégie, nommée $COLONIE_{bloc}$, est présentée dans la Figure 4.7. Elle associe chaque colonie à un bloc CUDA et applique une stratégie similaire à la stratégie $FOURMI_{thread}$ où les fourmis de la colonie sont associées aux threads du bloc. Certaines structures de données, telles que les matrices de phéromone (de taille $O(n^2)$) et la meilleure tournée des fourmis (de taille $O(n)$), doivent être créées pour chaque colonie. Dans le but de limiter les transferts CPU-GPU prohibitifs pour cette quantité de données conséquente, la totalité de l'algorithme est déportée sur le GPU. Comme pour l'approche $FOURMI$, la matrice de distances et les listes de candidats ne sont pas modifiées durant l'évolution des colonies. Elles peuvent donc tirer profit du cache de mémoire texture. Cependant, les matrices de phéromone sont conservées dans la mémoire globale puisque les mises à jour sont déportées sur le GPU.

Cette stratégie est de type $pop_{bl}^{gl} sol_{th}^{gl} elt^{tex,reg}$. Les colonies sont associées aux blocs (pop_{bl}) et les fourmis aux threads (sol_{th}). Comme pour l'approche $FOURMI$, les données du problème sont la matrice de distance et les listes de candidats placées dans la mémoire texture ($elt^{tex,reg}$) et les paramètres de l'OCF stockés dans les registres. Quant à elles, les données de la colonie qui concernent la matrice de phéromone sont conservées dans la mémoire globale (pop^{gl}), tout comme les données des fourmis (sol^{gl}).

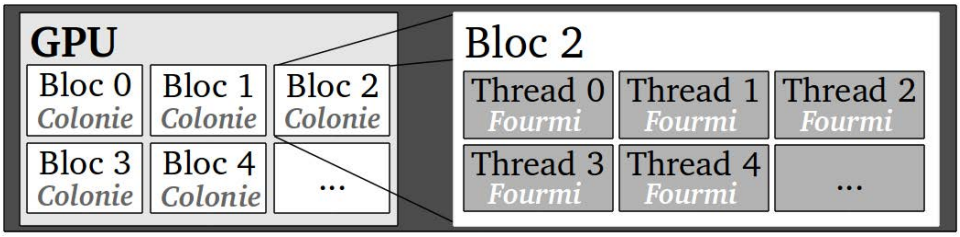


Figure 4.7 – Stratégie de parallélisation $COLONIE_{bloc}$.

L'Algorithme 4.3 fournit un pseudo-code de la stratégie $COLONIE_{bloc}$. Tout d'abord, les structures de données sont créées sur le CPU pour chaque colonie puis copiées dans la mémoire du GPU. Un noyau est ensuite lancé où chaque colonie est identifiée par l'indice d'un bloc et chaque fourmi, par l'indice d'un thread. Les colonies évoluent en parallèle durant $nbCycles$ itérations dans lesquelles chaque fourmi construit sa tournée. À la fin du noyau, la tournée la plus courte de chaque colonie est récupérée sur le CPU afin de déterminer la meilleure solution globale.

La seconde stratégie de parallélisation, nommée $COLONIE_{gpu}$, est illustrée dans la Figure 4.8. Elle permet d'utiliser un environnement contenant plusieurs GPU interconnectés et de bénéficier des stratégies décrites dans la Section 4.4.1. Elle associe chaque colonie à un GPU et peut être de 3 types différents : $pop_{gpu}^{tex} sol_{th}^{gl} elt^{tex,reg}$ comme la stratégie $FOURMI_{thread}$, $pop_{gpu}^{tex} sol_{bl}^{gl} elt_{th}^{tex,reg}$ comme la stratégie $FOURMI_{bloc}^{globale}$ et $pop_{gpu}^{tex} sol_{bl}^{par} elt_{th}^{tex,reg}$ comme la stratégie

Initialiser les paramètres des colonies et les matrices de phéromone τ
Copier la matrice de distance et les listes de candidats dans la mémoire texture du GPU
Copier les structures de données des colonies et τ dans la mémoire globale du GPU
Initialiser le nombre de blocs $nbbl$ et de threads par bloc $nbth$
Noyau avec $nbbl$ **blocs de** $nbth$ **threads**
  Récupérer l'identifiant du bloc $idCol$ et du thread $idFourmi$
  Pour $t = 1$ à $nbCycles$ **Faire**
    Construire la tournée de la fourmi $idFourmi$
    Calculer la longueur $L_{idFourmi}$ de la tournée $T_{idFourmi}$ produite par la fourmi $idFourmi$
    Mettre à jour τ^{idCol} et T_{gl}^{idCol}
Récupérer sur le CPU la tournée T_{gl}^{idCol} de chaque colonie $idCol$ et trouver la meilleure T_{gl}
Retourner la tournée la plus courte T_{gl}

Algorithme 4.3 – Pseudo-code de la stratégie $COLONIE_{bloc}$.

$FOURMI_{bloc}^{partagée}$. Dans le premier cas, chaque fourmi de la colonie est associée à un thread et dans les autres, à un bloc.

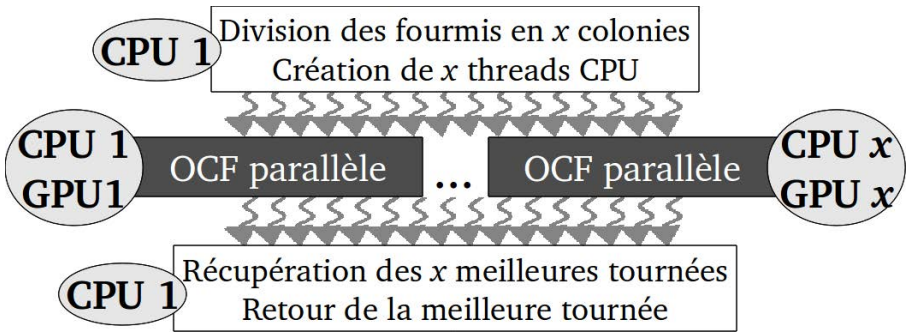


Figure 4.8 – Stratégie de parallélisation $COLONIE_{gpu}$.

Cette stratégie implique une forme de parallélisation CPU pour gérer chaque colonie avant et après l'exécution sur le GPU. Pour ce faire, une région parallèle crée plusieurs threads d'exécution CPU qui peuvent prendre la forme de threads ou de processus au niveau de l'application. Elle duplique ensuite l'algorithme et les données de chaque colonie. Chaque GPU devient alors le co-processeur SIMT d'un thread CPU donné, et d'un élément de calcul si un tel environnement est disponible au niveau du CPU. Lorsque chaque thread a initialisé sa colonie et transféré les données associées à son GPU, il applique la stratégie $FOURMI_{thread}$ ou $FOURMI_{bloc}$ à sa colonie. Quand la condition de fin est atteinte pour chaque colonie, chaque thread CPU récupère sa tournée la plus courte. Enfin, à la fin de la région parallèle CPU, la meilleure solution est gardée comme solution du problème. Un pseudo-code de cette stratégie est présenté dans l'Algorithme 4.4.

Une étude expérimentale étendue a été effectuée afin d'évaluer et de comparer ces différentes stratégies dans un environnement de calcul GPU de technologie de pointe. Cette étude est détaillée dans la prochaine section.

Région parallèle CPU avec $nbth$ **threads**
  Initialiser les paramètres des colonies
  Récupérer l'indice id du GPU : l'identifiant du thread CPU
  Appliquer la stratégie $FOURMI_{thread}$ ou $FOURMI_{bloc}$ sur le GPU_{id}
Récupérer la tournée T_{gl}^{id} de chaque colonie id et trouver la meilleure T_{gl}
Retourner la tournée la plus courte T_{gl}

Algorithme 4.4 – Pseudo-code de la stratégie $COLONIE_{gpu}$.

4.5 Résultats expérimentaux

Les expérimentations ont été réalisées sur un serveur contenant deux GPU Tesla C2050 de NVIDIA qui utilisent l'architecture Fermi. Chaque GPU contient 14 multiprocesseurs, 32 processeurs par multiprocesseur, 48 ko de mémoire partagée par multiprocesseur et une taille de warp de 32. Le serveur inclut également deux CPU quadri-cœur Xeon E5640 fonctionnant à 2,67 GHz et 24 Go de DDR3. Ces ressources matérielles sont disponibles au Centre de Calcul de Champagne-Ardenne ROMEO [rom11]. Les différentes implémentations ont été développées dans l'environnement de programmation "C pour CUDA V4.1".

Pour chaque approche $FOURMI$ et $COLONIE$, les stratégies de parallélisation GPU conçues dans les Sections 4.4.1 et 4.4.2 sont expérimentées et comparées sur de nombreux problèmes du VC dont les tailles varient de 51 à 13509 villes. Conformément aux principes expérimentaux adoptés par Stützle et Hoos [SH00], les paramètres ont été fixés à $\alpha = 1$, $\beta = 2$ et $\rho = 0,98$. Les minimums et moyennes des longueurs des tournées obtenues ont été calculés à partir de 25 essais pour les problèmes de moins de 1000 villes et de 10 essais pour les instances de plus grande taille. Stützle et Hoos fixent également le nombre de constructions de tournées à $nbc \cdot n$ avec $m = n$ et $nbc = \{2500, 10000\}$. Cependant, comme précisé dans le Chapitre 2, la documentation CUDA [cud11] conseille vivement d'utiliser un nombre de threads par bloc qui soit un multiple de la taille des warps (32) pour maximiser son efficacité. Afin de prendre en compte toutes les considérations, les valeurs de m ont été choisies telles qu'elles soient un multiple de 32. Par conséquent, le nombre d'itérations est fixé à $\frac{nbc \cdot n}{m}$ dans l'intention de garder le même nombre global de constructions de tournées. nbc est fixé à 10000 lorsque les algorithmes séquentiels proposés sont comparés avec les travaux de la littérature. Il est généralement fixé à 2500 lorsque les implémentations parallèles sont comparées aux implémentations séquentielles, sauf dans les cas particuliers où nbc est précisé.

Afin de mesurer la performance des implémentations parallèles, la métrique d'accélération est utilisée. Barr et Hickman [BH93] définissent la métrique d'accélération et la métrique d'accélération relative. L'accélération est le ratio entre le temps de résolution d'un problème avec le code séquentiel le plus rapide et le temps de résolution de ce même problème avec le code parallèle. L'accélération relative est le ratio entre le temps de résolution d'un problème avec le code parallèle sur un processeur et le temps de résolution de ce même problème avec le code parallèle sur plusieurs processeurs. Quant à lui, Alba [Alb02] définit l'accélération comme étant le ratio entre le temps d'exécution moyen sur un processeur $E[T_1]$ et le temps d'exécution moyen sur m processeurs $E[T_m]$:

$$accélération_m = \frac{E[T_1]}{E[T_m]} \tag{4.6}$$

Pour l'approche $FOURMI$ et la stratégie $COLONIE_{bloc}$ nécessitant seulement un GPU, les accélérations sont donc calculées en divisant le temps séquentiel CPU par le temps parallèle obtenu avec le même CPU et le GPU agissant comme co-processeur. Dans le cas de la stratégie $COLONIE_{gpu}$, le temps parallèle est obtenu avec autant de cœurs/CPU et de GPU qu'il y a de colonies, une combinaison CPU-GPU étant liée à chaque colonie :

$$accélération = \frac{temps\ séquentiel\ obtenu\ avec\ un\ cœur/CPU}{temps\ parallèle\ obtenu\ avec\ x\ cœurs/CPU\ et\ x\ GPU} \tag{4.7}$$

Les prochaines sections sont consacrées aux expérimentations des stratégies de parallélisation proposées pour les approches $FOURMI$ et $COLONIE$. Ces stratégies sont évaluées et comparées autant au niveau de la qualité des solutions que de la réduction du temps de calcul.

4.5.1 Approche $FOURMI$

L'approche $FOURMI$ utilise une seule colonie où chaque fourmi construit sa solution en parallèle. Dans la Section 4.3, il est précisé que le MMAS peut inclure des listes de candidats. Puisqu'elles permettent de réduire le temps de résolution de l'algorithme séquentiel, elles peuvent également réduire le temps d'exécution des implémentations parallèles. De plus, la stratégie $FOURMI_{bloc}$ effectue les évaluations des villes candidates en parallèle en les associant aux threads des blocs. Ces listes peuvent donc avoir une influence sur l'efficacité des stratégies de parallélisation. Une comparaison du MMAS avec et sans candidats est alors effectuée pour étudier cette influence. La prochaine section est dédiée aux expérimentations sur le MMAS sans listes de candidats.

4.5.1.1 Sans listes de candidats

Avant de pouvoir comparer les stratégies de parallélisation entre elles, il est nécessaire de les paramétrer afin qu'elles soient les plus performantes possibles. Cependant, le nombre de blocs et de threads par bloc peut avoir une grande influence sur le temps d'exécution d'une stratégie. En effet, le Chapitre 2 et la documentation CUDA conseillent d'utiliser un nombre de threads par bloc qui soit un multiple de 32 pour maximiser l'efficacité de la stratégie. Il est également spécifié que les retards introduits par les dépendances lecture-après-écriture des registres peuvent être ignorées si les multiprocesseurs contiennent au moins 192 threads actifs pour les masquer. De plus, le nombre de blocs actifs dépend du nombre de registres utilisés et de la quantité de mémoire partagée occupée, ces ressources étant partagées parmi tous les threads composant les blocs.

La première partie des expérimentations proposées consiste donc à étudier l'influence du paramétrage du nombre de blocs et de threads par bloc sur l'accélération obtenue par les stratégies $FOURMI_{thread}$ et $FOURMI_{bloc}$. Stützle et Hoos [SH00] recommandent d'utiliser un nombre de fourmis m égal au nombre de villes n qui n'est généralement pas un multiple de la taille des warps. Afin de prendre en compte toutes les considérations, les valeurs de m ont

été choisies telles qu'elles soient un multiple de la taille des warps le plus proche possible de n. Dans ce cas, $m = 32 \cdot 2^x$ où x est un nombre entier. Le Tableau 4.1 indique le nombre de fourmis m choisi pour chaque problème.

Problème	eil51	kroA100	d198, lin318	rat783	fl1577	d2103	pcb3038, fnl4461, rl5915	usa13509
m	64	128	256	512	1024	2048	4096	8192

Tableau 4.1 – Nombre de fourmis m pour chaque problème.

La stratégie $FOURMI_{thread}$ associe chaque fourmi à un thread. Il est donc nécessaire de répartir ces fourmis/threads au sein de blocs. Plusieurs configurations de blocs et de threads qui vérifient $m = \textit{nombre de blocs} \cdot \textit{nombre de threads}$ ont été testées pour des problèmes allant de 51 à 783 villes. Par exemple, pour le problème de 51 villes qui utilise 64 fourmis, les configurations utilisées sont : 64 blocs de 1 thread, 32 blocs de 2 threads, ... , 1 bloc de 64 threads. Le Tableau 4.2 présente les accélérations obtenues ainsi que le nombre de blocs actifs pour chaque configuration de blocs et de threads testée.

Nombre de threads	Nombre de blocs actifs	eil51	kroA100	d198	lin318	rat783
1	8	**2,58**	2,80	3,61	3,52	4,08
2	8	1,95	**4,06**	4,15	3,94	4,97
4	8	1,55	3,15	**5,81**	**5,47**	5,55
8	8	1,18	2,32	4,27	4,05	**7,47**
16	8	0,82	1,58	2,87	2,73	5,20
32	8	0,52	0,98	1,76	1,70	3,32
64	8	0,52	0,97	1,74	1,67	3,31
128	8	-	0,94	1,64	1,59	3,18
256	6	-	-	1,53	1,49	3,01
512	3	-	-	-	-	2,34

Tableau 4.2 – Accélérations obtenues pour la stratégie $FOURMI_{thread}$ en faisant varier le nombre de blocs et de threads.

L'analyse des résultats suggère que, pour cette stratégie, il vaut mieux choisir un grand nombre de blocs contenant peu de threads (1, 2, 4 ou 8) plutôt qu'un petit nombre de blocs contenant beaucoup de threads (de 16 à 512). Cette observation est donc contradictoire avec les recommandations de NVIDIA conseillant de choisir un nombre de threads multiple de 32. Cependant, il est précisé dans le Chapitre 2 que, plus il y a de warps (et de blocs) actifs par multiprocesseur, plus la latence de la mémoire globale lente peut être masquée. Puisque la stratégie $FOURMI_{thread}$ utilise cette mémoire pour stocker les données des fourmis, elle y effectue un grand nombre d'accès qu'il faut dissimuler le plus possible. Pour illustrer cette explication, le problème de 51 villes est considéré. En choisissant 1 bloc de 64 threads, seulement un multiprocesseur est utilisé et deux warps de 32 threads sont exécutés. En choisissant 64 blocs de 1 thread, 8 multiprocesseurs exécutent chacun 8 blocs de 1 thread (8 warps) simultanément

puisque 8 blocs sont actifs. Dans ce cas, un plus grand nombre de warps sont actifs par multiprocesseur et la latence de la mémoire globale peut être mieux dissimulée. De plus, puisque les warps contiennent seulement un thread, la divergence de threads au sein d'un warp et la sérialisation de l'algorithme sont évitées. Il est alors bénéfique de lancer un grand nombre de blocs de petite taille.

Cependant, la meilleure configuration pour le problème de 100 villes n'est pas 128 blocs de 1 thread comme pouvaient le présager les observations précédentes, mais 64 blocs de 2 threads. Puisque 8 blocs sont actifs par multiprocesseur, la configuration de 128 blocs de 1 thread nécessite 16 multiprocesseurs ($\frac{128\ blocs}{8\ blocs\ actifs}$) alors que la configuration de 64 blocs de 2 threads requiert 8 multiprocesseurs ($\frac{64\ blocs}{8\ blocs\ actifs}$). En revanche, le GPU utilisé contient 14 multiprocesseurs. La configuration de 128 blocs de 1 thread permet donc l'exécution simultanée de 112 blocs ($8\ blocs\ actifs \cdot 14\ multiprocesseurs$) puis de 14 blocs. Quant à elle, la configuration de 64 blocs de 2 threads permet l'exécution de 64 blocs simultanément. Il est alors bénéfique d'ajouter quelques threads au sein de chaque bloc plutôt que de ne pas pouvoir exécuter tous les blocs simultanément. Les meilleures configurations maximisent donc le nombre de blocs lancés sans dépasser 112 blocs : 64 blocs de 1 thread pour eil51, 64 blocs de 2 threads pour kroA100, 64 blocs de 4 threads pour d198 et lin318 et 64 blocs de 8 threads pour rat783.

La stratégie $FOURMI_{bloc}$ associe chaque fourmi à un bloc et les différentes villes candidates aux threads. Pour chaque problème, m blocs sont donc utilisés. Un thread peut alors s'occuper de l'exécution d'une ou de plusieurs villes. n threads sont tout d'abord testés puis des nombres de threads inférieurs ou égal à n, multiples de 32. Les Tableaux 4.3 et 4.4 présentent les accélérations obtenues pour les deux variantes de la stratégie $FOURMI_{bloc}$ ainsi que le nombre de blocs actifs lorsque le nombre de threads varie.

Nombre de threads	eil51		kroA100		d198		lin318		rat783	
n	**5,69**	(8)	**7,15**	(8)	7,73	(6)	6,37	(4)	1,89	(1)
32	5,24	(8)	6,41	(8)	8,77	(8)	9,16	(8)	11,14	(8)
64	-	-	6,78	(8)	9,07	(8)	9,54	(8)	11,38	(8)
128	-	-	-	-	**9,32**	(8)	**9,75**	(8)	**11,73**	(8)
256	-	-	-	-	-	-	7,68	(5)	8,32	(5)
512	-	-	-	-	-	-	-	-	3,64	(2)

Tableau 4.3 – Accélérations obtenues et nombre de blocs actifs (entre parenthèses) pour la stratégie $FOURMI_{bloc}^{globale}$ en faisant varier le nombre de threads.

Ces résultats montrent que, pour les petits problèmes eil51 et kroA100, les meilleures accélérations sont produites avec le nombre maximum de threads (51 et 100) même si ce dernier n'est pas un multiple de 32. Pour les autres problèmes, les meilleures accélérations sont obtenues avec 128 threads. Par exemple, le problème 198 villes utilise 256 fourmis/blocs. Lorsque 128 threads sont choisis, 8 blocs sont actifs. Cette configuration permet donc l'exécution simultanée de 112 blocs, de 112 blocs puis de 32 blocs ($256 = 8 \cdot 14 + 8 \cdot 14 + 32$), correspondant à 3 "exécutions". Lorsque 198 threads sont choisis, il ne reste que 6 blocs actifs. Cette configuration permet l'exécution simultanée de 84 blocs, de 84 blocs, de 84 blocs puis de 4 blocs ($256 = 6 \cdot 14 + 6 \cdot 14 + 6 \cdot 14 + 4$), correspondant à 4 "exécutions". Il est donc davantage bénéfique

Nombre de threads	eil51		kroA100		d198		lin318		rat783	
n	**7,54**	(8)	**9,71**	(8)	10,74	(6)	8,87	(4)	2,64	(1)
32	7,15	(8)	9,09	(8)	12,46	(8)	13,11	(8)	15,94	(8)
64	-	-	9,29	(8)	12,41	(8)	13,10	(8)	15,77	(8)
128	-	-	-	-	**12,58**	(8)	**13,26**	(8)	**16,19**	(8)
256	-	-	-	-	-	-	10,68	(5)	11,55	(5)
512	-	-	-	-	-	-	-	-	5,10	(2)

Tableau 4.4 – Accélérations obtenues et nombre de blocs actifs (entre parenthèses) pour la stratégie $FOURMI_{bloc}^{partagée}$ en faisant varier le nombre de threads.

d'utiliser 128 threads et d'obtenir 8 blocs actifs que d'utiliser un plus grand nombre de threads impliquant moins de blocs actifs. En effet, un plus grand nombre de blocs et de fourmis peuvent être exécutés à la fois.

Pour résumer, les meilleures configurations obtenues pour la stratégie $FOURMI_{thread}$ utilisent 64 blocs et 1 thread pour eil51, 2 threads pour kroA100, 4 threads pour d198 et lin318 et 8 threads pour rat783. Pour la stratégie $FOURMI_{bloc}$, les meilleures configurations utilisent m blocs de n threads pour les problèmes eil51 et kroA100 et de 128 threads pour les autres.

La seconde partie des expérimentations consiste à évaluer la qualité des solutions et les accélérations obtenues par les stratégies de parallélisation GPU proposées. Afin de les valider, il est essentiel de comparer la version séquentielle du MMAS proposée avec celle présentée dans les travaux de Stützle et Hoos [SH97]. Le Tableau 4.5 fournit donc les longueurs des tournées minimales et moyenne obtenues pour chaque problème ainsi que le pourcentage de déviation moyen entre les deux algorithmes. Les résultats montrent que les solutions obtenues avec le MMAS séquentiel proposé sont de meilleure qualité que celles obtenues avec l'algorithme présenté par Stützle et Hoos : de −1,09% à −0,14% de déviation moyenne par rapport aux travaux de la littérature.

Problème	Optimum	Stützle et Hoos	MMAS séquentiel		
		Moyenne	Minimum	Moyenne	
eil51	426	427,20	426	426,60	(−0,14)
kroA100	21282	21352,05	21282	21300,52	(−0,24)
d198	15780	16095,95	15850	15920,28	(−1,09)

Tableau 4.5 – Longueur minimale et moyenne des solutions obtenues et pourcentage de déviation moyen (entre parenthèses) du MMAS séquentiel proposé par rapport à la version originale de Stützle et Hoos [SH97].

Ensuite, la qualité des solutions obtenues par les versions parallèles du MMAS sont comparées à la version séquentielle. Le Tableau 4.6 présente les longueurs des tournées minimales et moyennes obtenues pour chaque problème ainsi que le pourcentage de déviation des implémentations parallèles par rapport à l'algorithme séquentiel. Dans certains cas, la qualité des solutions est légèrement améliorée ou dégradée : de +0,29% à −1,33% de déviation moyenne par rapport à l'algorithme séquentiel. Dans l'ensemble, les solutions obtenues sont de meilleure

qualité avec en moyenne −0,14%, −0,89% et −0,34% de déviation pour les trois stratégies de parallélisation.

Problème	Optimum	Séquentiel	$FOURMI_{thread}$		$FOURMI_{bloc}^{globale}$		$FOURMI_{bloc}^{partagée}$	
eil51	426	426	426	(±0,00)	426	(±0,00)	426	(±0,00)
		427,80	427,56	(−0,06)	427,96	(+0,04)	427,80	(±0,00)
kroA100	21282	21282	21282	(±0,00)	21282	(±0,00)	21282	(±0,00)
		21345,76	21345,12	(±0,00)	21342,46	(−0,02)	21346,36	(±0,00)
d198	15780	16163	16159	(−0,03)	16102	(−0,38)	16164	(+0,01)
		16376,20	16420,72	(+0,27)	16396,80	(+0,13)	16395,20	(+0,12)
lin318	42029	42597	42410	(−0,44)	42623	(+0,06)	42489	(−0,25)
		43174,44	43047,68	(−0,29)	43105,24	(−0,16)	42967,80	(−0,48)
rat783	8806	9703	9636	(−0,69)	9510	(−1,99)	9614	(−0,92)
		10325,92	10262,60	(−0,61)	10281,64	(−0,43)	10188,28	(−1,33)
Moyenne du pourcentage de déviation			minimum : (−0,23) moyen : (−0,14)		minimum : (−0,46) moyen : (−0,89)		minimum : (−0,23) moyen : (−0,34)	

Tableau 4.6 – Longueur minimale et moyenne des solutions obtenues et pourcentage de déviation (entre parenthèses) des versions parallèles par rapport à la version séquentielle.

Après avoir validé les stratégies de parallélisation au niveau de la qualité des solutions, la réduction du temps d'exécution obtenue pour chacune est évaluée et comparée. Les accélérations ont été calculées à partir des temps séquentiels fournis dans le Tableau 4.7 et sont présentées dans la Figure 4.9.

Problème	MMAS
eil51	8,56
kroA100	63,73
d198	459,81
lin318	1931,34
rat783	29630,09

Tableau 4.7 – Temps séquentiel de l'algorithme MMAS en secondes.

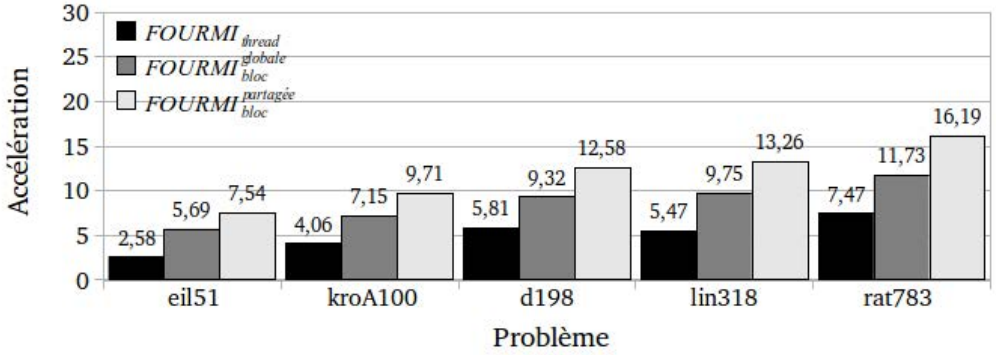


Figure 4.9 – Accélérations des stratégies de parallélisation du MMAS.

Il peut tout d'abord être observé que, généralement, plus la taille du problème augmente et plus l'accélération augmente. Une quantité de travail plus importante permet une meilleure

utilisation des ressources GPU disponibles. Ensuite, la meilleure accélération obtenue par la stratégie $FOURMI_{thread}$ est 7,47, largement inférieure aux accélérations de 11,73 et 16,19 pour les deux variantes de la stratégie $FOURMI_{bloc}$. En général, les accélérations de la stratégie $FOURMI_{thread}$ sont environ deux fois plus basses que celles de la stratégie $FOURMI_{bloc}$. Cette grande différence vient principalement de la divergence de code induite par le calcul de la règle de transition d'état de plusieurs fourmis du même bloc en mode SIMT, mais aussi du petit nombre de blocs et de threads qui ne dissimule pas efficacement les latences mémoire.

Les meilleures accélérations obtenues par la stratégie $FOURMI_{bloc}^{globale}$, allant de 5,69 à 11,73, montrent que partager le travail de chaque fourmi entre plusieurs threads est plus efficace. Distribuer le travail global sur un plus grand nombre de threads est donc bénéfique pour l'exécution parallèle sur GPU. Par exemple, pour 783 villes, cette stratégie dispose d'un total de 65536 threads contre 512 pour la stratégie $FOURMI_{thread}$. De plus, utiliser différents blocs favorise l'indépendance des fourmis durant l'exécution GPU. Cela montre que le modèle de calcul SIMT du GPU est mieux adapté au calcul de la règle de transition d'état par des fourmis seules que par plusieurs fourmis à la fois. Les résultats montrent également qu'assigner les fourmis aux blocs apporte des améliorations supplémentaires en utilisant la mémoire partagée. En effet, la stratégie $FOURMI_{bloc}^{partagée}$ fournit les meilleures accélérations allant de 7,54 à 16,19.

Afin de réduire le temps d'exécution des algorithmes pour résoudre des problèmes de plus grande taille et d'améliorer la qualité des solutions, la prochaine section présente les résultats obtenus avec l'algorithme du MMAS utilisant des listes de candidats pour calculer la règle de transition d'état.

4.5.1.2 Avec listes de candidats

Afin de valider les stratégies de parallélisation comprenant des listes de candidats, la version séquentielle du MMAS proposée est comparée avec celle présentée dans les travaux de Stützle et Hoos [SH00]. Conformément aux principes expérimentaux adoptés par Stützle et Hoos, la taille des listes de candidats est fixée à 20. Le Tableau 4.8 fournit les longueurs des tournées minimales et moyenne obtenues pour chaque problème ainsi que le pourcentage de déviation moyen entre les deux algorithmes. Les résultats montrent que les solutions obtenues avec le MMAS séquentiel proposé sont de meilleure qualité que celles obtenues avec l'algorithme présenté par Stützle et Hoos : de $-0{,}08\%$ à $-0{,}04\%$ de déviation moyenne par rapport aux travaux de la littérature.

Problème	Optimum	Stützle et Hoos	MMAS séquentiel	
		Moyenne	Minimum	Moyenne
eil51	426	427,10	426	426,76 (−0,08)
kroA100	21282	21291,60	21282	21282,56 (−0,04)
d198	15780	15956,80	15840	15945,92 (−0,07)

Tableau 4.8 – Longueur minimale et moyenne des solutions obtenues et pourcentage de déviation moyen (entre parenthèses) du MMAS séquentiel proposé par rapport à celui présenté par Stützle et Hoos [SH00].

Ensuite, la qualité des solutions obtenues par les versions parallèles du MMAS sont comparées à la version séquentielle pour des problèmes allant de 51 à 13509 villes. Le paramétrage des stratégies de parallélisation est effectué selon l'étude de l'influence du nombre de blocs et de threads par bloc présentée dans la Section 4.5.1.1. Pour la stratégie $FOURMI_{thread}$, les configurations de blocs et de threads vérifient toujours $m = nombre\ de\ blocs \cdot nombre\ de\ threads$ avec 64 blocs utilisés pour chaque problème. Dans la stratégie $FOURMI_{bloc}$, m blocs sont utilisés, chacun d'entre eux étant composé d'un nombre de threads égal à la taille des listes de candidats (ici, 20). Le Tableau 4.9 présente les longueurs des tournées minimales et moyennes obtenues pour chaque problème ainsi que le pourcentage de déviation des implémentations parallèles par rapport à l'algorithme séquentiel. Pour le problème de 13509 villes utilisant 8192 fourmis, *nbc* a été fixé à 100 plutôt que 2500. Dans certains cas, la qualité des solutions est légèrement améliorée ou dégradée : de +0,59% à −0,48% de déviation moyenne par rapport à l'algorithme séquentiel. Dans l'ensemble, les solutions obtenues sont de meilleure qualité avec en moyenne −0,10%, −0,08% et +0,01% de déviation pour les trois stratégies de parallélisation.

Problème	Optimum	Séquentiel	$FOURMI_{thread}$	$FOURMI_{bloc}^{globale}$	$FOURMI_{bloc}^{partagée}$
eil51	426	426	426 (±0,00)	426 (±0,00)	426 (±0,00)
		427,48	427,44 (−0,01)	427,28 (−0,05)	427,40 (−0,02)
kroA100	21282	21282	21282 (±0,00)	21282 (±0,00)	21282 (±0,00)
		21317,40	21310,64 (−0,03)	21322,52 (+0,02)	21318,44 (±0,00)
d198	15780	15864	15876 (+0,08)	15861 (−0,02)	15855 (−0,06)
		15973,12	15975,04 (+0,01)	15964,40 (−0,05)	15974,08 (+0,01)
lin318	42029	42162	42184 (+0,05)	42091 (−0,17)	42160 (±0,00)
		42318,40	42321,52 (+0,01)	42325,28 (+0,02)	42331,84 (+0,03)
rat783	8806	8914	8901 (−0,15)	8913 (−0,01)	8917 (+0,03)
		9112,20	9092,12 (−0,22)	9068,64 (−0,48)	9166,16 (+0,59)
fl1577	22249	24364	24240 (−0,51)	24233 (−0,54)	24009 (−1,46)
		24533,10	24426,00 (−0,44)	24555,40 (+0,09)	24488,60 (−0,18)
d2103	80450	82636	82374 (−0,32)	82165 (−0,57)	82368 (−0,32)
		82999,10	82312,90 (−0,22)	82672,50 (−0,39)	82724,20 (−0,33)
pcb3038	137694	171062	170736 (−0,19)	170583 (−0,28)	170800 (−0,15)
		171741,90	171848,40 (+0,06)	171578,40 (−0,10)	171793,40 (+0,03)
fnl4461	182566	226059	227610 (+0,69)	227706 (+0,73)	227376 (+0,58)
		228534,90	228329,33 (−0,09)	228546,90 (+0,01)	228480,40 (−0,02)
rl5915	565530	636392	636690 (+0,05)	637427 (+0,16)	635858 (−0,08)
		640375,10	638619,67 (−0,27)	640451,10 (+0,01)	639135,30 (−0,19)
usa13509	19982859	28693319	28754635 (+0,21)	28640122 (−0,19)	28655859 (−0,13)
		28752516,30	28791921,00 (+0,14)	28728442,11 (−0,08)	28778402,80 (+0,09)
Moyenne du pourcentage de déviation			minimum : (−0,01) moyen : (−0,10)	minimum : (−0,08) moyen : (−0,08)	minimum : (−0,14) moyen : (+0,01)

Tableau 4.9 – Longueur minimale et moyenne des solutions obtenues et pourcentage de déviation (entre parenthèses) des versions parallèles par rapport à la version séquentielle.

La réduction du temps d'exécution obtenue pour chaque stratégie de parallélisation est ensuite évaluée et comparée. Les accélérations ont été calculées à partir des temps séquentiels fournis dans le Tableau 4.10 et sont présentées dans la Figure 4.10. Il peut être observé que pour les problèmes allant jusqu'à 783 villes, les accélérations suivent les mêmes tendances que celles obtenues avec le MMAS sans candidats. Les accélérations continuent d'augmenter globalement

jusqu'au problème de 4461 villes mais diminuent à partir de 5915 villes. Dans ce cas, la charge de travail et les grosses structures de données impliquent des latences mémoire et des conflits de banques dont les coûts augmentent plus vite que les bénéfices de la parallélisation du travail disponible.

Problème	Temps séquentiel
eil51	3,61
kroA100	14,87
d198	59,92
lin318	161,56
rat783	1350,59
fl1577	7553,01
d2103	10887,99
pcb3038	46274,67
fnl4461	198335,54
rl5915	286255,59
usa13509	187346,79

Tableau 4.10 – Temps séquentiel en secondes de l'algorithme MMAS avec candidats.

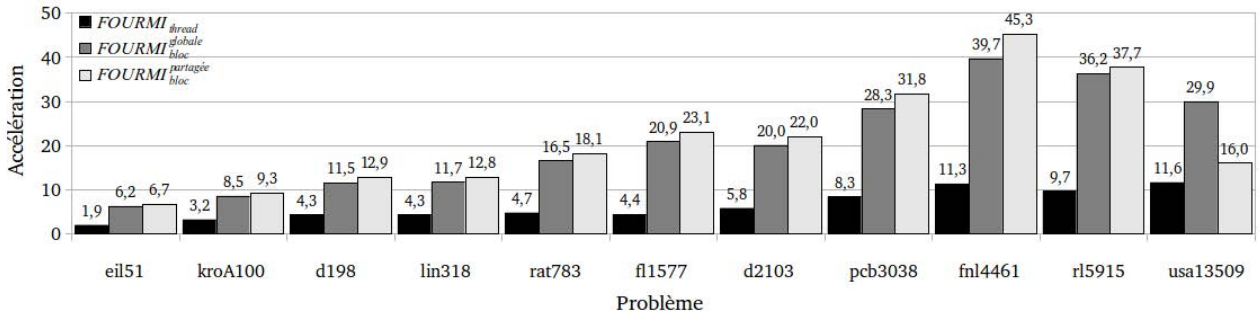


Figure 4.10 – Accélérations des stratégies de parallélisation du MMAS avec candidats.

Dans l'ensemble, les accélérations de la stratégie $FOURMI_{bloc}$ augmentent plus rapidement avec la taille du problème que la stratégie $FOURMI_{thread}$, comme avec le MMAS sans candidats. La meilleure accélération pour la stratégie $FOURMI_{thread}$ est de 11,59, largement inférieure aux accélérations 45,28 et 39,73 pour les deux variantes de la stratégie $FOURMI_{bloc}$. En général, les accélérations de la stratégie $FOURMI_{thread}$ sont environ trois fois plus basses que celles de la stratégie $FOURMI_{bloc}$ et le fossé devient encore plus large quand la taille du problème augmente. Les meilleures accélérations obtenues par la stratégie $FOURMI_{bloc}^{globale}$, allant de 6,22 à 39,73, montrent que le partage du travail de chaque fourmi entre plusieurs threads est plus efficace. Les résultats indiquent également qu'assigner les fourmis aux blocs apporte des améliorations supplémentaires en utilisant la mémoire partagée. En effet, la stratégie $FOURMI_{bloc}^{partagée}$ fournit les meilleures accélérations allant de 6,71 à 44,52. Cependant, la chute d'accélération entre 5915 et 13509 villes est brutale : l'accélération moyenne passe de 37,69 à 16,01. L'accélération obtenue par la stratégie $FOURMI_{bloc}^{globale}$ pour 13509 villes de 29,94 dépasse même celle obtenue par la stratégie $FOURMI_{bloc}^{partagée}$ de 16,01. En effet, la stratégie

$FOURMI_{bloc}^{globale}$ autorise toujours 8 blocs actifs par multiprocesseur alors qu'en utilisant la mémoire partagée, 7 blocs sont actifs pour 5915 villes et seulement 3 pour 13509 villes. Le nombre de blocs augmente mais le nombre de blocs actifs baisse. La petite taille de la mémoire partagée pénalise donc le temps de résolution des plus gros problèmes.

4.5.2 Approche *COLONIE*

Après avoir évalué l'approche de parallélisation utilisant une seule colonie, les stratégies proposées dans la Section 4.4.2 pour l'approche *COLONIE* sont également testées. Le même nombre total de constructions de tournées est conservé pour chaque configuration. Par conséquent, quand le nombre de colonies augmente, le nombre de fourmis par colonie diminue. Par exemple, pour le problème de 51 villes où 64 fourmis sont utilisées, deux colonies contiendront chacune 32 fourmis. Les deux prochaines sections présentent les résultats obtenus par les stratégies $COLONIE_{bloc}$ et $COLONIE_{gpu}$.

4.5.2.1 Stratégie $COLONIE_{bloc}$

Dans la stratégie $COLONIE_{bloc}$, chaque colonie est exécutée sur un bloc et les fourmis sont associées aux threads. Le nombre de blocs et de threads correspond donc au nombre de colonies et de fourmis par colonie respectivement. Les accélérations obtenues sont présentées dans la Figure 4.11 pour 1, 2, 10, 100, 250 et 500 colonies. La qualité des solutions minimum et moyenne est fournie dans le Tableau 4.11. Le pourcentage de déviation de la stratégie $COLONIE_{bloc}$ par rapport à la version séquentielle est présentée dans le Tableau 4.12. Le problème de 13509 villes ainsi que certaines configurations n'ont pas pu être testés car ils utilisaient 0 fourmi, plus de 1024 fourmis (nombre de threads maximum par bloc) ou une quantité de mémoire globale GPU supérieure à la quantité disponible (ici, 3 Go).

D'après la Figure 4.11, il peut être observé que les accélérations obtenues sont toutes inférieures comparativement à celles de la stratégie $FOURMI_{thread}$. Puisque toutes les fourmis d'une colonie sont associées aux threads d'un bloc, ces threads doivent se synchroniser durant la construction des tournées. De plus, les matrices de phéromone associées à chaque colonie sont mises à jour de façon à ne plus tirer avantage du cache texture. Dans l'ensemble, pour obtenir la meilleure accélération pour chaque problème, il faut utiliser un nombre moyen de colonies. Lorsque trop peu de colonies sont utilisées, et donc de blocs, les capacités du GPU ne sont pas bien exploitées. En outre, déporter la totalité de l'algorithme sur le GPU implique également un grand nombre de registres qui diminue considérablement le nombre de blocs actifs par multiprocesseur. Par exemple, 8 blocs sont actifs quand les colonies contiennent 128 fourmis mais plus que 2 blocs sont actifs quand les colonies contiennent 512 fourmis. Lorsque trop de colonies sont lancées, la quantité de mémoire utilisée et le grand nombre de transferts mémoire effectués pénalisent l'exécution de l'algorithme. De plus, les latences mémoires ne sont pas bien dissimulées.

L'analyse des Tableaux 4.11 et 4.12 montre que plus le nombre de colonies est grand, plus la qualité des solutions est dégradée. En effet, lorsque 2 colonies sont exécutées, une dégradation moyenne de 0,35% par rapport à la version séquentielle est constatée et lorsque

500 colonies sont exécutées, cette dégradation augmente à 14,79%. Dans l'ensemble, en utilisant cette configuration simple de division du travail et de paramétrage, cette stratégie offre un potentiel limité.

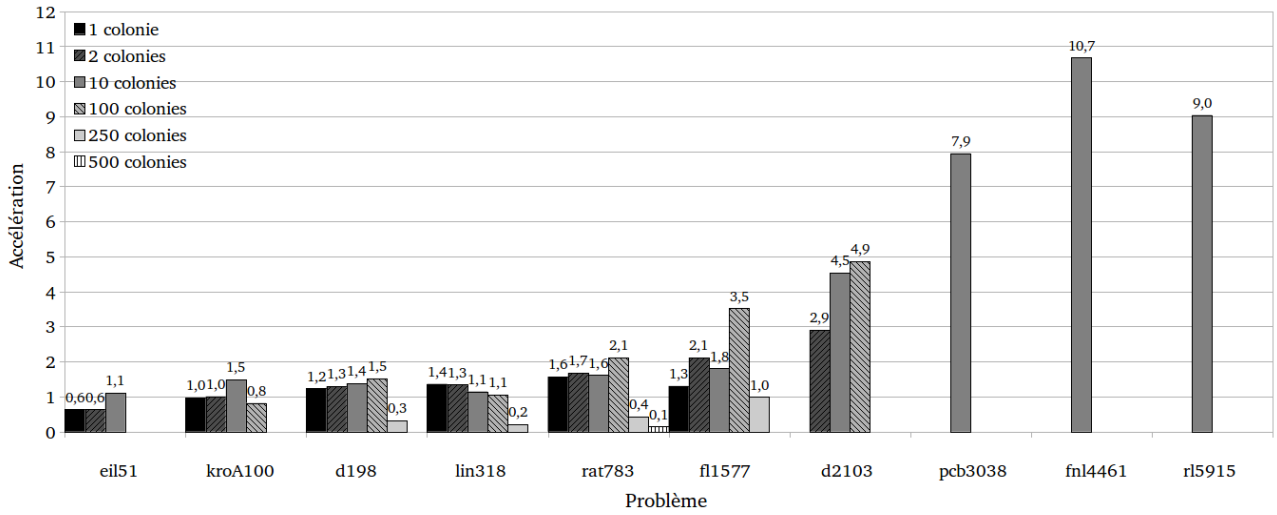


Figure 4.11 – Accélérations de la stratégie $COLONIE_{bloc}$ obtenues avec 1 à 500 colonies.

Problème	Optimum	Séquentiel	Nombre de colonies					
			1	2	10	100	250	500
eil51	426	426 427,48	426 427,60	426 427,24	426 429,48	- -	- -	- -
kroA100	21282	21282 21317,40	21282 21329,76	21282 21317,32	21292 21499,32	22185 22519,64	- -	- -
d198	15780	15864 15973,12	15941 15976,92	15956 16012,08	16262 16449,92	16801 16964,48	16761 17016,68	- -
lin318	42029	42162 42318,40	42091 42334,40	42163 42280,28	42656 43096,72	45928 46251,12	46000 46648,80	- -
rat783	8806	8914 9112,20	8916 9068,40	9087 9310,92	9667 9873,76	10174 10261,00	10192 10373,36	10314 10459,76
fl1577	22249	24364 24533,10	24139 24494,90	24385 24686,80	24550 24998,20	25156 25426,50	25390 25579,40	- -
d2103	80450	82636 82999,10	- -	82428 82654,50	82720 83005,10	84351 84609,30	- -	- -
pcb3038	137694	171062 171741,90	- -	- -	171660 172527,60	- -	- -	- -
fnl4461	182566	226059 228534,90	- -	- -	226839 229615,40	- -	- -	- -
rl5915	565530	636392 640375,10	- -	- -	647804 647878,00	- -	- -	- -

Tableau 4.11 – Longueur minimale et moyenne des solutions obtenues par la version séquentielle de l'algorithme MMAS et la stratégie de parallélisation $COLONIE_{bloc}$.

Problème		Nombre de colonies					
		1	2	10	100	250	500
eil51	Δ^{seq}_{min}	±0,00	±0,00	±0,00	-	-	-
	Δ^{seq}_{moy}	+0,03	−0,06	+0,47	-	-	-
kroA100	Δ^{seq}_{min}	±0,00	±0,00	+0,05	+4,24	-	-
	Δ^{seq}_{moy}	+0,06	±0,00	+0,85	+5,64	-	-
d198	Δ^{seq}_{min}	+0,49	+0,58	+2,51	+5,91	+5,65	-
	Δ^{seq}_{moy}	+0,02	+0,24	+2,99	+6,21	+6,53	-
lin318	Δ^{seq}_{min}	−0,17	±0,00	+1,17	+8,93	+9,10	-
	Δ^{seq}_{moy}	+0,04	−0,09	+1,84	+9,29	+10,23	-
rat783	Δ^{seq}_{min}	+0,02	+1,94	+8,45	+14,14	+14,34	+15,71
	Δ^{seq}_{moy}	−0,48	+2,18	+8,36	+12,61	+13,84	+14,79
fl1577	Δ^{seq}_{min}	−0,92	+0,09	+0,76	+3,25	+4,21	-
	Δ^{seq}_{moy}	−0,16	+0,63	+1,90	+3,64	+4,26	-
d2103	Δ^{seq}_{min}	-	−0,25	+0,10	+2,08	-	-
	Δ^{seq}_{moy}	-	−0,42	+0,01	+1,94	-	-
pcb3038	Δ^{seq}_{min}	-	-	+0,35	-	-	-
	Δ^{seq}_{moy}	-	-	+0,46	-	-	-
fnl4461	Δ^{seq}_{min}	-	-	+0,35	-	-	-
	Δ^{seq}_{moy}	-	-	+0,47	-	-	-
rl5915	Δ^{seq}_{min}	-	-	+1,79	-	-	-
	Δ^{seq}_{moy}	-	-	+1,17	-	-	-
Moyenne du pourcentage de déviation	minimum	−0,10	+0,34	+1,55	+6,43	+8,33	+15,71
	moyen	−0,08	+0,35	+1,85	+6,56	+8,72	+14,79

Tableau 4.12 – Pourcentage de déviation minimum Δ^{seq}_{min} et moyen Δ^{seq}_{moy} de la stratégie de parallélisation $COLONIE_{bloc}$ par rapport à la version séquentielle de l'algorithme MMAS.

4.5.2.2 Stratégie $COLONIE_{gpu}$

Dans le but d'appliquer la stratégie $COLONIE_{gpu}$ sur les problèmes étudiés, les deux GPU disponibles sont utilisés. La stratégie $FOURMI^{globale}_{bloc}$ est intégrée au sein de chaque colonie car elle offre les meilleures performances sans présenter les limites de la mémoire partagée pour les plus gros problèmes. Le nombre de blocs est égal au nombre de fourmis par colonie et le nombre de threads, au nombre de candidats. Les accélérations sont présentées dans la Figure 4.12 pour 1 et 2 colonies. La qualité des solutions obtenues minimum et moyenne, ainsi que le pourcentage de déviation de la stratégie de parallélisation $COLONIE_{gpu}$ par rapport à la version séquentielle, sont fournis dans le Tableau 4.13.

Lorsque deux colonies sont utilisées, des accélérations allant jusqu'à 72,62 peuvent être obtenues. Les accélérations sont d'ailleurs quasiment doublées dans tous les cas comparé à une simple colonie. Cela montre qu'une parallélisation multi-GPU est efficace. Cependant, utiliser un plus petit nombre de fourmis par colonie dégrade légèrement la qualité des solutions obtenues : 0,30% en moyenne et jusqu'à 1,64% de déviation par rapport à la version séquentielle. Dans ce contexte, une façon d'améliorer la longueur des tournées serait d'ajouter des échanges d'information entre les colonies.

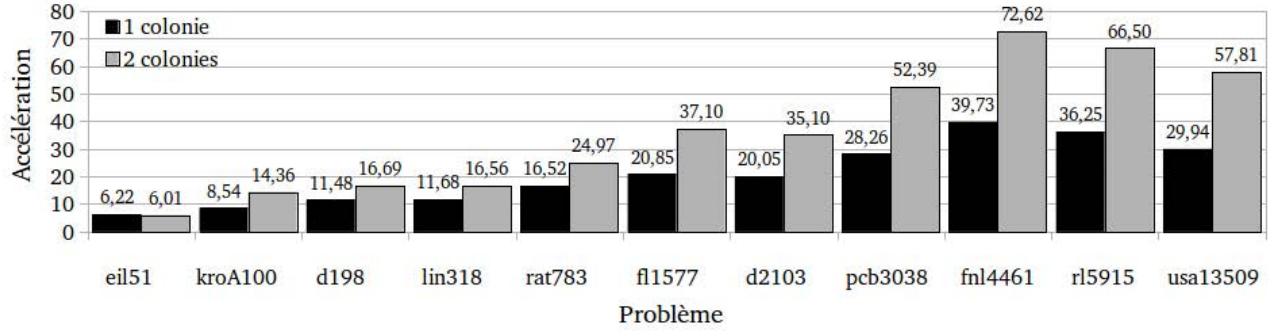


Figure 4.12 – Accélérations de la stratégie $COLONIE_{gpu}$ obtenues avec 1 et 2 colonies.

Problème	Optimum	Séquentiel	1 colonie		2 colonies	
eil51	426	426	426	(±0,00)	426	(±0,00)
		427,48	427,28	(−0,05)	427,00	(−0,11)
kroA100	21282	21282	21282	(±0,00)	21282	(±0,00)
		21317,40	21322,52	(+0,02)	21329,60	(+0,06)
d198	15780	15864	15861	(−0,02)	15956	(+0,58)
		15973,12	15964,40	(−0,05)	16032,88	(+0,37)
lin318	42029	42162	42091	(−0,17)	42162	(±0,00)
		42318,40	42325,28	(+0,02)	42311,08	(−0,02)
rat783	8806	8914	8913	(−0,01)	9009	(+1,07)
		9112,20	9068,64	(−0,48)	9262,00	(+1,64)
fl1577	22249	24364	24233	(−0,54)	24551	(+0,77)
		24533,10	24555,40	(+0,09)	24731,00	(+0,81)
d2103	80450	82636	82165	(−0,57)	82405	(−0,28)
		82999,10	82672,50	(−0,39)	82684,30	(−0,38)
pcb3038	137694	171062	170583	(−0,28)	171421	(+0,21)
		171741,90	171578,40	(−0,10)	172385,10	(+0,37)
fnl4461	182566	226059	227706	(+0,73)	228343	(+0,01)
		228534,90	228546,90	(+0,01)	228883,90	(+0,15)
rl5915	565530	636392	637427	(+0,16)	638588	(+0,35)
		640375,10	640451,10	(+0,01)	643415,30	(+0,47)
usa13509	19982859	28693319	28640122	(−0,19)	28672351	(−0,07)
		28752516,30	28728442,11	(−0,08)	28737407,90	(−0,05)
Moyenne du pourcentage de déviation			minimum :	(−0,08)	minimum :	(+0,24)
			moyen :	(−0,09)	moyen :	(+0,30)

Tableau 4.13 – Longueur minimale et moyenne des solutions obtenues et pourcentage de déviation (entre parenthèses) de la stratégie $COLONIE_{gpu}$ par rapport à la version séquentielle.

4.6 Conclusion

Le but de ce chapitre était de proposer des stratégies de parallélisation efficaces pour l'implémentation de l'Optimisation par Colonie de Fourmis sur GPU et de montrer les gains de performance pouvant être obtenus. Les stratégies $FOURMI_{thread}$ et $FOURMI_{bloc}$ visent à associer la construction des tournées des fourmis à l'exécution des processeurs et des multiprocesseurs respectivement. D'après la taxonomie proposée dans le Chapitre 3, elles sont de type $pop_{gpu}sol_{th}$ et $pop_{gpu}sol_{bl}elt_{th}$ respectivement. Deux variantes de la stratégie $FOURMI_{bloc}$ ont été proposées : celle de type $pop_{gpu}sol_{bl}^{par}elt_{th}$ stocke les données de chaque fourmi en mémoire partagée alors que celle de type $pop_{gpu}sol_{bl}^{gl}elt_{th}$ les conserve dans la mémoire globale. D'autre part, les stratégies $COLONIE_{bloc}$ et $COLONIE_{gpu}$ utilisent plusieurs colonies qu'elles attribuent aux multiprocesseurs et aux GPU respectivement. Elles sont de type $pop_{bl}sol_{th}$ et $pop_{gpu}sol_{bl}elt_{th}$ respectivement.

Les résultats des expérimentations ont montré que les deux approches $FOURMI$ et $COLONIE$ pouvaient être efficacement implémentées sur une architecture GPU. En effet, la stratégie $FOURMI_{bloc}$ fournit des accélérations allant jusqu'à 44,52 tout en obtenant une meilleure qualité de solution que les travaux de la littérature. Les accélérations s'élèvent encore plus haut avec la stratégie $COLONIE_{gpu}$ qui atteint une accélération maximale de 72,62 avec une légère dégradation de la qualité des solutions pour les plus problèmes de plus grande taille. Dans l'ensemble, ces expérimentations prouvent qu'il est possible de réduire considérablement le temps d'exécution des algorithmes OCF sur GPU. À titre de comparaison, Li *et al.* [LHPQ09] résolvaient le problème du VC avec l'algorithme MMAS et obtenaient une accélération maximale de 11 avec une qualité de solution similaire à celle de l'algorithme CPU. Cecilia *et al.* [CGU+11, CGN+12, CNA+12] résolvaient ce même problème avec l'algorithme AS et obtenaient une qualité de solution similaire ou légèrement meilleure à celle obtenue par la version séquentielle. Les phases parallèles de construction des tournées et de gestion de la phéromone permettaient alors d'obtenir des accélérations de 21 et 20.

Cette étude comparative souligne l'impact de la granularité de la parallélisation sur la performance des stratégies de parallélisation. Les stratégies $FOURMI_{thread}$ et $COLONIE_{bloc}$, qui associent toutes les deux la fourmi à un thread, apportent toujours les plus petites accélérations. Elles impliquent une divergence de threads et une sérialisation de l'algorithme. De plus, elles nécessitent un plus grand apport de travail pour pouvoir utiliser au maximum les ressources du GPU. Quant à elles, les stratégies $FOURMI_{bloc}$ et $COLONIE_{gpu}$ de type $pop_{gpu}sol_{bl}elt_{th}$, montrent que partager le travail disponible sur un plus grand nombre d'éléments de calcul est plus efficace. Cette étude souligne également l'impact des paramètres de l'OCF et des configurations techniques du GPU. En effet, selon les configurations blocs/threads utilisées, les accélérations peuvent être multipliées jusqu'à 3,5 fois.

Cependant, comme c'est le cas dans le domaine de l'OCF parallèle et des métaheuristiques parallèles en général, il reste encore beaucoup de travail à effectuer pour utiliser efficacement la puissance de calcul des GPU. En effet, la variété des stratégies proposées et l'étude comparative approfondie fournie dans ce chapitre apporte son lot de questions et de pistes de recherche. Par exemple, même si l'utilisation de la mémoire partagée du GPU conduit à un gain significatif de temps, sa petite taille pénalise le temps de résolution des plus gros problèmes. De nouvelles technologies matérielles fournissant une mémoire partagée plus importante pourront résoudre

partiellement ces problèmes. Néanmoins, il est peu probable que cette progression puisse compenser la taille et la complexité des problèmes à résoudre. Un compromis entre l'utilisation des différentes mémoires doit donc être trouvé. Une nouvelle représentation des fourmis pourrait également être proposée pour que ses données puissent être contenues dans la mémoire partagée, peu importe la taille du problème. De plus, l'exploitation maximale des ressources du GPU nécessitent souvent des configurations algorithmiques qui ne laissent pas l'OCF effectuer une exploration et une exploitation efficaces de l'espace de recherche. Il pourrait donc être pertinent de repenser la dynamique de fonctionnement de la métaheuristique de sorte à trouver un meilleur équilibre entre le parallélisme et la qualité de l'optimisation.

Les métaheuristiques les plus performantes intègrent un mécanisme de Recherche Locale. Ce dernier vise à améliorer les solutions obtenues mais implique également de nouvelles problématiques. Par exemple, les Recherches Locales qui résolvent le problème du VC requièrent peu de calculs par rapport à la grande quantité de lectures et d'écritures effectuées dans les mémoires du GPU. Cependant, elles offrent un potentiel de parallélisation intéressant quand elles sont étendues à une approche à base de population où différents individus améliorent leur solution en exécutant le même algorithme sur plusieurs unités de calcul. Le chapitre suivant est donc dédié à la parallélisation GPU du mécanisme de Recherche Locale intégré au sein de la métaheuristique de Recherche Locale Itérée.

Chapitre 5

Stratégies de parallélisation GPU pour la Recherche Locale Itérée

Les métaheuristiques sont bien souvent plus performantes quand elles sont hybridées avec un mécanisme de Recherche Locale (RL). Par conséquent, la phase de RL doit être parallélisée efficacement pour obtenir une méthode de résolution globalement performante. Dans [DDGK13], une RL a été intégrée aux stratégies de parallélisation de l'OCF présentées dans le Chapitre 4. Ces travaux ont montré que la parallélisation GPU de cette méthode implique de nouvelles problématiques, comme les nombreux accès à la mémoire globale lente du GPU. La RL trouve des optima locaux souvent éloignés de l'optimalité mais son intégration à un mécanisme de guidage tel que défini dans la métaheuristique de Recherche Locale Itérée (RLI) permet généralement de remédier à ce problème. La RLI est d'ailleurs considérée comme une des méthodes approchées les plus performantes pour le problème du Voyageur de Commerce (VC) [HS04]. De plus, elle n'est pas influencée par des mécanismes plus complexes de plusieurs autres métaheuristiques, par exemple la mise à jour de la phéromone pour les algorithmes OCF. Le but de ce chapitre est donc de proposer des approches de parallélisation de RL performantes adaptées à l'environnement de calcul GPU et intégrées dans la métaheuristique à solution unique RLI. Les principales questions algorithmiques, techniques et de gestion de la mémoire sont également abordées dans ce contexte.

Ce chapitre est organisé comme suit. Le fonctionnement général des algorithmes de la RL et de la RLI pour le problème du VC est tout d'abord explicité. Le problème du VC ainsi que la Recherche Locale 3-opt sont utilisés comme cadre de référence pour la RLI afin de comparer les résultats obtenus avec ceux décrits dans les travaux de Stützle et Hoos [SH01] et Lourenço *et al.* [LMS10], mais également avec les précédents résultats obtenus avec l'OCF. Plusieurs stratégies parallèles de la RLI sur GPU sont ensuite proposées. Elles reposent sur de multiples recherches qui permettent de choisir le nombre de solutions à gérer. Comme pour la parallélisation GPU de l'OCF, ces stratégies diffèrent par leur définition d'élément de calcul (processeur ou multiprocesseur) et par leur utilisation des mémoires du GPU (mémoire partagée ou mémoire globale). Cependant, elles se distinguent également par la règle de pivotement utilisée : *first-improvement* et *best-improvement* ; chacune faisant l'objet d'une approche : $RLI - FI$ et $RLI - RKBI$ respectivement.

La méthodologie de parallélisation employée pour les algorithmes OCF est transposée

à la première approche $RLI - FI$. Les stratégies parallèles sont donc de type : $pop_{gpu}sol_{th}$, $pop_{gpu}sol^{par}_{bl}elt_{th}$ et $pop_{gpu}sol^{gl}_{bl}elt_{th}$ selon la taxonomie proposée dans le Chapitre 3. Dans le premier cas, la notion d'élément de calcul est associé au processeur et chaque RL correspond à un thread. Dans les deux autres cas, la notion d'élément de calcul est associée au multiprocesseur. Chaque RL correspond donc à un bloc et les différents voisins sont évalués en parallèle par les threads. La stratégie de type $pop_{gpu}sol^{par}_{bl}elt_{th}$ stocke les données des solutions en mémoire partagée alors que celle de type $pop_{gpu}sol^{gl}_{bl}elt_{th}$ les conserve en mémoire globale. Cette approche, qui a fait l'objet d'une publication dans les actes de la conférence internationale *Learning and Intelligent OptimizatioN (LION 6)* [DDK12b], démontre ainsi la généricité du cadre proposé.

Pour la seconde approche $RLI - RKBI$, une nouvelle stratégie de parallélisation spécialement conçue pour tirer avantage des capacités du GPU est proposée. Elle est hybride car elle est composée de deux noyaux de types différents : $pop_{gpu}sol_{blS}elt_{th}$ et $pop_{gpu}sol_{bl}elt_{th}$. Le premier noyau est consacré à l'évaluation des voisins en parallèle par un grand nombre de threads. Il permet ainsi d'associer une solution à plusieurs blocs. Le second noyau est dédié au remplacement de la solution courante, assignée à un bloc, par son meilleur voisin. Les deux approches sont comparées et leur influence sur la qualité des solutions obtenues ainsi que sur la réduction du temps de calcul est étudiée. L'ensemble de ces travaux a fait l'objet d'une publication dans les actes de la conférence internationale *18th International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'12)* [DDK12c].

5.1 Recherche Locale

Une Recherche Locale (RL) vise à améliorer itérativement une solution S. Elle débute avec une solution initiale générée aléatoirement ou par une heuristique de construction. Le voisinage $V(S)$ de S est ensuite calculé en appliquant à S des transformations locales, comme l'inversion qui échange deux éléments voisins dans la solution. Plusieurs exemples sont fournis dans la Figure 5.1 [Rei94, Dra10]. Les voisins générés sont alors évalués et la solution courante S est remplacée par un voisin de coût plus petit. La recherche se termine lorsque un optimum local est atteint, c'est-à-dire, lorsque S ne possède plus aucun voisin améliorant.

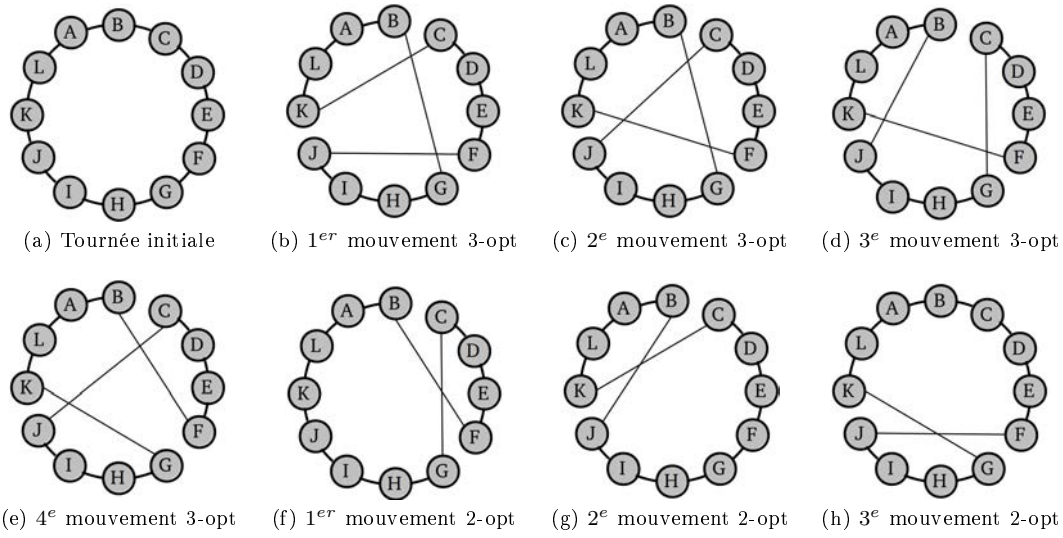
Transformation locales	Avant	Après
Inversion (échange de deux éléments voisins)	0 1 ② ③ 4 5 6 7 8 9	0 1 ③ ② 4 5 6 7 8 9
Transposition (échange d'éléments)	0 1 ② 3 4 5 6 ⑦ 8 9	0 1 ⑦ 3 4 5 6 ② 8 9
Déplacement d'un élément (insertion d'un nœud)	0 1 2 3 4 5 6 ⑦ 8 9	0 1 ⑦ 2 3 4 5 6 8 9
Inversion d'une sous-chaine	0 1 2 ③ ④ ⑤ ⑥ 7 8 9	0 1 2 ⑥ ⑤ ④ ③ 7 8 9
Déplacement d'une sous-chaîne (insertion d'arêtes)	0 1 2 3 ④ ⑤ ⑥ 7 8 9	0 ④ ⑤ ⑥ 1 2 3 7 8 9

Figure 5.1 – Exemple de transformations locales.

Les algorithmes de RL les plus connus pour le problème du Voyageur de Commerce (VC) sont basés sur des échanges k-opt qui suppriment k arêtes de la solution courante et reconnectent les tournées partielles avec k autres arêtes, de toutes les manières possibles. L'Algorithme 5.1 décrit la procédure spécifique du 3-opt [Lin65]. Elle débute avec une solution initiale S à laquelle trois arêtes $(a, a+1)$, $(b, b+1)$ et $(c, c+1)$ ont été supprimées. Pour un problème de n villes, le nombre de combinaisons (a, b, c) possibles est de $\binom{n}{3}$ [GP02]. Le voisinage $V(S)$ de cette solution est ensuite généré en reconnectent les tournées partielles avec trois autres arêtes. Pour chaque combinaison de trois arêtes supprimées, il y a sept façons de reconnecter les tournées partielles comme illustré dans la Figure 5.2. Quatre mouvements sont de réels 3-opt alors que les trois autres sont des mouvements 2-opt. Les sept voisins générés sont ensuite évalués et le meilleur remplace la solution courante S. La recherche se poursuit jusqu'à ce que S devienne un optimum local.

Tant que la solution S n'est pas un optimum local **Faire**
  Pour chaque combinaison $a, b, c \in [0; n]$ **Faire**
    Supprimer les arêtes $(a, a+1)$, $(b, b+1)$ et $(c, c+1)$
    Produire les voisins $V(S)$ de S en reconnectant les tournées partielles
    Évaluer les voisins $V(S)$
  Remplacer S par le voisin choisi par la règle de pivotement
Retourner la meilleure solution S

Algorithme 5.1 – Recherche Locale 3-opt



(a) Tournée initiale (b) 1^{er} mouvement 3-opt (c) 2^{e} mouvement 3-opt (d) 3^{e} mouvement 3-opt

(e) 4^{e} mouvement 3-opt (f) 1^{er} mouvement 2-opt (g) 2^{e} mouvement 2-opt (h) 3^{e} mouvement 2-opt

Figure 5.2 – Mouvements 3-opt : 7 façons de reconnecter les tournées partielles quand 3 arêtes sont supprimées.

Parmi les procédures les plus populaires, il est également possible de citer les algorithmes 2-opt [Cro58] et de Lin-Kernighan [LK73]. Trois mouvements 2-opt sont présentés dans la Figure 5.2. Ils correspondent à transformation locale nommée "inversion d'une sous-chaine" [Dra10]. Un mouvement 2-opt consiste à éliminer deux arêtes de la solution et à reconnecter les deux chemins résultants d'une manière différente. Contrairement au mouvement 3-opt où les chemins

peuvent être reliés de sept façons différentes, il n'y a qu'une manière possible de reconnecter les deux sous-tournées générées par le 2-opt. L'algorithme de Lin-Kernighan est une généralisation des algorithmes k-opt. Cependant, le nombre d'éléments échangés k n'est pas fixé et dépend de la solution courante.

Un des éléments clé de la RL est la règle de pivotement qui dicte le choix de la solution voisine qui remplacera la solution courante, ce choix pouvant considérablement affecter la complexité de la RL [Yan90]. Les plus utilisées sont les règles *best-improvement* et *first-improvement* [HS04]. Dans le premier cas, tous les voisins de la solution courante sont évalués et celui qui produit la plus grande amélioration est sélectionné. Formellement, étant donnée une solution S, $f^* = min\{f(S') \mid S' \in V(S)\}$ est la meilleure valeur de fonction d'évaluation f dans le voisinage $V(S)$ de S. L'ensemble des voisins améliorants de valeur maximale de S correspond alors à $I^*(S) = \{S' \in V(S) \mid f(S') = f^*\}$. Dans le second cas, la règle first-improvement accepte le premier voisin améliorant et les autres ne sont pas pris en compte. Formellement, elle évalue les solutions voisines $S' \in V(S)$ de S dans un ordre particulier fixe et le premier voisin S' pour lequel $f(S') < f(S)$ est choisi. À partir de la même solution initiale, plusieurs applications de la règle first-improvement aboutissent au même optimum local. Afin de diversifier le processus de recherche, il est possible d'utiliser des ordres aléatoires qui, à partir d'une solution initiale, permettent d'atteindre des optima locaux différents. L'ordre dans lequel les voisins sont évalués peut donc avoir une influence significative sur l'efficacité de cette méthode. Selon la règle de pivotement choisie, un compromis entre le nombre d'étapes de recherche et le temps de chaque étape est nécessaire. Dans la règle first-improvement, les étapes peuvent être calculées plus efficacement que dans la règle best-improvement mais l'amélioration obtenue à chaque étape est plus petite. Un plus grand nombre d'étapes est donc effectué.

D'autres méthodes peuvent être utilisées telles que les règles *random-improvement* et *least-improvement*. La première choisit un voisin améliorant aléatoirement dans l'ensemble $I(S) = \{S' \in V(S) \mid f(S') < f(S)\}$. Cette stratégie peut être implémentée comme une règle first-improvement où un nouvel ordre d'évaluation aléatoire est utilisé à chaque itération. La seconde règle choisit le voisin dans $I(S)$ qui produit l'amélioration la plus petite. Contrairement à la stratégie first-improvement qui choisit le premier voisin améliorant, les règles best-improvement, least-improvement et random-improvement doivent évaluer l'ensemble du voisinage avant de choisir la prochaine solution. Anderson [And96] a défini un paramètre k associé au nombre de mouvements améliorants trouvés avant de choisir le meilleur. Quand $k = 1$, l'algorithme utilise une stratégie first-improvement et plus k augmente, plus l'exploration du voisinage est approfondie. Il en déduit que la règle first-improvement est généralement le meilleur choix pour le problème du VC.

Dans le but d'accélérer l'exécution des algorithmes de RL pour le problème du VC, différents mécanismes sont généralement utilisés pour réduire le voisinage de la solution courante : listes de candidats, recherche de voisinage à rayon fixe et *don't look bits*. Premièrement, les listes de candidats contiennent les cl villes les plus proches ordonnées par distance croissante pour chaque ville i. Lorsqu'elles sont utilisées, une ville i ne peut être reconnectée par une arête à une ville j que si cette dernière fait partie de sa liste de candidats. L'ensemble des villes n'est donc plus considéré pour relier les tournées partielles mais seulement une partie. L'utilisation des listes de candidats au sein de la règle first-improvement biaise l'évaluation du voisinage. En examinant les mouvement les plus prometteurs en premier, la qualité des optima locaux

trouvés est alors considérablement améliorée [HS04].

Deuxièmement, la recherche de voisinage à rayon fixe (*fixed-radius neighbor search*) permet de reconnecter les chemins avec de nouvelles arêtes dont la somme des coûts est potentiellement inférieure à la somme des coûts des arêtes supprimées. Par exemple, si v_1, v_2, v_3 et v_4 sont les villes impliquées dans un mouvement 2-opt, $d(v_1, v_2) > d(v_1, v_3)$ ou $d(v_3, v_4) > d(v_4, v_2)$ ou les deux. Comme le montre la Figure 5.3, si (A, B) est le premier arc à supprimer (I), la recherche de voisinage à rayon fixe centrée en A avec un rayon $d(A, B)$ est appliquée (II). Pour chaque ville C trouvée durant la recherche (III), son voisin D approprié est considéré (IV). Si la longueur de la nouvelle tournée est plus courte que l'ancienne, le mouvement 2-opt est effectué : l'arc (C, D) est supprimé et les arcs (A, C) et (B, D) sont ajoutés. Si le voisin D est mal choisi (V), la tournée est brisée en deux sous-tournées disjointes (VI).

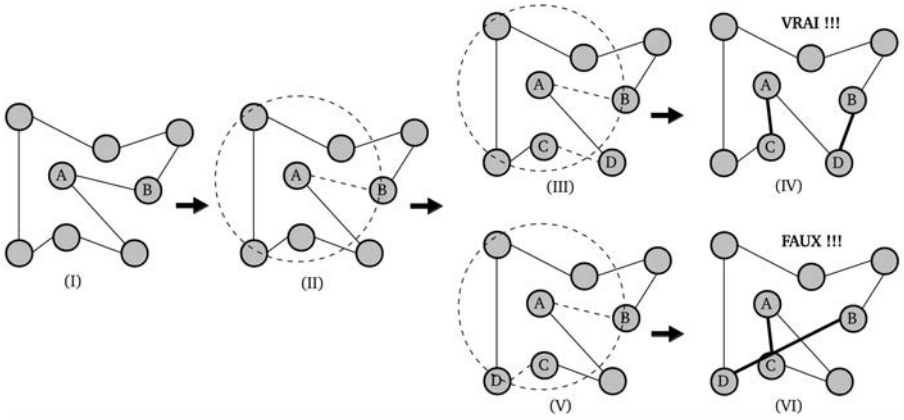


Figure 5.3 – Recherche de voisinage à rayon fixe pour le 2-opt.

Pour le mouvement 3-opt, si v_1, v_2, v_3, v_4, v_5 et v_6 sont les villes impliquées, alors $d(v_1, v_2) > d(v_2, v_3)$ et $d(v_1, v_2) + d(v_2, v_3) > d(v_2, v_3) + d(v_4, v_5)$. Comme le montre la Figure 5.4, si (A, B) est le premier arc à supprimer (I), la recherche de voisinage à rayon fixe centrée en A avec un rayon $d(A, B)$ est appliquée (II). Pour chaque ville C trouvée durant la recherche, son voisin D approprié est considéré (III). Les arcs (A, B) et (C, D) sont supprimés et l'arc (A, C) est ajouté. La recherche de voisinage à rayon fixe centrée en C avec un rayon $d(A, B) + d(C, D) - d(A, C)$ est ensuite appliquée (IV). Pour chaque ville E trouvée durant la recherche, son voisin F approprié est considéré. Si la longueur de la nouvelle tournée est plus courte que l'ancienne, le mouvement 3-opt est effectué : l'arc (E, F) est supprimé et les arcs (B, F) et (D, E) sont ajoutés (V).

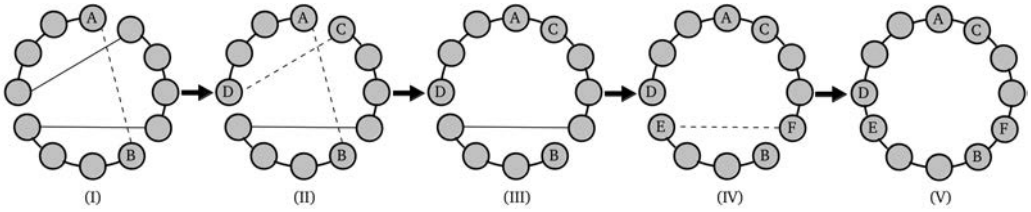


Figure 5.4 – Recherche de voisinage à rayon fixe pour le 3-opt.

Troisièmement, un bit (*don't look bit*) est associé à chaque ville dans le but d'éloigner la recherche des arêtes qui ont récemment mené à des mouvements non améliorants. Ces bits permettent de stocker l'état d'une ville et sont tous éteints (initialisés à 0) au début de l'algorithme. Lorsque aucun mouvement n'est possible pour une ville de départ, son bit est allumé et son état passe à 1. Cette ville n'est alors plus considérée comme ville de départ durant les prochaines itérations. Cependant, lorsque elle est à nouveau impliquée dans un mouvement où une de ses arêtes incidentes est modifiée, son état est remis à 0. Ce mécanisme réduit significativement la complexité en temps des recherches first-improvement puisque après quelques évaluations du voisinage, la plupart des états sont passés à 1 [HS04]. Une description complète des mécanismes pouvant être intégrés dans les méthodes de RL pour le problème du VC est trouvée dans Bentley [Ben92] et dans Johnson et McGeoch [JM97].

Un algorithme de RL est bloqué dans un optimum local lorsque plus aucun mouvement dans le voisinage de la solution courante n'est possible. Ce problème peut être en partie contré en l'intégrant dans une métaheuristique telle que la Recherche Locale Itérée (RLI). La RL 3-opt est utilisée comme cadre de référence afin de comparer les résultats obtenus avec ceux décrits dans les travaux de la littérature sur la RLI [SH01, LMS10]. De plus, la RL 3-opt offre un bon compromis entre le temps de résolution et la qualité de solutions. En effet, selon Lourenço *et al.* [LMS10], l'algorithme de Lin-Kernighan itéré donne généralement de meilleures solutions que le 3-opt itéré, ce dernier donnant de meilleures solutions que le 2-opt itéré. L'algorithme de Lin-Kernighan est un peu plus lent que le 3-opt, qui est un peu plus lent que le 2-opt mais l'amélioration des solutions compense le temps de calcul supplémentaire. Cependant, certaines RL comme le 4-opt impliquent des temps de calcul trop grands par rapport à l'amélioration des solutions. Il est donc préférable d'appliquer plus fréquemment une RL rapide mais moins efficace qu'une RL lente et puissante. La RLI est présentée dans la section suivante.

5.2 Recherche Locale Itérée

La métaheuristique de Recherche Locale Itérée (RLI) permet à la RL d'obtenir des solutions de meilleure qualité. Elle est retrouvée sous de nombreux noms dans la littérature : descente itérée (*iterated descent*), chaînes de Markov "à grand pas" (*large-step Markov chains*), Lin-Kernighan itéré (*iterated lin-kernighan*) et optimisation locale chaînée (*chained local optimization*). La RLI est divisée en quatre étapes principales mises en évidence dans l'Algorithme 5.2 : génération, RL, perturbation et acceptation. Une description complète de cette métaheuristique peut être trouvée dans Lourenço *et al.* [LMS10], Stützle et Hoos [SH01] et Hoos et Stützle [HS04].

La première étape génère une solution initiale S avec une heuristique de construction ou aléatoirement. La qualité de cette solution initiale est importante si des solutions de haute qualité veulent être obtenues rapidement. L'utilisation d'une heuristique de construction, par rapport à une génération aléatoire, permet donc généralement d'obtenir des solutions de meilleure qualité. De plus, une RL débutant avec une solution générée par une heuristique de construction nécessite moins d'étapes d'amélioration et donc moins de temps calcul.

La seconde étape applique la procédure de RL à S pour l'amener à un optimum local. Bien souvent, plus la RL est performante et plus la RLI correspondante est efficace. La qualité

Générer une solution S
Appliquer une procédure de RL sur S
Évaluer la longueur L de la solution S
Tant que le critère de fin n'est pas atteint **Faire**
  Transformer S en S' par un mouvement perturbant
  Appliquer une procédure de RL sur S'
  Évaluer la longueur L de la solution S'
  Si $L' < L$ **Alors**
    Remplacer S par S' // *critère d'acceptation*
Retourner la meilleure solution S

Algorithme 5.2 – Recherche Locale Itérée

des solutions obtenues par la RLI est donc fortement liée à la RL mais également aux autres composants de la RLI. En effet, le meilleur choix de perturbation dépend de la RL alors que le meilleur choix de critère d'acceptation dépend de la RL et de la perturbation.

La troisième étape est une perturbation qui transforme S en S' dans le but de s'extraire de l'optimum local obtenu avec la RL. Le nombre d'arêtes modifiées dans la solution correspond à la force de la perturbation. Si la perturbation est trop faible, la RL retombe dans l'optimum local qu'elle vient juste de visiter. La RL ne doit donc pas être capable de déconstruire la perturbation. Si la perturbation est trop forte, la RLI a le comportement d'une RL à relances aléatoires multiples et la probabilité de trouver les meilleures solutions est faible. Un mouvement aléatoire dans le voisinage de grand ordre, autre que celui de la RL, peut être efficace. La perturbation la plus utilisée pour le problème du VC est le mouvement *double-bridge* présenté dans la Figure 5.5. Il supprime quatre arêtes et reconnecte les quatre sous-tournées résultantes $s_1 - s_2 - s_3 - s_4$ de cette façon : $s_4 - s_3 - s_2 - s_1$.

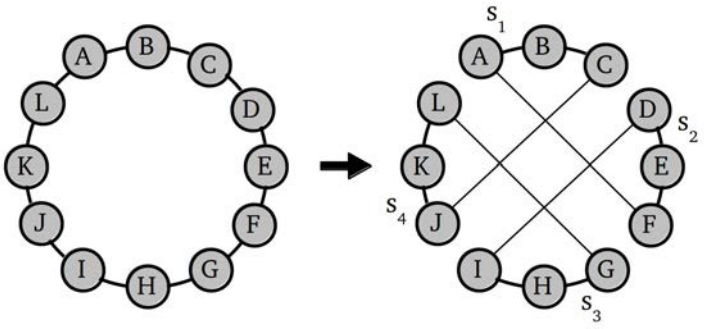


Figure 5.5 – Mouvement double-bridge.

Enfin, un critère d'acceptation est évalué pour choisir quelle solution (S ou S') continuera la recherche. Il permet de contrôler l'équilibre entre l'intensification et la diversification de la recherche. Les trois critères d'acceptation principaux sont : Markovien, Marche Aléatoire et Relance. Le premier favorise l'intensification de la recherche en acceptant seulement les meilleures solutions. Il est défini comme suit :

$$Markovien(S, S') = \begin{cases} S' \ si \ f(S') < f(S) \\ S \ sinon \end{cases} \quad (5.1)$$

A l'opposé, le critère d'acceptation Marche Aléatoire (MR) favorise la diversification en acceptant toujours la nouvelle solution, peu importe sa qualité. Il est défini par l'équation suivante :

$$MR(S, S') = S' \tag{5.2}$$

Enfin, le dernier critère d'acceptation permet de relancer l'algorithme lorsque la meilleure solution n'a pas été améliorée durant i_r itérations. Soit i l'itération courante et i_m la dernière itération où la meilleure solution a été remplacée :

$$Relance(S, S') = \begin{cases} S' \ si \ f(S') < f(S) \\ S_n \ si \ f(S') \geq f(S) \ et \ i - i_m > i_r \\ S \ sinon \end{cases} \tag{5.3}$$

où S_n est générée aléatoirement ou avec une heuristique de construction. Les trois dernières étapes RL, perturbation et acceptation sont répétées jusqu'à ce qu'un critère de fin soit atteint, comme une limite maximale de temps ou un nombre d'itérations par exemple.

La métaheuristique RLI est considérée comme une des méthodes approchées les plus performantes pour le problème du VC [HS04]. En effet, les travaux de Stützle et Hoos [SH01] et de Lourenço *et al.* [LMS10] montrent sa compétitivité en résolvant différents problèmes du VC dont la taille varie entre 100 et 5915 villes. Cependant, face à de larges problèmes d'optimisation difficile, un temps de calcul important et une grande quantité d'espace mémoire peut être nécessaire pour être efficace dans l'exploration de l'espace de recherche. Une façon d'accélérer cette exploration est d'utiliser le calcul parallèle. Par conséquent, la phase de RL doit être parallélisée efficacement pour obtenir une méthode de résolution globalement performante. Les sections suivantes présentes les stratégies de parallélisation GPU proposées.

5.3 Stratégies de parallélisation GPU

Plusieurs stratégies de parallélisation pour la RLI qui résolvent efficacement le problème du VC dans un environnement GPU sont proposées. Elles reposent sur de multiples recherches qui permettent de choisir le nombre de solutions à gérer. Toutefois, seule la phase de RL est parallélisée sur le GPU au lieu des recherches entières. Comme pour la parallélisation de l'OCF, ces stratégies diffèrent par leur définition d'élément de calcul et par leur utilisation des mémoires du GPU. Cependant, elles se distinguent également par la règle de pivotement choisie pour sélectionner un voisin améliorant.

La Section 5.3.1 est consacrée à l'approche $RLI-FI$ qui utilise la règle de pivotement first-improvement et les mécanismes permettant de réduire la taille du voisinage. La méthodologie de parallélisation employée pour les algorithmes OCF est transposée à cette approche afin de montrer la généricité du cadre proposé. La première stratégie proposée associe donc chaque solution à un thread et la seconde, à un bloc. Dans cette dernière, les données des solutions peuvent être stockées dans la mémoire partagée ou globale.

La Section 5.3.2 est consacrée à l'approche $RLI-RKBI$. Une stratégie de parallélisation est spécialement conçue pour tirer avantage des capacités du GPU. Elle est basée sur une

évaluation synchrone d'un nombre fixe de voisins. Cette stratégie hybride est composée de deux noyaux de types différents. Le premier évalue les voisins en parallèle par un grand nombre de threads et associe une solution à plusieurs blocs. Le second est dédié au remplacement de la solution courante par son meilleur voisin.

5.3.1 Stratégies de parallélisation pour l'approche $RLI - FI$

La RLI offre un potentiel de parallélisation intéressant quand elle est étendue à une approche à base de population. Différentes solutions sont alors améliorées par le même algorithme exécuté sur plusieurs éléments de calculs. La partie la plus consommatrice en temps de la RLI est à l'application de la RL. Puisque chaque RL peut améliorer une solution indépendamment des autres, cette phase est déplacée du CPU au GPU. La plupart des problèmes rencontrés durant le processus de parallélisation de la RLI sont reliés à la gestion des mémoires, comme pour les algorithmes OCF. La méthodologie de parallélisation employée pour ces algorithmes est donc transposée à l'approche $RLI - FI$.

Les transferts de données entre le CPU et le GPU ainsi que les accès à la mémoire requièrent un temps considérable qui peut souvent être réduit en stockant les structures de données reliées dans la mémoire partagée. Cependant, dans le cas de la RLI appliquée au problème du VC, les données nécessaires à toutes les solutions de la population sont la matrice des distances et les listes de candidats. Elles sont trop importantes en taille (de $O(n \cdot cl)$ à $O(n^2)$) pour être contenues dans la mémoire partagée pour les plus gros problèmes. Elles sont donc conservées dans la mémoire globale. Toutefois, comme elles ne sont pas modifiées durant la phase de RL, il est possible de bénéficier du cache de texture pour réduire les temps d'accès. Ce modèle de parallélisation général de l'approche $RLI - FI$ est présenté dans la Figure 5.6.

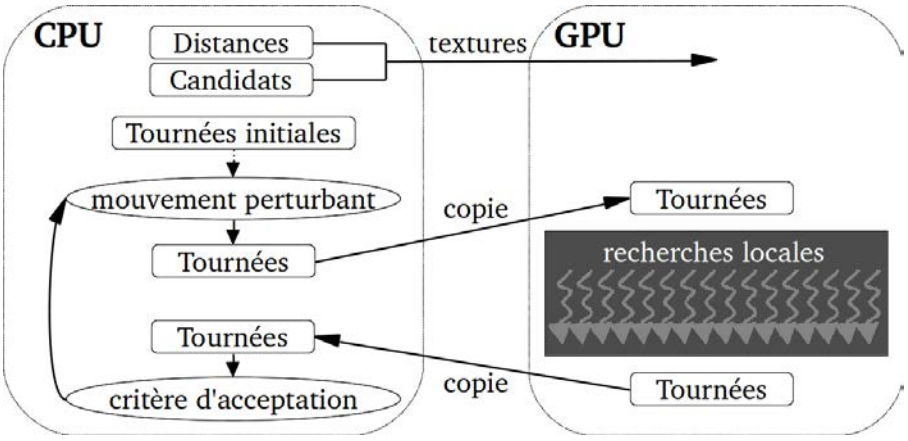


Figure 5.6 – Stratégie de parallélisation générale de l'approche $RLI - FI$.

Le fonctionnement de cette approche est décrit dans l'Algorithme 5.3. Tout d'abord, les solutions initiales S sont générées. Le CPU initialise ensuite les différentes structures de données, alloue l'espace mémoire nécessaire sur le GPU et y copie les données. Un noyau de RL est ensuite exécuté sur les solutions S. Il récupère l'identifiant de la solution et lui applique une RL. Lorsque toutes les solutions ont été améliorées sur le GPU, elles sont copiées sur le

CPU et leur longueur est calculée. Ensuite, tant que le critère de fin n'est pas atteint, les étapes suivantes sont itérées. Les solutions S sont transformées en S' par un mouvement perturbant sur le CPU et le noyau de RL est de nouveau appliqué sur les solutions S'. Enfin, le CPU les récupère et calcule leur longueur afin d'appliquer le critère d'acceptation.

```
Générer S, l'ensemble des solutions S_x avec x ∈ {0, ..., nb_sol}
Copier la matrice de distance et les listes de candidats dans la mémoire texture du GPU
Copier S dans la mémoire globale du GPU
Initialiser le nombre de blocs nbbl et de threads par bloc nbth
Noyau avec nbbl blocs de nbth threads
    Récupérer l'identifiant id de la solution
    Appliquer la procédure de RL sur S_id
Récupérer S du GPU sur le CPU
Pour x = 0 à nb_sol Faire
    Calculer la longueur L_x de la solution S_x
Tant que le critère de fin n'est pas atteint Faire
    Pour x = 0 à nb_sol Faire
        Transformer la solution S_x en S'_x par un mouvement perturbant
    Copier S' dans la mémoire globale du GPU
    Noyau avec nbbl blocs de nbth threads
        Récupérer l'identifiant id de la solution
        Appliquer la procédure de RL sur S'_id
    Récupérer S' du GPU sur le CPU
    Pour x = 0 à nb_sol Faire
        Calculer la longueur L'_x de la solution S'_x
    Si L'_x < L_x Alors // critère d'acceptation
        S_x = S'_x et L_x = L'_x
Retourner la meilleure solution
```

Algorithme 5.3 – Pseudo-code de l'approche générale $RLI - FI$.

À partir de cette structure générale, deux stratégies spécifiques de parallélisation de la RL sont proposées. La première est nommée $RLI - FI_{thread}$ et correspond à la stratégie $FOURMI_{thread}$ de l'OCF. Elle est basée sur l'association de l'élément de calcul à un processeur. Comme le montre la Figure 5.7, chaque RL est donc appliquée à une solution par un thread CUDA pour l'améliorer de façon SIMD. Dans le noyau de RL décrit dans l'Algorithme 5.3, l'identifiant id de la solution correspond donc à l'identifiant du thread. Cette stratégie permet d'effectuer un grand nombre de RL sur chaque multiprocesseur mais limite l'utilisation de la mémoire rapide du GPU.

Cette stratégie est de type $pop_{gpu}sol^{gl}_{th}elt^{tex,reg}$ selon la taxonomie proposée dans le Chapitre 3. La population est déportée sur le GPU (pop_{gpu}) et les solutions sont associées aux threads (sol_{th}). Comme la stratégie $FOURMI_{thread}$, les données du problème sont les paramètres de la RLI, la matrice de distance et les listes de candidats. Les paramètres sont stockés dans les registres (elt^{reg}) et les deux autres structures sont stockées dans la mémoire texture (elt^{tex}). Chaque RL a besoin de ses propres structures de données, principalement les tableaux contenant les tournées et les *don't look bits* (de taille $O(n)$), pour explorer efficacement le voisinage. Les données propres aux solutions sont donc stockées dans la mémoire globale du GPU (sol^{gl}) car

peu de RL pourraient être effectuées sur chaque multiprocesseur si la mémoire partagée était utilisée.

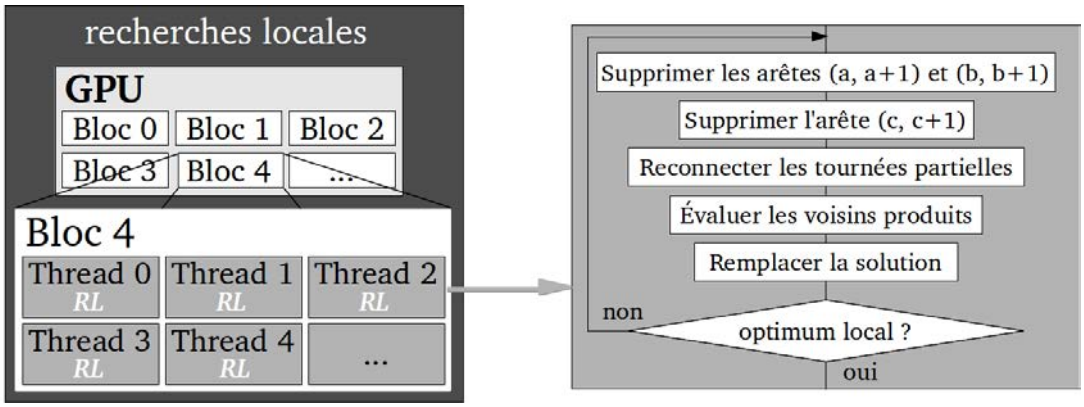


Figure 5.7 – Stratégie de parallélisation $RLI - FI_{thread}$.

La seconde stratégie est nommé $RLI - FI_{bloc}$ et correspond à la stratégie $FOURMI_{bloc}$ de l'OCF. Elle est basée sur l'association de l'élément de calcul à un multiprocesseur. Cette stratégie, illustrée dans la Figure 5.8, associe chaque solution améliorée par une RL à un bloc CUDA. Dans le noyau de RL décrit dans l'Algorithme 5.3, l'identifiant *id* de la solution correspond donc à l'identifiant du bloc. Un thread d'un bloc donné est en charge d'appliquer la RL à une solution. Le parallélisme est alors préservé pour la phase de RL. Cependant, un second degré de parallélisme est exploité dans lequel la génération et l'évaluation des différents voisins sont partagées entre les threads du bloc. Plus précisément, les deux premières arêtes de la solution à améliorer sont tout d'abord supprimées. Chaque thread est ensuite affecté à la suppression d'un troisième arc différent. Il génère alors les sept voisins possibles en reconnectant les sous-tournées et les évalue. S'il a généré au moins un voisin améliorant, il stocke le premier obtenu. Une réduction parallèle est ensuite effectuée pour trouver le voisin qui remplacera la solution courante. Pour ce faire, les voisins stockés sont parcourus dans l'ordre croissant des indices de threads et le premier trouvé est sélectionné.

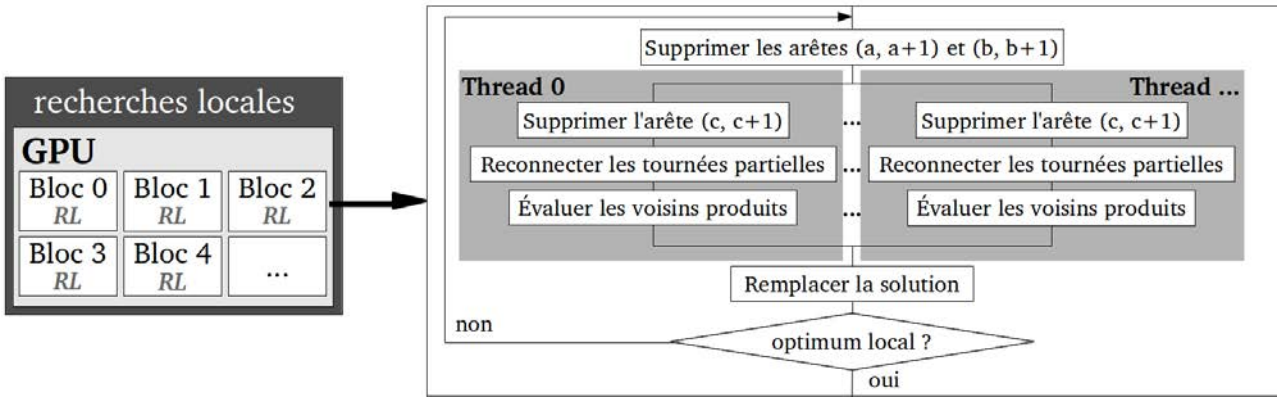


Figure 5.8 – Stratégie de parallélisation $RLI - FI_{bloc}$.

Cette stratégie est donc de type $pop_{gpu}sol_{bl}elt_{th}^{tex,reg}$. La population est déportée sur le GPU (pop_{gpu}), les solutions sont associées aux blocs (sol_{bl}) et les voisins sont assignées aux threads (elt_{th}). Comme pour la stratégie $RLI - FI_{thread}$, les données du problème ($elt^{tex,reg}$) sont placées dans la mémoire texture et dans les registres. Puisqu'une seule solution est assignée au bloc, il devient possible de stocker les structures de données nécessaires pour améliorer la solution dans la mémoire partagée. Deux variantes de la stratégie $RLI - FI_{bloc}$ sont alors distinguées : $RLI - FI_{bloc}^{globale}$ et $RLI - FI_{bloc}^{partagée}$. Dans le premier cas, la stratégie est de type $pop_{gpu}sol_{bl}^{gl}elt_{th}^{tex,reg}$ car les données des solutions sont conservées dans la mémoire globale (sol^{gl}). Dans le second cas, la stratégie est de type $pop_{gpu}sol_{bl}^{par}elt_{th}^{tex,reg}$ car la mémoire partagée est utilisée (sol^{par}).

Actuellement, les méthodes de RL les plus efficaces pour résoudre le problème du VC sont basées sur la règle de pivotement first-improvement et les mécanismes réduisant les temps d'exécution. Ils sont manifestement conçus et optimisés pour les architectures traditionnelles qui possèdent un seul processeur. Alors qu'il peut être possible d'obtenir de bonnes performances sur de petits systèmes multiprocesseur/multicœur avec peu de modifications des algorithmes existants, le processus de parallélisation se complexifie pour une architecture massivement parallèle et synchrone comme le GPU. D'une part, les implémentations séquentielles first-improvement sont basées sur une réduction variable du nombre de voisins calculés à chaque étape de la RL. Cependant, le GPU est mieux adapté aux applications qui comportent un grand nombre fixe de calculs à effectuer. En parallélisant la phase d'évaluation de la RL, un certain nombre de voisins sont donc générés inutilement. La stratégie GPU effectue alors un plus grand nombre de calculs que l'algorithme séquentiel.

D'autre part, les mécanismes d'accélération sont souvent basés sur des instructions conditionnelles qui induisent une divergence des threads au sein des warps et une sérialisation de l'algorithme. De plus, les listes de candidats impliquent de nombreux accès à la mémoire globale quand la mémoire partagée n'est pas utilisée. En effet, elles nécessitent de rechercher plusieurs fois à chaque itération l'indice d'une ville candidate dans la tournée ou bien, de créer une seconde structure de données pour conserver les indices des villes. Par ailleurs, les accès à la mémoire globale sont le plus souvent non coalescents car les villes candidates ne sont pas forcément adjacentes dans la tournée. Ces observations ont donc conduit à la proposition d'une nouvelle stratégie de parallélisation basée sur une évaluation synchrone d'un nombre fixe de voisins.

5.3.2 Stratégies de parallélisation pour l'approche $RLI - RKBI$

Dans une RL best-improvement basique, la totalité du voisinage doit être évalué, ce qui correspond à $7 \cdot \binom{n}{3}$ voisins comme précisé dans la Section 5.1. Les recherches sur le problème du VC montrent que ce schéma aboutit à des temps d'exécution prohibitifs et des optima locaux plus éloignés de ceux obtenus avec une règle first-improvement [And96]. Une règle de pivotement nommée *random-k-best-improvement* est donc proposée. Elle offre un compromis entre les règles first-improvement et best-improvement ainsi qu'un modèle mieux adapté aux GPU. De la même façon que la règle best-improvement, le nombre de voisins évalué dans la version séquentielle est fixe. De plus, une réduction du voisinage est effectuée comme dans la règle first-improvement : seuls k voisins sélectionnés aléatoirement sont évalués.

Le fonctionnement de la RL qui intègre cette règle de pivotement est décrit dans l'Algorithme 5.4. À chaque étape de l'algorithme, sept voisins sont générés et évalués. $\frac{k}{7}$ étapes sont donc effectuées au total. Dans chacune d'entre elles, les trois arêtes supprimées sont choisies aléatoirement. Les sous-tournées sont ensuite reconnectées de toutes les façons possibles. Celle qui produit la plus grande amélioration est sélectionnée pour modifier la solution. En assignant différentes valeurs à k, le comportement de la recherche et la quantité de travail effectuée par le GPU peut être personnalisée.

Tant que la solution S n'est pas un optimum local **Faire**
  Pour $v = 1$ à $k/7$ **Faire**
    Choisir aléatoirement une combinaison $(a, b, c) \in [0; n]$
    Supprimer les arêtes $(a, a+1)$, $(b, b+1)$ et $(c, c+1)$
    Produire les 7 voisins $V(S)$ de S en reconnectant les tournées partielles
    Évaluer les voisins $V(S)$
  Remplacer S par le meilleur voisin améliorant
Retourner la meilleure solution S

Algorithme 5.4 – Recherche Locale 3-opt avec règle de pivotement random-k-best-improvement.

Cette règle de pivotement conduit à la proposition d'une approche de parallélisation nommée $RLI - RKBI$. Elle repose sur le modèle général présenté dans la Figure 5.9. La matrice des distances est conservée dans la mémoire texture du GPU pour réduire les temps d'accès à la mémoire globale, comme pour l'approche $RLI - FI$. Cependant, la phase de RL est divisée en deux parties distinctes. La première génère et évalue les voisins en parallèle alors que la seconde remplace la solution courante par son meilleur voisin. Des allers-retours fréquents entre le CPU et le GPU sont effectués afin de synchroniser ces deux parties.

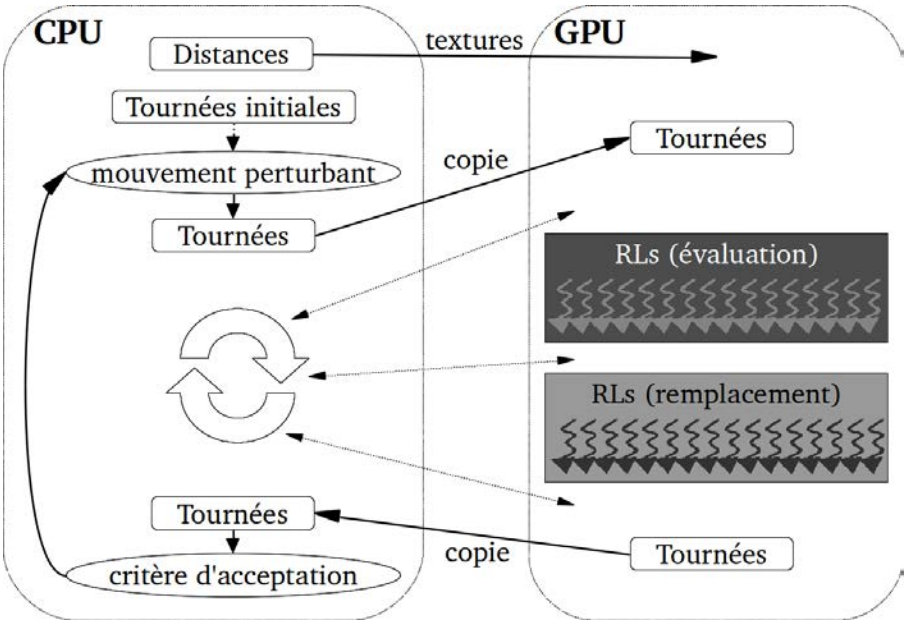


Figure 5.9 – Stratégie de parallélisation générale de l'approche $RLI - RKBI$.

À partir de cette structure générale, une stratégie spécifique de parallélisation de la RL est proposée. Elle est nommée $RLI - RKBI_{blocs}$ et présentée dans la Figure 5.10.

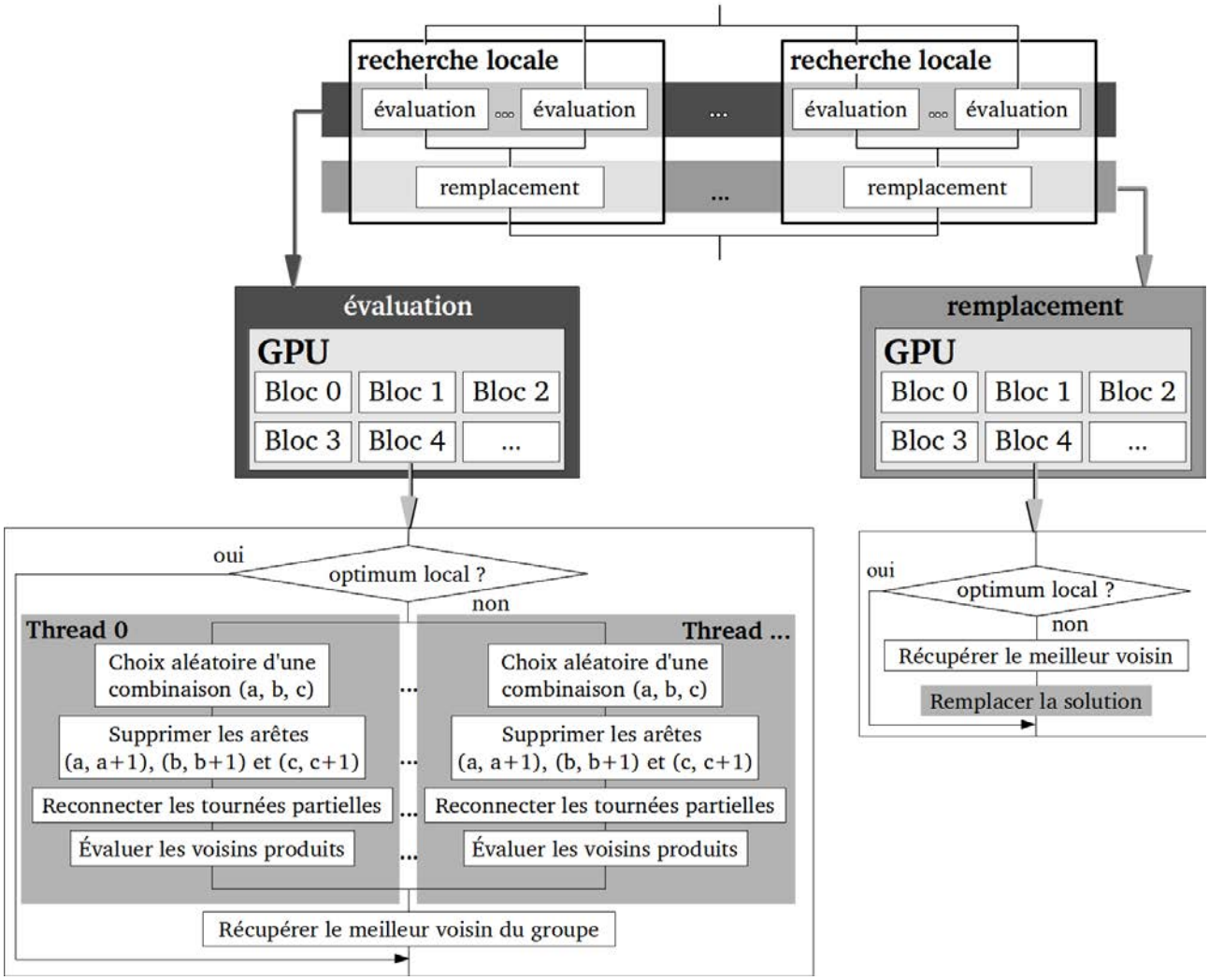


Figure 5.10 – Stratégie de parallélisation générale $RLI - RKBI_{blocs}$.

Cette stratégie décompose les calculs en deux noyaux qui correspondent aux deux parties distinctes de la RL : évaluation et remplacement. Selon la taxonomie proposée dans le Chapitre 3, elle est donc considérée comme stratégie hybride puisque les deux noyaux sont de type différent. Le premier noyau est dédié à la génération et l'évaluation du voisinage qui est la partie la plus coûteuse de l'algorithme. Il est de type $pop_{gpu}sol_{blS}^{gl,par}elt_{th}^{tex,reg}$. La population est déportée sur le GPU (pop_{gpu}) et chaque solution est associée à plusieurs blocs à la fois (sol_{blS}). Les voisins des solutions sont divisés en groupes affectés aux blocs. Chaque thread est alors en charge d'une partie des voisins de chaque groupe (elt_{th}). Comme pour l'approche $RLI-FI$, les données du problème sont placées dans la mémoire texture et dans les registres ($elt^{tex,reg}$). Les données propres aux solutions dont la taille varie selon le type de problème sont stockées dans la mémoire globale ($sol^{gl,par}$) du GPU. Certaines structures permettent de stocker les données qui caractérisent les meilleurs voisins de chaque thread. Elles ont une taille dépendante du

nombre de threads, quelque soit la taille du problème. Puisque le nombre de threads maximum par bloc est 1024, ces structures peuvent être contenues dans la mémoire partagée ($sol^{gl,par}$). Le second noyau est consacré au remplacement de la solution par le meilleur voisin. Il est de type $pop_{gpu} sol_{bl}^{gl} elt_{th}^{tex,reg}$. La population est déportée sur le GPU (pop_{gpu}), chaque solution est associée à un bloc (sol_{bl}) et les éléments aux threads (elt_{th}). Les données du problème sont placées dans la mémoire texture et dans les registres ($elt^{tex,reg}$). Les données des solutions sont dépendantes de la taille des problèmes donc elles sont stockées dans la mémoire globale (sol_{bl}^{gl}).

Le fonctionnement de cette stratégie est décrit dans l'Algorithme 5.5. Tout d'abord, les solutions sont copiées dans la mémoire globale du GPU. Une structure de données *tabopt*, dont la taille est égale au nombre de solutions, est également copiée dans la mémoire globale. Pour chaque solution, elle permet d'indiquer si la RL est terminée, c'est-à-dire, si un optimum local est obtenu. Le critère de fin de la RL est atteint lorsque toutes les solutions sont des optima locaux. Tant qu'elles ne le sont pas toutes, plusieurs étapes se succèdent. Premièrement, le noyau d'évaluation attribue chaque solution à plusieurs blocs. Chaque thread est affecté à la génération et l'évaluation d'une partie des voisins de son bloc associé. Pour générer ses voisins, un thread choisit alors aléatoirement les trois arêtes à supprimer de la solution puis évalue les sept reconnexions possibles. Il stocke les données permettant de caractériser le meilleur voisin dans la mémoire partagée : l'amélioration en longueur, le type de reconnexion parmi les sept possibles et l'indice du thread. Ce dernier permet de récupérer les indices des villes adjacentes aux arêtes supprimées et ajoutées qui sont stockées dans les registres de chaque thread. Une fois tous les voisins évalués, une réduction parallèle est effectuée dans la mémoire partagée pour trouver le meilleur voisin du bloc. Ce dernier est est ensuite stocké dans la mémoire globale de façon à être accessible au second noyau. Les blocs se synchronisent implicitement au retour sur le CPU et le noyau de remplacement est lancé. Chaque solution est alors associée à un bloc et récupère les meilleurs voisins calculés par le noyau précédent dans la mémoire globale. Elle trouve ensuite le meilleur et récupère les données le caractérisant. Par l'usage d'une structure de données intermédiaire et des différents threads composant le bloc, la solution courante est remplacée par son meilleur voisin. Les blocs se synchronisent de nouveau au retour sur le CPU et la structure *tabopt* est récupérée du GPU par le CPU. Cette structure est parcourue afin de savoir s'il y a encore des solutions à améliorer. Si c'est le cas, le premier noyau est de nouveau lancé. Sinon, les solutions S sont récupérées sur le CPU pour continuer l'application de la RLI.

```
Copier S dans la mémoire globale du GPU
Initialiser les cases du tableau tabopt à 0 et le copier dans la mémoire globale du GPU
fin ← 0
Tant que fin = 0 Faire
    Noyau d'évaluation
    Noyau de remplacement
    Récupération de tabopt sur le CPU
    Si tous chaque élément de tabopt = 1 Alors
        fin ← 1
Récupérer S du GPU sur le CPU
```

Algorithme 5.5 – Pseudo-code de la stratégie $RLI - RKBI_{blocs}$ (partie Recherche Locale).

Une étude expérimentale structurée et minutieuse est réalisée dans la prochaine section afin d'évaluer et de comparer ces différentes stratégies proposées, autant au niveau de la réduction du temps de calcul qu'à celui de la qualité des solutions obtenues.

5.4 Résultats expérimentaux

Les expérimentations ont été réalisées avec les configurations matérielles et logicielles décrites dans le Chapitre 4. Les accélérations ont également été calculées de la même façon.

Pour chaque approche $RLI-FI$ et $RLI-RKBI$, les stratégies GPU conçues dans les Sections 5.3.1 et 5.3.2 sont expérimentées et comparées sur plusieurs problèmes du VC dont les tailles varient entre 51 à 13509 villes. Conformément aux principes expérimentaux adoptés par Stützle et Hoos [SH01] et Lourenço *et al.* [LMS10], les solutions initiales de la RLI sont construites avec l'heuristique du plus proche voisin, les solutions sont améliorées avec une RL 3-opt et le mouvement *double-bridge* est utilisé comme procédure de perturbation.

Une étape préliminaire consiste à valider la version séquentielle de l'approche $RLI-FI$ par une étude comparative avec les travaux de Stützle et Hoos et Lourenço *et al.* Conformément aux principes expérimentaux adoptés par ces auteurs, la RL 3-opt utilise la règle de pivotement first-improvement, la procédure *don't look bits* et la recherche de voisinage à rayon fixe restreinte à des listes de candidats de taille 40. Les stratégies de parallélisation $RLI-FI_{thread}$ et $RLI-FI_{bloc}$ sont ensuite comparées entre elles autant au niveau de la qualité des solutions trouvées que de la réduction du temps de calcul. Une étude sur la variation du nombre de candidats est également effectuée afin de montrer l'influence de ce paramètre.

La seconde partie des expérimentations concerne l'approche $RLI-RKBI$. Tout d'abord, une étude préliminaire est effectuée afin de fixer les paramètres de l'algorithme pour chaque problème. La stratégie de parallélisation $RLI-RKBI_{blocs}$ est ensuite comparée aux stratégies de l'approche $RLI-FI$ selon différents critères liés à la qualité des solutions et aux accélérations obtenues.

5.4.1 Approche $RLI-FI$

La première partie des expérimentations consiste à comparer la version séquentielle de l'approche $RLI-FI$ avec les travaux de la littérature. Stützle et Hoos [SH01] exécutent la RLI séquentielle durant $it_{25000} = 25000$ itérations de façon à déterminer le temps d'exécution CPU t_{25000}. Des limites de temps d'exécution maximales t_{lim} sont ensuite fixées et l'algorithme est de nouveau exécuté afin d'évaluer la qualité des solutions trouvées pour les différents problèmes. Ces données sont fournies dans le Tableau 5.1. En vue de permettre une étude comparative des résultats obtenus par l'approche $RLI-FI$ avec les travaux de Stützle et Hoos, les mêmes principes expérimentaux doivent autant que possible être adoptés. Cependant, les temps d'exécution CPU peuvent varier selon le type de processeur utilisé. Un nombre d'itérations it^a_{lim} doit donc être fixé. À partir de it_{25000}, t_{25000} et t_{lim}, une estimation de it^a_{lim} est calculée et présentée dans le Tableau 5.1. Quant à eux, Lourenço *et al.* [LMS10] fixent le nombe d'itérations it^b_{lim} pour chaque problème. Ces valeurs sont également présentées dans le Tableau 5.1.

La prochaine section compare la qualité des solutions obtenues par la version séquentielle de l'approche $RLI - FI$ à celle fournie dans les travaux de ces auteurs.

Problème	Stützle et Hoos			Lourenço *et al.*
	t_{25000}	t_{lim}	it^a_{lim}	it^b_{lim}
kroA100	-	-	-	56186
d198	68,00	120	44118	36849
lin318	97,20	120	30865	25540
rat783	118,50	900	189874	21937
fl1577	124,80	2400	480770	22438
pcb3038	209,70	7200	858370	13323
rl5915	-	-	-	8820

Tableau 5.1 – Temps d'exécution t_{25000} pour $it_{25000} = 25000$ itérations et limites de temps t_{lim} fournies par Stützle et Hoos permettant d'estimer le nombre maximum moyen d'itérations it^a_{lim}. Nombre maximum d'itérations it^b_{lim} utilisé dans les travaux de Lourenço *et al.*

5.4.1.1 Qualité des solutions de l'approche séquentielle

Les principes expérimentaux décrits dans la section précédente sont appliqués à l'approche $RLI - FI$. Tout d'abord, la qualité des solutions obtenues est comparée avec les résultats fournis par Stützle et Hoos [SH01]. Le Tableau 5.2 présente le temps total d'exécution t_{total} et la qualité des solutions obtenues : la fréquence d'obtention de la solution optimale connue f_{opt} et le pourcentage moyen de déviation par rapport à l'optimum Δ_{moy}. Il présente également le temps moyen t_{moy} et le nombre d'itérations moyen it_{moy} pour trouver la meilleure solution d'un essai. De la même façon que it_{lim} a été calculé, it_{moy} est estimé selon les valeurs de it_{25000}, t_{25000} et t_{moy}. Les résultats sont calculés avec 100 essais pour les problèmes dont la taille est inférieure à 1000 villes, 25 essais pour les problèmes dont la taille est comprise entre 1000 et 2000 villes et 10 essais pour les instances les plus grandes.

Problème	f_{opt}		Δ_{moy}		t_{moy}		it_{moy}		t_{total}
	(Stü.)	(Del.)	(Stü.)	(Del.)	(Stü.)	(Del.)	(Stü.)	(Del.)	(Del.)
d198	1,00	1,00	0,000	0,000	1,10	0,15	404,41	220,33	28,96
lin318	0,65	1,00	0,100	0,000	13,90	3,64	3575,10	4148,01	26,98
rat783	0,71	0,79	0,029	0,024	238,80	72,14	50379,75	62498,92	218,73
fl1577	0,12	0,28	0,520	0,300	494,60	211,21	99078,53	141257,32	725,31
pcb3038	0,00	0,00	0,220	0,171	3687,60	2703,65	439628,04	643597,60	3624,26

Tableau 5.2 – Fréquence d'obtention de l'optimum f_{opt}, pourcentage moyen de déviation par rapport à l'optimum Δ_{moy}, temps moyen t_{moy} et nombre moyen d'itérations it_{moy} pour trouver la meilleure solution d'un essai et temps total d'exécution t_{total} obtenus avec l'implémentation proposée (Del.) en comparaison des résultats fournis par Stützle et Hoos (Stü.).

D'après les résutats du Tableau 5.2, il peut être observé que la qualité des solutions trouvées est égale ou meilleure que les résultats de référence sur tous les problèmes. L'optimum est trouvé plus fréquemment : $f_{opt} = \{1,00, \quad 0,79, \quad 0,28\}$ au lieu de $f_{opt} = \{0,65, \quad 0,71, \quad 0,12\}$;

mis à part pour les problèmes d198 et pcb3038 où il est obtenu aussi souvent. De plus, la moyenne des solutions est généralement plus proche de l'optimum puisque le pourcentage de déviation moyen obtenu est inférieur ou égal à celui obtenu par les auteurs. Cependant, le nombre moyen d'itérations pour trouver la meilleure solution d'un essai est bien souvent supérieur.

La qualité des solutions obtenues est ensuite comparée avec les résultats fournis par Lourenço *et al.* [LMS10]. Le Tableau 5.3 présente le pourcentage moyen de déviation par rapport à l'optimum Δ_{moy} et le pourcentage moyen de déviation Δ_{moy}^{Lou} par rapport aux résultats de Lourenço *et al.*. Ils ont été calculés avec 10 essais pour tous les problèmes. La qualité des solutions trouvées est égale ou légèrement meilleure à celle des travaux de référence à l'exception des problèmes de 1577 et 3038 villes où $\Delta_{moy}^{Lou} = +0,030$ et $\Delta_{moy}^{Lou} = +0,023$ respectivement. De façon générale, les implémentations séquentielles proposées fournissent des résultats comparables à ceux des travaux de Stützle et Hoos et Lourenço *et al.* Elles peuvent donc être utilisés comme cadre de référence pour les implémentations parallèles des stratégies proposées pour l'approche $RLI - FI$. La section suivante présente les résultats relatifs à la réduction des temps de calcul et compare la qualité des solutions trouvées par les stratégies de parallélisation par rapport à la version séquentielle.

Problème	Δ_{moy}		Δ_{moy}^{Lou}
	(Lou.)	(Del.)	(Del.)
kroA100	0,000	0,000	±0,000
d198	0,000	0,000	±0,000
lin318	0,120	0,027	−0,093
rat783	0,120	0,116	−0,004
fl1577	0,330	0,360	+0,030
pcb3038	0,470	0,493	+0,023
rl5915	0,660	0,490	−0,170

Tableau 5.3 – Nombre maximum d'itérations it_{lim} utilisées et pourcentage moyen de déviation par rapport à l'optimum Δ_{moy} obtenus avec l'implémentation proposée (Del.) en comparaison des résultats de Lourenço *et al.* (Lou.). Pourcentage moyen de déviation Δ_{moy}^{Lou} par rapport aux résultats de Lourenço *et al.*.

5.4.1.2 Qualité des solutions et accélérations des implémentations parallèles

Les stratégies de parallélisation proposées pour l'approche $RLI - FI$ reposent sur de multiples recherches. Une population de nb_{sol} solutions est donc utilisée où $nb_{sol} = 2^x$ avec $x \in \{0, 3, 6, 7, 8, 9, 10, 11\}$. Elle évolue durant un nombre total de it_{total} itérations. Dans le but d'effectuer globalement le même nombre de RL pour toutes les valeurs de nb_{sol}, la procédure de RLI est limitée à $it_{lim} = \frac{it_{total}}{nb_{sol}}$ itérations pour chaque solution. it_{total} est fixé à 1048576 pour chaque problème afin que le même nombre de RL soit effectué quelque soit la taille de l'instance, le nombre de blocs et le nombre de threads utilisés. Cette valeur correspond à la première puissance de 2 plus grande que le nombre d'itérations it_{lim}^{a} nécessaire pour résoudre le plus gros problème pcb3038 par l'algorithme séquentiel dans la Section 5.4.1.1 (Tableau 5.1). De plus, elle est facilement divisible par les nombres usuels utilisés dans les configurations nb_{sol}/it_{lim} et blocs/threads.

Les accélérations obtenues pour les stratégies de parallélisation de l'approche $RLI - FI$ ont été calculées à partir des temps séquentiels présentés dans le Tableau 5.4.

Problème	1	8	64	128	256	512	1024	2048
eil51	89,76	89,89	89,62	89,74	89,73	89,68	89,70	89,61
kroA100	270,99	272,96	267,00	268,70	268,17	269,94	267,57	268,07
d198	677,30	678,42	678,62	680,02	678,26	682,52	681,65	683,22
lin318	907,67	908,70	915,93	917,33	916,00	916,75	928,77	925,37
rat783	1186,16	1193,73	1195,57	1203,15	1236,36	1226,14	1249,01	1272,12
fl1577	1447,15	1443,31	1474,16	1475,61	1481,04	1527,62	1513,93	1614,39
d2103	1315,36	1368,98	1413,99	1431,27	1426,25	1469,53	1569,06	1557,82
pcb3038	3866,89	3906,92	4306,15	5288,64	4122,29	4166,17	4261,04	4431,25
fnl4461	12741,18	13124,47	14003,67	14149,98	14994,15	11816,76	15320,61	16585,62
rl5915	11880,08	13520,81	13731,77	13608,88	13842,93	15100,53	15532,11	19417,32
usa13509	46561,87	47947,28	50684,82	51754,19	55104,86	54329,17	55544,71	58699,04

Tableau 5.4 – Temps séquentiel de l'algorithme pour chaque problème et chaque valeur de nb_{sol}.

Le paramétrage des stratégies de parallélisation est ensuite effectué selon l'étude de l'influence du nombre de blocs et de threads par bloc présentée dans la Section 4.5.1.1 du Chapitre 4. Pour la stratégie $RLI - FI_{thread}$, le nombre de blocs et de threads est donc configuré de façon à maximiser le nombre de blocs sans dépasser le nombre total maximum de blocs pouvant être actifs simultanément sur le GPU. Les configurations utilisées sont présentées dans le Tableau 5.5. Pour la stratégie $RLI - FI_{bloc}$, le nombre de blocs utilisés est fixé à nb_{sol} et le nombre de threads par bloc à la taille des listes de candidats (ici, 40).

nb_{sol}	1	8	64	128	256	512	1024	2048
Nombre de blocs	1	8	64	64	64	64	64	64
Nombre de threads	1	1	1	2	4	8	16	32

Tableau 5.5 – Configurations du nombre de blocs et du nombre de threads par bloc pour la stratégie $RLI - FI_{thread}$.

Selon le nombre de solutions nb_{sol} et la stratégie de parallélisation utilisée, le temps de résolution de l'algorithme peut être pénalisé et donc, être beaucoup plus important que le temps séquentiel. De ce fait, la qualité des solutions et les accélérations sont calculées à partir d'un nombre d'essais variable. Les valeurs utilisées sont présentées dans le Tableau 5.6.

	Stratégie $RLI - FI_{thread}$			Stratégie $RLI - FI_{bloc}$		
	$nb_{sol} = 1$	$nb_{sol} = 8$	$nb_{sol} > 8$	$nb_{sol} = 1$	$nb_{sol} = 8$	$nb_{sol} > 8$
$<$ 1000 villes	3	5	20	3	20	20
$\geq$ 1000 villes	2	3	10	2	10	10
13509 villes	-	3	10	2	10	10

Tableau 5.6 – Nombre d'essais effectués selon la taille du problème, nb_{sol} et la stratégie de parallélisation.

Une fois les paramètres des algorithmes parallèles fixés, la réduction du temps de calcul peut être étudiée. La Figure 5.11 montre donc les accélérations obtenues pour chaque problème, chaque valeur nb_{sol} et chaque stratégie de parallélisation.

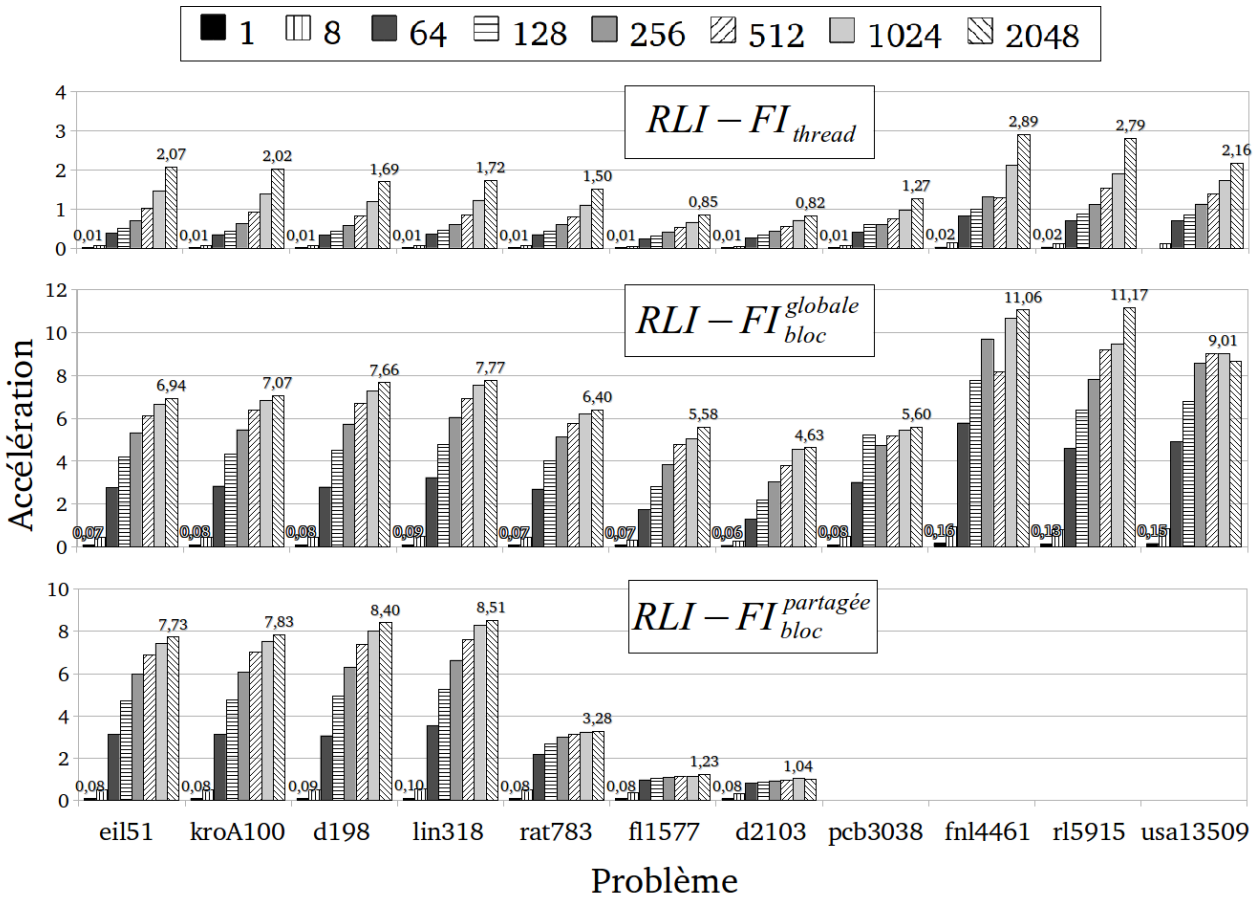


Figure 5.11 – Accélérations obtenues par les stratégies de parallélisation pour les différents problèmes et les différentes valeurs de nb_{sol}.

Tout d'abord, il peut être observé qu'en augmentant le nombre de solutions nb_{sol} et donc, le nombre total de threads, les accélérations augmentent pour toutes les stratégies dans presque tous les cas. En général, si le nombre de threads utilisé est trop petit, les ressources GPU ne sont pas bien exploitées et les latences mémoires ne sont pas dissimulées efficacement.

L'analyse de la Figure 5.11 montre également que les accélérations obtenues avec la stratégie $RLI - FI_{thread}$ sont toujours plus basses qu'avec la stratégie $RLI - FI_{bloc}$. Comme cette stratégie n'exécute pas assez de threads en parallèle pour dissimuler efficacement les latences mémoire, l'accélération maximale rapportée est de 2,89. En effet, des accélérations sont bien souvent obtenues seulement avec 1024 et 2048 threads. De plus, la divergence de code induite par le calcul des voisins en mode SIMT de beaucoup de solutions/threads au sein d'un même bloc implique une importante sérialisation de l'algorithme.

Les accélérations obtenues avec la stratégie $RLI - FI^{globale}_{bloc}$, dont la valeur maximale est de 11,17, montrent que partager le travail de chaque solution entre plusieurs threads est plus

efficace. Par exemple, quand nb_{sol} est fixé à 2048, la stratégie $RLI-FI_{thread}$ utilise 2048 threads contre 81920 pour la stratégie $RLI - FI_{bloc}$. D'autre part, les accélérations augmentent de 51 à 318 villes puis diminuent légèrement pour augmenter à nouveau aux problèmes de 4461 et 5915 villes. Une légère baisse d'accélération est ensuite constatée pour le plus gros problème. Dans ce cas, la charge de travail et les grosses structures de données impliquent des latences mémoire dont les coûts augmentent plus vite que les bénéfices de la parallélisation du travail disponible.

Des améliorations supplémentaires peuvent être apportées sur les petits problèmes par l'utilisation de la mémoire partagée, introduite dans la stratégie $RLI-FI_{bloc}^{partagée}$. Cette dernière produit une accélération maximale de 8,51 pour le problème de 318 villes. Les résultats obtenus pour les plus gros problèmes montrent néanmoins que les limites de ce type de mémoire sont rapidement atteintes. En effet, les accélérations sont plus faibles en utilisant la mémoire partagée pour les problèmes de 783, 1577 et 2103 villes. De plus, les plus gros problèmes ne peuvent pas être résolus. Puisque la taille de cette mémoire est très limitée, le nombre de blocs actifs par multiprocesseur est réduit lorsque un trop grand nombre de données y sont stockées. Pour le problème de 2103, 1 bloc est actif pour la stratégie $RLI - FI_{bloc}^{partagée}$ contre 8 pour la stratégie $RLI - FI_{bloc}^{globale}$. Pour les problèmes dont la taille varie de 3038 à 13509 villes, la mémoire partagée ne peut pas contenir les données et aucun bloc ne peut être actif.

Une analyse de la qualité des solutions est fournie dans les Tableaux 5.7, 5.8 et 5.9. Ces derniers présentent la fréquence d'obtention de la solution optimale connue f_{opt}, le pourcentage moyen de déviation par rapport à l'optimum Δ_{moy} et la moyenne des pourcentages moyens de déviation Δ_{moy}^{seq} par rapport à la version séquentielle.

Problème	nb_{sol}	Séquentiel		$RLI - FI_{thread}$		$RLI - FI_{bloc}^{globale}$		$RLI - FI_{bloc}^{partagée}$	
		f_{opt}	Δ_{moy}	f_{opt}	Δ_{moy}	f_{opt}	Δ_{moy}	f_{opt}	Δ_{moy}
eil51, kroA100, d198, lin318	1	1,00	0,000	1,00	0,000	1,00	0,000	1,00	0,000
	8	1,00	0,000	1,00	0,000	1,00	0,000	1,00	0,000
	64	1,00	0,000	1,00	0,000	1,00	0,000	1,00	0,000
	128	1,00	0,000	1,00	0,000	1,00	0,000	1,00	0,000
	256	1,00	0,000	1,00	0,000	1,00	0,000	1,00	0,000
	512	1,00	0,000	1,00	0,000	1,00	0,000	1,00	0,000
	1024	1,00	0,000	1,00	0,000	1,00	0,000	1,00	0,000
	2048	1,00	0,000	1,00	0,000	1,00	0,000	1,00	0,000
		Δ_{moy}^{seq}			±0,000		±0,000		±0,000

Tableau 5.7 – Fréquence d'obtention de la solution optimale connue f_{opt} et pourcentage moyen de déviation par rapport à l'optimum Δ_{moy} pour chaque problème et chaque valeur de nb_{sol} pour les versions séquentielle et parallèles. Moyenne des pourcentages moyens de déviation Δ_{moy}^{seq} par rapport à la version séquentielle (partie 1).

Problème	nb_{sol}	Séquentiel		$RLI-FI_{thread}$		$RLI-FI_{bloc}^{globale}$		$RLI-FI_{bloc}^{partagée}$	
		f_{opt}	Δ_{moy}	f_{opt}	Δ_{moy}	f_{opt}	Δ_{moy}	f_{opt}	Δ_{moy}
rat783	1	0,95	0,003	1,00	0,000	1,00	0,000	1,00	0,000
	8	1,00	0,000	1,00	0,000	1,00	0,000	1,00	0,000
	64	1,00	0,000	1,00	0,000	1,00	0,000	1,00	0,000
	128	1,00	0,000	1,00	0,000	1,00	0,000	1,00	0,000
	256	0,65	0,006	0,80	0,002	0,75	0,003	0,85	0,002
	512	0,25	0,015	0,35	0,014	0,30	0,020	0,35	0,020
	1024	0,00	0,061	0,10	0,056	0,05	0,064	0,00	0,070
	2048	0,00	0,147	0,00	0,143	0,00	0,144	0,00	0,146
		Δ_{moy}^{seq}			−0,002		±0,000		−0,001
fl1577	1	0,50	0,141	0,50	0,328	0,50	0,328	0,50	0,339
	8	0,60	0,009	0,40	0,009	0,50	0,071	0,70	0,002
	64	0,80	0,003	0,70	0,004	0,80	0,010	0,50	0,002
	128	0,70	0,001	0,70	0,003	0,90	0,000	1,00	0,000
	256	0,20	0,006	0,50	0,005	0,40	0,004	0,40	0,003
	512	0,10	0,013	0,30	0,009	0,20	0,009	0,20	0,012
	1024	0,00	0,027	0,00	0,025	0,00	0,028	0,00	0,035
	2048	0,00	0,069	0,00	0,067	0,00	0,068	0,00	0,075
		Δ_{moy}^{seq}			+0,022		+0,030		+0,025
d2103	1	0,00	0,105	0,00	0,064	0,00	0,045	0,00	0,088
	8	0,00	0,029	0,00	0,027	0,00	0,004	0,00	0,009
	64	0,00	0,003	0,00	0,004	0,00	0,005	0,00	0,004
	128	0,00	0,004	0,00	0,005	0,00	0,004	0,00	0,004
	256	0,00	0,006	0,00	0,005	0,00	0,005	0,00	0,006
	512	0,00	0,007	0,00	0,006	0,00	0,005	0,00	0,006
	1024	0,00	0,010	0,00	0,010	0,00	0,011	0,00	0,010
	2048	0,00	0,020	0,00	0,016	0,00	0,020	0,00	0,018
		Δ_{moy}^{seq}			−0,016		−0,011		−0,005
pcb3038	1	0,00	0,194	0,00	0,143	0,00	0,277	-	-
	8	0,00	0,181	0,00	0,141	0,00	0,172	-	-
	64	0,00	0,302	0,00	0,314	0,00	0,299	-	-
	128	0,00	0,399	0,00	0,381	0,00	0,402	-	-
	256	0,00	0,511	0,00	0,509	0,00	0,509	-	-
	512	0,00	0,666	0,00	0,645	0,00	0,667	-	-
	1024	0,00	0,867	0,00	0,856	0,00	0,881	-	-
	2048	0,00	1,120	0,00	1,092	0,00	1,116	-	-
		Δ_{moy}^{seq}			−0,015		+0,014		-

Tableau 5.8 – Fréquence d'obtention de la solution optimale connue f_{opt} et pourcentage moyen de déviation par rapport à l'optimum Δ_{moy} pour chaque problème et chaque valeur de nb_{sol} pour les versions séquentielle et parallèles. Moyenne des pourcentages moyens de déviation Δ_{moy}^{seq} par rapport à la version séquentielle (partie 2).

Problème	nb_{sol}	Séquentiel		$RLI-FI_{thread}$		$RLI-FI_{bloc}^{globale}$		$RLI-FI_{bloc}^{partagée}$	
		f_{opt}	Δ_{moy}	f_{opt}	Δ_{moy}	f_{opt}	Δ_{moy}	f_{opt}	Δ_{moy}
fnl4461	1	0,00	0,210	0,00	0,191	0,00	0,243	-	-
	8	0,00	0,310	0,00	0,331	0,00	0,314	-	-
	64	0,00	0,625	0,00	0,594	0,00	0,623	-	-
	128	0,00	0,771	0,00	0,746	0,00	0,778	-	-
	256	0,00	0,958	0,00	0,936	0,00	0,963	-	-
	512	0,00	1,136	0,00	1,121	0,00	1,143	-	-
	1024	0,00	1,331	0,00	1,300	0,00	1,333	-	-
	2048	0,00	1,511	0,00	1,521	0,00	1,526	-	-
	Δ_{moy}^{seq}				−0,014		+0,008		-
rl5915	1	0,00	0,175	0,00	0,109	0,00	0,224	-	-
	8	0,00	0,079	0,00	0,050	0,00	0,070	-	-
	64	0,00	0,063	0,00	0,088	0,00	0,079	-	-
	128	0,00	0,103	0,00	0,108	0,00	0,090	-	-
	256	0,00	0,146	0,00	0,133	0,00	0,130	-	-
	512	0,00	0,209	0,00	0,191	0,00	0,201	-	-
	1024	0,00	0,356	0,00	0,307	0,00	0,320	-	-
	2048	0,00	0,509	0,00	0,530	0,00	0,507	-	-
	Δ_{moy}^{seq}				−0,015		−0,003		-
usa13509	1	0,00	0,282	-	-	0,00	0,377	-	-
	8	0,00	0,444	0,00	0,408	0,00	0,459	-	-
	64	0,00	0,894	0,00	0,895	0,00	0,929	-	-
	128	0,00	1,104	0,00	1,105	0,00	1,143	-	-
	256	0,00	1,305	0,00	1,287	0,00	1,368	-	-
	512	0,00	1,530	0,00	1,531	0,00	1,579	-	-
	1024	0,00	1,740	0,00	1,740	0,00	1,759	-	-
	2048	0,00	1,962	0,00	1,961	0,00	1,979	-	-
	Δ_{moy}^{seq}				−0,007		+0,041		-

Tableau 5.9 – Fréquence d'obtention de la solution optimale connue f_{opt} et du pourcentage moyen de déviation par rapport à l'optimum Δ_{moy} pour chaque problème et chaque valeur de nb_{sol} pour les versions séquentielle et parallèles. Moyenne des pourcentages moyens de déviation Δ_{moy}^{seq} par rapport à la version séquentielle (partie 3).

L'analyse de la moyenne des pourcentages moyens de déviation Δ_{moy}^{seq} par rapport à la version séquentielle indique que la qualité des solutions est légèrement améliorée d'au maximum $0,016\%$ ou légèrement dégradée d'au maximum $0,041\%$. Pour les petits problèmes, la solution optimale est toujours trouvée par les implémentations parallèles : $f_{opt} = 1,00$ et $\Delta_{moy} = 0,000$. Pour les deux problèmes de taille moyenne (783 et 1577 villes), la qualité des solutions obtenues est meilleure lorsque nb_{sol} atteint 64 ou 128 solutions. En effet, en augmentant le nombre de solutions, la probabilité de trouver une meilleure solution est plus grande. Cependant, si nb_{sol} augmente un peu plus, la solution optimale est généralement de moins en moins trouvée car le nombre d'itérations par solution devient trop petit pour fournir une recherche approfondie. Ainsi, la solution optimale n'est jamais trouvée pour les gros problèmes ($f_{opt} = 0,00$) et la

qualité des solutions est de plus en plus dégradée.

Pour résumer, associer chaque solution à un bloc est globalement la meilleure stratégie et apporte une accélération maximale de 11,17. Cependant, un compromis doit être réalisé entre l'accélération et la qualité de solution obtenues lorsque les paramètres des algorithmes parallèles sont choisis. Une étude sur la variation du nombre de candidats est donc effectuée afin de montrer l'influence de ce paramètre.

5.4.1.3 Variation du nombre de candidats

Certains paramètres fixés dans la littérature pour l'exécution des méthodes séquentielles sont mal adaptés à la parallélisation GPU. Par exemple, l'utilisation de 40 candidats pour la stratégie $RLI - FI_{bloc}$ ne maximise pas l'utilisation des capacités du GPU. En effet, comme précisé dans le Chapitre 2, la documentation CUDA [cud11] conseille d'utiliser un nombre de threads par bloc qui soit un multiple de la taille des warps pour maximiser son efficacité, c'est-à-dire 32 threads. L'impact de ce paramètre est donc étudié sur l'accélération et la qualité des solutions de la RLI. Un nombre de candidats égal à 64 a été choisi car c'est le premier multiple de 32 supérieur à 40. Un plus grand nombre de candidats/threads n'a pas été utilisé afin de ne pas trop altérer le fonctionnement de base de l'algorithme. Une comparaison des algorithmes séquentiels et parallèles utilisant 40 et 64 candidats est donc effectuée. Le Tableau 5.10 présente les temps séquentiels, les temps parallèles, l'accélération et la qualité des solutions obtenues par la stratégie $RLI - FI_{bloc}^{globale}$ avec $nb_{sol} = 2048$. La qualité des solutions est évaluée avec la fréquence d'obtention de la solution optimale connue f_{opt}, le pourcentage de déviation par rapport à l'optimum Δ_{moy} et le pourcentage moyen de déviation Δ_{moy}^{40} de la version utilisant 64 candidats par rapport celle utilisant 40 candidats. Le problème eil51 n'a pas pu être exécuté car le nombre de candidats est supérieur au nombre de villes.

En utilisant un nombre de threads multiple de 32 et une quantité de travail plus importante, l'accélération augmente généralement pour chaque problème mis à part pour ceux de 4461 et 5915 villes. Cependant, le temps séquentiel de la RLI augmente également puisqu'une quantité de travail supplémentaire est effectuée. L'analyse des solutions obtenues montre que l'optimum est trouvé aussi fréquemment dans tous les cas. De plus, la qualité des solutions est similaire pour les petits problèmes : $\Delta_{moy}^{40} = \pm 0,000$. Concernant les problèmes de plus grande taille, la qualité des solutions est légèrement améliorée d'au maximum $0,032\%$ ou légèrement dégradée d'au maximum $0,074\%$.

Lorsque le nombre de candidats est fixé à 64, l'algorithme est dans l'ensemble mieux adapté à la parallélisation GPU. Cependant, la qualité des solutions est légèrement dégradée. Un compromis entre l'accélération et la qualité des solutions obtenues est, là encore, nécessaire lorsque les paramètres de l'approche $RLI - FI$ sont choisis. La prochaine section analyse donc l'influence de la parallélisation GPU sur la règle de pivotement random-k-best-improvement de l'approche $RLI - RKBI$. Cette dernière utilise un nombre de voisins fixe et aucun mécanisme d'accélération.

Problème	Nombre de candidats	Temps séquentiel	Temps parallèle	Accélération	f_{opt}	Δ_{moy}	Δ^{40}_{moy}
kroA100	40	268,07	37,92	7,07	1,00	0,000	
	64	394,30	39,34	**10,02**	1,00	0,000	±0,000
d198	40	683,22	89,25	7,66	1,00	0,000	
	64	1155,27	100,67	**11,48**	1,00	0,000	±0,000
lin318	40	925,37	119,12	7,77	1,00	0,000	
	64	1378,16	114,02	**12,09**	1,00	0,000	±0,000
rat783	40	1272,12	198,87	6,40	0,00	0,144	
	64	1841,76	211,84	**8,69**	0,00	0,158	+0,014
fl1577	40	1614,39	289,29	5,58	0,00	0,068	
	64	3011,08	378,30	**7,96**	0,00	0,036	−0,032
d2103	40	1507,82	336,50	4,63	0,00	0,020	
	64	2443,37	368,04	**6,64**	0,00	0,014	−0,006
pcb3038	40	4431,25	791,83	5,60	0,00	1,116	
	64	6372,99	850,14	**7,50**	0,00	1,168	+0,052
fnl4461	40	16585,62	1499,07	**11,06**	0,00	1,526	
	64	19052,67	1831,63	10,40	0,00	1,599	+0,073
rl5915	40	19417,32	1737,89	**11,17**	0,00	0,507	
	64	23359,12	2249,78	10,38	0,00	0,527	+0,020
usa13509	40	58699,04	6784,15	8,65	0,00	1,979	
	64	86220,61	8088,23	**10,66**	0,00	2,053	+0,074

Tableau 5.10 – Temps séquentiel et parallèle, accélération et qualité des solutions (fréquence d'obtention de la solution optimale connue f_{opt}, pourcentage de déviation moyen par rapport à l'optimum Δ_{moy} et pourcentage de déviation moyen Δ^{40}_{moy} de la version utilisant 64 candidats par rapport celle utilisant 40 candidats) pour la stratégie $RLI - FI^{globale}_{bloc}$ avec $nb_{sol} = 2048$.

5.4.2 Approche $RLI - RKBI$

L'objectif des expérimentations de cette section est de montrer les gains de performance pouvant être obtenus avec la stratégie de parallélisation $RLI - RKBI_{blocs}$ spécialement conçue pour le GPU. Ces expérimentations sont divisées en deux phases. La première concerne le paramétrage du nombre d'itérations it_{total} et de voisins évalués k de la règle de pivotement random-k-best-improvement. La seconde permet de comparer les accélérations et la qualité des solutions obtenues aux résultats des stratégies de l'approche $RLI - FI$.

La première étape de ces expérimentations consiste donc à réaliser une étude empirique pour déterminer les paramètres it_{total} et k de la stratégie $RLI - RKBI_{blocs}$. nb_{sol} est fixé à 64 et les nombres de voisins évalués k testés à 8960000, 4480000, 2240000, 1120000, 560000, 280000 et 140000. Afin de trouver le nombre d'itérations total it_{total} correspondant, sa valeur est modifiée itérativement jusqu'à ce que le temps séquentiel obtenu soit le plus proche possible de celui obtenu par la version séquentielle de l'approche $RLI - FI$. De plus, it_{total} a été fixé au minimum à 127 itérations pour que chaque solution puisse effectuer au moins une itération de la RLI. Le nombre de threads par bloc a été fixé à 64 car c'est un multiple de 32 qui

maximise généralement le nombre de blocs actifs par multiprocesseur. Pour le premier noyau de la stratégie $RLI-RKBI_{blocs}$, le nombre de blocs par solution a été fixé empiriquement selon le Tableau 5.11. Pour le second noyau, le nombre de blocs total lancé est égal à nb_{sol} puisque chaque bloc est associé à une solution.

Problème	8960000	4480000	2240000	1120000	560000	280000	140000
kroA100	30	30	30	30	30	30	30
lin318	40	40	30	30	30	30	30
fl1577	60	65	60	60	60	60	60
pcb3038	95	95	90	90	90	90	90
rl5915	110	100	110	110	110	100	100
usa13509	90	100	100	100	110	90	90

Tableau 5.11 – Nombre de blocs par solution selon le problème et le nombre de voisins k.

Afin de déterminer k et it_{total}, l'analyse des résultats obtenus est nécessaire. Le Tableau 5.12 présente le temps séquentiel, le temps parallèle et les accélérations obtenues par les stratégies $RLI-FI_{bloc}^{globale}$ (FI) et $RLI-RKBI_{blocs}$ (RKBI) pour chaque problème et chaque configuration k/it_{total} lorsque $nb_{sol}=64$. Le Tableau 5.13 fournit, quant à lui, la qualité des solutions obtenues : la fréquence d'obtention de la solution optimale connue f_{opt}, le pourcentage moyen de déviation par rapport à l'optimum Δ_{moy} et le pourcentage de déviation moyen Δ_{moy}^{FI} de la stratégie $RLI-RKBI_{blocs}$ par rapport à la stratégie $RLI-FI_{bloc}^{globale}$. Ces résultats ont été calculés à partir de 20 essais pour les problèmes de moins de 1000 villes et 10 essais pour les plus grandes instances.

Les résultats du Tableau 5.12 montrent que le temps parallèle obtenu par la stratégie $RLI-RKBI_{blocs}$ est beaucoup plus petit que celui obtenu par la stratégie $RLI-FI_{bloc}^{globale}$. De même, les accélérations sont multipliées jusqu'à 16 fois avec une valeur maximale de 45,71. Elles sont également supérieures à la meilleure accélération obtenue quelque soit nb_{sol} et la stratégie de l'approche $RLI-FI$. La règle de pivotement random-k-best-improvement et la stratégie $RLI-RKBI_{blocs}$ sont donc mieux adaptées à la parallélisation GPU. Cependant, la qualité de solution est très dégradée, comme le montre le Tableau 5.13 : jusqu'à $18,683\%$ de dégradation par rapport à la stratégie $RLI-FI_{bloc}^{globale}$.

Afin de réaliser la deuxième partie des expérimentations, les valeurs du nombre de voisins évalués k et du nombre d'itérations it_{total} doivent être fixés. Un compromis entre les accélérations et la qualité des solutions obtenues est donc effectué. Tout d'abord, le premier résultat pour les problèmes de 1577, 3038 et 13509 villes (en italique et grisé) n'est pas pris en compte puisque le temps d'exécution séquentiel est beaucoup trop long par rapport à la stratégie $RLI-FI_{bloc}^{globale}$ alors que le nombre d'itérations minimal a été fixé. Ensuite, deux valeurs sont affectées à chaque combinaison : leur rang $rang_{accélération}$ lorsqu'elles sont classées par ordre décroissant d'accélération et leur rang $rang_{qualité}$ lorsqu'elles sont classées par ordre croissant de qualité de solution (Δ_{moy}). Pour chaque problème, la somme $rang_{somme}$ de ces deux rangs est effectuée. La configuration k/it_{total} alors choisie est celle qui produit la plus petite valeur. Le Tableau 5.14 prend l'exemple du problème rl5915 pour lequel la configuration 8960000/200 est retenue. De la même façon, les configurations choisies pour les autres problèmes sont : 2240000/3000 pour kroA100, 4480000/2500 pour lin318, 2240000/1200 pour fl1577, 2240000/1000 pour pcb3038 et 4480000/300 pour usa13509.

Problème	k	it_{total}	Temps séquentiel		Temps parallèle		Accélération		
			FI	RKBI	FI	RKBI	FI	FI_m	RKBI
kroA100	8960000	700	267,00	282,59	94,23	6,65	2,83	7,83	42,50
	4480000	1500		275,92		6,64			41,50
	2240000	**3000**		**300,90**		**6,58**			**45,71**
	1120000	5500		299,78		6,62			45,29
	560000	10000		278,24		6,87			40,51
	280000	18000		274,02		6,99			39,20
	140000	33000		266,32		7,37			36,14
lin318	8960000	1000	915,93	923,27	283,85	21,22	3,23	8,51	43,51
	4480000	**2500**		**944,52**		**21,84**			**43,26**
	2240000	6000		977,03		23,66			41,30
	1120000	11000		972,93		21,43			45,40
	560000	23000		987,39		22,88			43,16
	280000	43000		943,13		24,03			39,24
	140000	70000		1024,33		26,44			38,75
fl1577	*8960000*	*127*	1474,16	*2170,08*	854,32	*84,46*	1,73	5,58	*25,69*
	4480000	400		1464,79		55,30			26,49
	2240000	**1200**		**1576,80**		**53,41**			**29,52**
	1120000	2000		1495,71		54,74			27,33
	560000	3000		1565,27		53,80			29,09
	280000	5000		1713,72		57,41			29,85
	140000	10000		1685,32		64,95			25,95
pcb3038	*8960000*	*127*	4306,15	*10885,04*	1432,43	*291,52*	3,01	5,60	*34,59*
	4480000	400		4291,12		147,36			29,12
	2240000	**1000**		**4849,50**		**152,96**			**31,70**
	1120000	1500		4146,71		128,41			32,29
	560000	3000		4096,93		132,91			30,83
	280000	6000		4308,74		134,69			31,99
	140000	15000		4062,37		148,65			27,33
rl5915	**8960000**	**200**	13731,77	**14069,96**	2994,50	**364,92**	4,59	11,17	**38,56**
	4480000	800		13385,87		368,53			36,32
	2240000	1500		13347,03		348,41			38,31
	1120000	3000		13320,80		352,57			37,78
	560000	8000		14024,25		429,36			32,66
	280000	15000		10627,67		332,01			32,01
	140000	50000		13531,40		399,63			33,86
usa13509	*8960000*	*127*	50684,82	*86053,65*	10361,19	*2762,27*	4,89	9,01	*31,15*
	4480000	300		**57859,51**		**1801,97**			**32,11**
	2240000	1000		54219,37		1770,24			30,63
	1120000	3000		47141,25		1871,92			25,18
	560000	10000		50976,67		2039,67			24,99
	280000	30000		52533,57		1951,75			26,92
	140000	100000		52730,81		1933,90			27,27

Tableau 5.12 – Nombre de voisins k et d'itérations it_{total}, temps séquentiel, temps parallèle et accélérations pour la stratégie $RLI-FI_{bloc}^{globale}$ (FI) et la stratégie $RLI-RKBI_{blocs}$ (RKBI) pour chaque problème et pour $nb_{sol} = 64$. L'accélération FI_m correspond à la meilleure accélération obtenue par problème quelque soit nb_{sol} et quelque soit la stratégie de l'approche $RLI - FI$.

Problème	k	it_{total}	f_{opt}		Δ_{moy}		Δ^{FI}_{moy}
			FI	RKBI	FI	RKBI	
kroA100	8960000	700	1,00	1,00	0,000	0,000	±0,000
	4480000	1500		1,00		0,000	±0,000
	2240000	**3000**		**1,00**		**0,000**	±0,000
	1120000	5500		1,00		0,000	±0,000
	560000	10000		1,00		0,000	±0,000
	280000	18000		1,00		0,000	±0,000
	140000	33000		1,00		0,000	±0,000
lin318	8960000	1000	1,00	0,00	0,000	0,549	+0,549
	4480000	**2500**		**0,00**		**0,351**	+0,351
	2240000	6000		0,00		0,777	+0,777
	1120000	11000		0,00		0,633	+0,633
	560000	23000		0,00		0,564	+0,564
	280000	43000		0,00		0,387	+0,387
	140000	70000		0,05		0,202	+0,202
fl1577	*8960000*	*127*	0,80	*0,00*	0,001	*1,868*	+1,867
	4480000	400		0,00		1,836	+1,835
	2240000	**1200**		**0,00**		**1,831**	+1,830
	1120000	2000		0,00		1,932	+1,931
	560000	3000		0,00		2,004	+2,003
	280000	5000		0,00		2,477	+2,476
	140000	10000		0,00		3,442	+3,441
pcb3038	*8960000*	*127*	0,00	*0,00*	0,299	*5,052*	+4,753
	4480000	400		0,00		5,168	+4,869
	2240000	**1000**		**0,00**		**5,162**	+4,863
	1120000	1500		0,00		5,461	+5,162
	560000	3000		0,00		6,023	+5,724
	280000	6000		0,00		6,871	+6,572
	140000	15000		0,00		8,777	+8,478
rl5915	**8960000**	**200**	0,00	**0,00**	0,079	**4,583**	+4,504
	4480000	800		0,00		4,692	+4,613
	2240000	1500		0,00		5,235	+5,156
	1120000	3000		0,00		6,069	+5,990
	560000	8000		0,00		7,764	+7,685
	280000	15000		0,00		10,929	+10,850
	140000	50000		0,00		15,349	+15,270
usa13509	*8960000*	*127*	0,00	*0,00*	0,929	*5,433*	+4,504
	4480000	300		**0,00**		**5,659**	+4,730
	2240000	1000		0,00		6,210	+5,281
	1120000	3000		0,00		7,304	+6,375
	560000	10000		0,00		9,451	+8,522
	280000	30000		0,00		13,579	+12,650
	140000	100000		0,00		19,612	+18,683

Tableau 5.13 – Nombre de voisins k et d'itérations it_{total}, fréquence d'obtention de l'optimum f_{opt} et pourcentage moyen de déviation par rapport à l'optimum Δ_{moy} pour les stratégies $RLI - FI^{globale}_{bloc}$ (FI) et $RLI - RKBI_{blocs}$ (RKBI) avec $nb_{sol} = 64$. Pourcentage de déviation moyen Δ^{FI}_{moy} de la stratégie $RLI - RKBI_{blocs}$ par rapport à la stratégie $RLI - FI^{globale}_{bloc}$.

k	it_{total}	$rang_{accélération}$	$rang_{qualité}$	$rang_{somme}$
8960000	**200**	**1**	**1**	**2**
4480000	800	4	2	6
2240000	1500	2	3	5
1120000	3000	3	4	7
560000	8000	6	5	11
280000	30000	7	6	13
140000	50000	5	7	12

Tableau 5.14 – Exemple de classement par rang pour le problème de 5915 villes.

La seconde partie des expérimentations consiste, pour chaque problème, à faire évoluer nb_{sol} et à comparer les résultats obtenus. Le Tableau 5.15 fournit le nombre de blocs par solution pour évaluer les k voisins qui a été choisi de façon empirique. La Figure 5.12 montre les accélérations obtenues pour la stratégie $RLI - RKBI_{blocs}$ (rkbi) comparé à celles obtenues pour la stratégie $RLI - FI_{bloc}$ (fi). Ces accélérations ont été calculées avec les temps séquentiels présentés dans le Tableau 5.16.

	1	8	64	128	256	512	1024	2048
kroA100	120	70	30	15	10	5	2	1
lin318	110	70	40	20	10	5	3	2
fl1577	140	90	60	50	20	15	15	15
pcb3038	120	100	90	80	30	20	15	20
rl5915	110	90	110	90	40	30	20	30
usa13509	120	90	100	90	40	40	20	30

Tableau 5.15 – Nombre de blocs par solution pour les différents problèmes et valeurs de nb_{sol}.

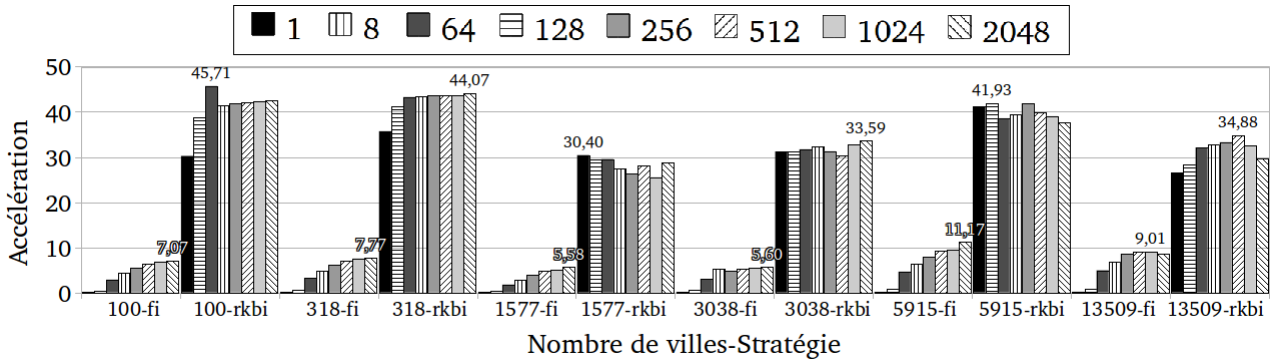


Figure 5.12 – Accélérations obtenues pour les stratégies $RLI - RKBI_{blocs}$ (rkbi) et $RLI - FI_{bloc}$ (fi) pour chaque problème selon nb_{sol}.

Avec la stratégie $RLI - RKBI_{blocs}$, des accélérations significatives ont été obtenues quelque soit la valeur de nb_{sol}. Une accélération maximale de 45,71 a été atteinte lorsque $nb_{sol} = 64$ pour le problème de 100 villes. L'augmentation de la taille du voisinage, la meilleure répartition des voisins entre les blocs et la réduction de la divergence des threads permet de mieux exploiter les ressources du GPU. Utiliser des structures de données de taille fixe pour stocker les meilleurs

voisins permet également de tirer profit de la mémoire partagée rapide même pour les plus gros problèmes. De plus, mis à part pour les deux plus petits problèmes avec $nb_{sol} = 1$, les accélérations sont du même ordre de grandeur quelque soit la valeur de nb_{sol} pour n'importe quel problème. Cette constatation permet de montrer la scalabilité de la stratégie de distribution du voisinage pour différentes tailles de population. Une comparaison avec les accélérations obtenues pour la stratégie $RLI - FI_{bloc}$ montre que quand l'accélération est considérée, la stratégie $RLI - RKBI_{blocs}$ est beaucoup plus performante.

Problème	nb_{sol}	Temps séquentiel	f_{opt}	Δ_{moy}
kroA100	1	255,84	1,00	0,000
	8	258,78	1,00	0,000
	64	300,90	1,00	0,000
	128	290,15	1,00	0,000
	256	314,65	1,00	0,000
	512	365,48	1,00	0,000
	1024	463,38	1,00	0,000
	2048	745,77	1,00	0,000
lin318	1	737,57	0,00	0,785
	8	774,93	0,00	0,682
	64	944,52	0,00	0,828
	128	1115,64	0,00	0,858
	256	1454,52	0,00	0,821
	512	2105,76	0,00	0,783
	1024	3529,38	0,00	0,840
	2048	6403,74	0,00	0,726
fl1577	1	979,30	0,00	1,034
	8	1041,28	0,00	1,703
	64	1576,80	0,00	1,831
	128	2101,96	0,00	1,863
	256	3084,09	0,00	1,822
	512	5793,20	0,00	1,776
	1024	9782,15	0,00	1,742
	2048	21164,38	0,00	1,741

Problème	nb_{sol}	Temps séquentiel	f_{opt}	Δ_{moy}
pcb3038	1	2416,13	0,00	4,636
	8	2798,58	0,00	4,940
	64	4849,50	0,00	5,162
	128	7154,08	0,00	5,186
	256	11021,79	0,00	5,136
	512	18701,70	0,00	5,104
	1024	41436,90	0,00	5,057
	2048	85017,66	0,00	4,970
rl5915	1	3053,57	0,00	4,408
	8	4630,76	0,00	4,701
	64	14069,96	0,00	4,583
	128	24688,60	0,00	4,484
	256	106595,00	0,00	4,422
	512	100035,45	0,00	4,357
	1024	216645,77	0,00	4,321
	2048	403247,45	0,00	4,247
usa13509	1	17061,78	0,00	6,138
	8	23322,68	0,00	5,894
	64	57859,51	0,00	5,659
	128	97796,57	0,00	5,645
	256	177806,04	0,00	5,614
	512	372594,10	0,00	5,585
	1024	693356,14	0,00	5,517
	2048	1264743,56	0,00	5,510

Tableau 5.16 – Temps séquentiel et qualité de solution (fréquence d'obtention de la solution optimale connue f_{opt} et pourcentage de déviation par rapport à l'optimum Δ_{moy}).

L'analyse du Tableau 5.16 montre cependant que les gains d'accélération impliquent également la détérioration de la qualité des solutions. Plus nb_{sol} augmente, plus la qualité est dégradée dans n'importe quel cas. Le choix aléatoire des voisins et l'incapacité d'utiliser les mécanismes d'accélération de l'approche $RLI - FI$ ne permet pas à la stratégie $RLI - RKBI_{blocs}$ de garder le même niveau d'optimisation que les stratégies $RLI - FI_{thread}$ et $RLI - FI_{bloc}$ pour le même temps d'exécution. De plus, l'approche $RLI - RKBI$ a conservé la même structure de RLI (génération, perturbation et critère d'acceptation) que l'approche $RLI - FI$ alors que le type de RL a été modifié, ce qui peut avoir des conséquences sur la qualité des solutions obtenues. Cette dernière est certes beaucoup liée à la RL utilisée mais également aux autres composants de la

RLI. En effet, le meilleur choix de perturbation dépend de la RL alors que celui du critère d'acceptation dépend de la RL et de la perturbation [LMS10]. L'ensemble de ces résultats montre, encore une fois, qu'un compromis doit être trouvé entre l'accélération et la qualité de solutions obtenues lorsque des algorithmes de RLI parallèles sont conçus pour les GPU. De nouvelles pistes doivent également être pensées pour restaurer, si possible, l'équilibre délicat entre tous les composants de l'algorithme dans ce contexte.

5.5 Conclusion

L'objectif de ce chapitre était de concevoir des stratégies de parallélisation efficaces de la Recherche Locale Itérée sur GPU pour résoudre le problème du Voyageur de Commerce. Elles se distinguent par la règle de pivotement utilisée : first-improvement pour l'approche $RLI-FI$ et random-k-best-improvement pour l'approche $RLI-RKBI$. Dans le premier cas, la méthodologie de parallélisation employée pour l'algorithme d'Optimisation par Colonie de Fourmis est transposée à la RLI démontrant ainsi la généricité du cadre de référence fourni par la taxonomie proposée dans le Chapitre 3. D'après cette dernière, les stratégies $RLI-FI_{thread}$ et $RLI-FI_{bloc}$ sont de type $pop_{gpu}sol_{th}$ et $pop_{gpu}sol_{bl}elt_{th}$ respectivement. Elles associent la phase de Recherche Locale à l'exécution des processeurs et des multiprocesseurs respectivement. Elles produisent des accélérations maximales de 2,89 et 11,17 avec une qualité de solutions souvent égale ou proche des optima pour les petits problèmes et légèrement dégradée pour les plus gros.

La stratégie $RLI-FI_{thread}$ produit les plus basses accélérations car elle n'exécute pas assez de threads en parallèle pour dissimuler efficacement les latences mémoires. De plus, la divergence de code induite par le calcul des voisins au sein d'un même bloc implique une importante sérialisation de l'algorithme. La stratégie $RLI-FI_{bloc}$ montre, quant à elle, que partager le travail entre un grand nombre de threads est plus efficace. Des améliorations supplémentaires au niveau de l'accélération peuvent être apportées sur les petits problèmes par l'utilisation de la mémoire partagée mais ses limites sont rapidement atteintes. Pour les problèmes de taille moyenne, les accélérations obtenues sont plus faibles en utilisant la mémoire partagée et les plus gros problèmes ne peuvent pas être résolus. Dans la stratégie $RLI-FI_{bloc}$, les paramètres de la RLI peuvent également influencer le temps de résolution des méthodes parallèles. Par exemple, choisir un nombre de candidats multiple de la taille des *warps* permet d'augmenter les accélérations obtenues dans la majorité des cas mais détériore légèrement la qualité des solutions.

Par l'utilisation de la règle de pivotement first-improvement et les mécanismes réduisant les temps d'exécution, ces stratégies produisent des défauts importants dans l'utilisation des ressources de calcul du GPU. Dans le but de dépasser ces limites, la stratégie $RLI-RKBI_{blocs}$ a spécialement été conçue pour tirer avantage des capacités du GPU. Cette stratégie hybride est basée sur une évaluation synchrone d'un nombre fixe de voisins. Le premier noyau est de type $pop_{gpu}sol_{blS}elt_{th}$. Il permet d'augmenter la taille de voisinage et d'associer l'évaluation de chaque solution à plusieurs blocs. Le second noyau est de type $pop_{gpu}sol_{bl}elt_{th}$ et assigne chaque solution à un bloc. Des accélérations significatives allant jusqu'à 45,71 ont été obtenues au prix d'une détérioration variable de la qualité des solutions. À titre de comparaison, Luong *et al.* [LMT10b, LMT11b] résolvaient le problème du VC avec une RL à large voisinage et obtenaient

une accélération maximale de 19,70. O'Neil *et al.* [OTB11] utilisaient un algorithme de RL à relances multiples spécialisé pour un problème de 100 villes et rapportaient une accélération maximale de 61,90. Enfin, Luong *et al.* [LMT11a] utilisaient également un algorithme de RL à relances multiples pour résoudre le problème d'affectation quadratique et obtenaient une accélération maximale de 12.

Bien souvent, l'exploitation maximale des ressources du GPU nécessite des configurations algorithmiques qui ne laissent pas la RLI effectuer une exploration et une exploitation efficaces de l'espace de recherche. Par conséquent, la performance de ces stratégies est fortement influencée par les effets combinés des paramètres techniques de la métaheuristique (règle de pivotement, listes de candidats, etc.), de la taille du voisinage et de la granularité de la parallélisation. Cela mène à l'idée que résoudre le problème du Voyageur de Commerce et d'autres problèmes d'optimisation combinatoire sur GPU est actuellement une question de compromis entre accélération et efficacité de la recherche.

Chapitre 6

Conclusion et perspectives

Le travail effectué dans cette thèse a permis de mettre en évidence la contribution significative que pouvait apporter le parallélisme sur GPU (*Graphics Processing Units*) à l'optimisation combinatoire et plus spécifiquement, aux métaheuristiques. Il a également souligné les difficultés d'algorithmique et de programmation pouvant être rencontrées lors de l'exploitation des ressources de calcul disponibles sur ce type d'architecture. Malgré le nombre grandissant de travaux réalisés au cours de ces quatre dernières années, ce domaine est encore assez jeune. Il reste donc encore beaucoup de travail conceptuel, technique et comparatif à accomplir dans le but d'exploiter cette architecture abordable et massivement parallèle pour l'optimisation combinatoire. Plusieurs implémentations de métaheuristiques parallèles sur GPU ont permis de résoudre plus rapidement un grand nombre de problèmes mais n'ont pas permis d'apporter une vision globale et fondamentale à ce domaine. Il était donc souhaitable de formaliser ces différentes implémentations dans un cadre méthodologique général cohérent. Par conséquent, cette thèse a permis d'apporter des contributions à travers l'élaboration d'une revue de littérature multi-domaines, la proposition d'une taxonomie originale de métaheuristiques parallèles sur GPU et le développement de stratégies de parallélisation performantes basées sur cette taxonomie.

Ces travaux de recherche se situent à l'intersection des domaines du parallélisme et des métaheuristiques. Une revue de la littérature a donc été réalisée en trois parties : architecture GPU/modèle de programmation CUDA, métaheuristiques/parallélisation GPU et classifications générales des métaheuristiques parallèles. Ce processus a permis d'apporter une vue générale de ces deux domaines ainsi que de leur fusion, tout en soulignant certaines questions importantes. Deux objectifs de recherche ont alors été formulés afin de répondre à ces questions.

Le premier objectif de cette thèse consistait à proposer une taxonomie originale des métaheuristiques parallèles sur architectures GPU afin de classifier les implémentations recensées et de formaliser les stratégies de parallélisation sur GPU dans un cadre méthodologique général cohérent. Dans un premier temps, les éléments clés des métaheuristiques et de l'architecture GPU ont été identifiés. Leur analyse a permis d'élaborer une taxonomie basée sur deux dimensions. La première associe les composantes de la métaheuristique aux différents niveaux de parallélisme offerts par la structure des éléments de calcul du GPU. La seconde spécifie le type de mémoire du GPU utilisée pour stocker les différentes données nécessaires au fonctionnement

de la métaheuristique. La parallélisation des métaheuristiques sur GPU a donc été généralisée ce qui permet de faciliter la conception de stratégies et le développement de nouvelles implémentations.

Le deuxième objectif visait à valider cette taxonomie dans un contexte appliqué de conception et de développement de méthodes d'optimisation parallèles compétitives. Plusieurs stratégies de parallélisation originales et performantes de métaheuristiques à population de solutions et à solution unique ont donc été proposées dans la résolution du problème du Voyageur de Commerce. Le processus de parallélisation a été effectué en exploitant les principales composantes de la taxonomie. Une étude expérimentale structurée et minutieuse a ensuite été réalisée de façon à évaluer l'influence des stratégies proposées sur la qualité des solutions obtenues et sur la réduction du temps de calcul. De plus, un effort particulier a été mis en œuvre afin de comparer les résultats obtenus à ceux des implémentations séquentielles originales définies dans la littérature de référence.

La parallélisation d'une métaheuristique à population de solutions a tout d'abord été étudiée. Plusieurs implémentations parallèles performantes d'un algorithme d'Optimisation par Colonie de Fourmis ont été proposées. Les deux premières correspondent à l'approche $FOURMI$ et associent la construction des tournées des fourmis à l'exécution d'un processeur ou d'un multiprocesseur. Les deux autres correspondent à l'approche $COLONIE$ et attribuent plusieurs colonies entières aux multiprocesseurs ou aux GPU. Dans tous les cas, les données des fourmis peuvent être stockées dans la mémoire rapide de petite taille du GPU. Les résultats obtenus ont montré que la qualité des solutions trouvées par les versions séquentielles est meilleure que celle obtenue dans les travaux de la littérature. De bonnes performances en accélération ont été rapportées avec une qualité de solution déviant entre $-1,33\%$ et $+1,64\%$ des versions séquentielles. Dans l'ensemble, cela prouve que les deux approches générales peuvent être efficacement implémentées sur une architecture GPU et qu'il est possible de réduire considérablement le temps d'exécution des algorithmes OCF. Cette étude comparative a permis de souligner l'impact de la granularité de la parallélisation sur la performance des stratégies parallèles, mais également celui des paramètres de l'OCF, des configurations techniques du GPU et des différents type de mémoire.

La parallélisation d'une métaheuristique à solution unique associée à une méthode de Recherche Locale a ensuite été étudiée. Plusieurs implémentations parallèles performantes de la Rercherche Locale Itérée qui reposent sur de multiples recherches ont été proposées. Elles se distinguent principalement par la règle de pivotement employée. La première approche $RLI-FI$ utilise la règle *first-improvement*. Dans ce cas, la méthodologie de parallélisation employée pour l'OCF a été transposée à la Recherche Locale Itérée démontrant ainsi la généricité du cadre proposé. Une réduction du temps de calcul a été obtenue tout en conservant une qualité de solution souvent égale ou proche des optima. L'approche $RLI - RKBI$ utilise une règle de pivotement spécialement conçue pour tirer bénéfice des ressources de calcul du GPU. Elle offre un compromis entre les règles *first-improvement* et *best-improvement*. Cette stratégie hybride permet ainsi d'associer une recherche à plusieurs multiprocesseurs tout en étant basée sur une évaluation synchrone d'un nombre fixe de voisins. Des performances significatives en accélération ont été obtenues au prix d'une détérioration variable de la qualité des solutions. Cette étude comparative a donc souligné l'influence de la règle de pivotement, de la taille du voisinage et de la granularité de la parallélisation sur le niveau de performance obtenu. Elle a également mené

à l'idée que résoudre le problème du Voyageur de Commerce et d'autres problèmes d'optimisation combinatoire sur GPU est actuellement une question de compromis entre accélération et efficacité de la recherche.

En résumé, les implémentations parallèles proposées dans les Chapitres 4 et 5 ont permis de montrer l'efficacité de plusieurs stratégies définies à partir de la taxonomie proposée. Elles ont également mis en évidence l'importance des deux dimensions de cette taxonomie pour la parallélisation GPU qui influencent grandement la performance des stratégies de parallélisation. Cette thèse a donc atteint ses objectifs tout en ouvrant la voie à plusieurs perspectives de recherche futures.

Bien souvent, l'exploitation maximale des ressources du GPU nécessitent des configurations algorithmiques qui ne laissent pas les métaheuristiques effectuer une exploration et une exploitation efficaces de l'espace de recherche. Cela mène donc à trouver un compromis entre accélération et efficacité de la recherche. Dans ce contexte, une façon d'améliorer la longueur des tournées serait d'ajouter des stratégies d'échange d'informations entre les colonies pour l'OCF et entre les solutions pour la RLI. De plus, une étape ultérieure consistera à examiner d'autres versions avancées de la métaheuristique OCF comme le *Ant Colony System* et à étudier d'autres approches de décomposition de la Recherche Locale comme le partionnement basé sur les tournées. Suivant cette ligne de pensée, de futurs travaux visent à utiliser le cadre et les connaissances accumulées dans ce manuscrit pour proposer des métaheuristiques OCF et RLI spécifiquement conçues pour l'exécution sur GPU.

De plus, afin de fournir une meilleure compréhension des goulets d'étranglement identifiés dans ce travail, notamment ceux relatifs à la mémoire et à la répartition des processus de recherche, une analyse plus approfondie serait souhaitable. Par exemple, les exigences relatives aux différents types de mémoire (accélérateur, partagée, etc.) pourraient être analysées en fonction de la taille du problème, du nombre de fourmis ou de voisins, du nombre de colonies ou de solutions, du nombre de multiprocesseurs et de processeurs, etc. Une telle analyse pourrait conduire à la proposition d'algorithmes qui déterminent automatiquement les configurations thread/bloc/GPU les plus efficaces pour l'OCF, la RLI et d'autres métaheuristiques. L'acceptation globale de GPU en tant que composants de systèmes d'optimisation nécessite des algorithmes et logiciels qui ne sont pas seulement efficaces, mais aussi utilisables par un large éventail d'universitaires et de praticiens.

Durant les dernières années, les processeurs graphiques NVIDIA et l'environnement de programmation CUDA étaient précurseurs dans le domaine du calcul haute performance sur accélérateurs matériels. Par conséquent, les travaux de parallélisation de métaheuristiques recensés dans cette thèse et les contributions proposées sont principalement basés sur cette technologie. Cependant, l'utilisation d'accélérateurs semble en voie de se généraliser et de nouvelles architectures émergent comme les processeurs Intel Xeon Phi. Comme celles-ci semblent destinées à arborer des modèles de calcul de type SIMD ainsi que de multiples niveaux hiérarchiques de processeurs et de mémoires, une extension naturelle des travaux de cette thèse consiste a généraliser la taxonomie et les approches proposées à un ensemble d'accélérateurs matériels. Au niveau expérimental, il est également intéressant d'implémenter les approches avec des modèles de programmation autres que CUDA, notamment OpenCL qui vise à être adaptable à différents types d'accélérateurs matériels. Enfin, les implémentations parallèles proposées pour le problème du VC pourraient être généralisés à d'autres problèmes académiques et industriels

connexes. Cette thèse, au même titre que plusieurs autres travaux, a montré l'apport important que peut apporter le parallélisme aux métaheuristiques et l'intérêt de l'unification des approches. Elle représente une première étape vers une approche de parallélisation unifiée des métaheuristiques sur des architectures parallèles basées sur les accélérateurs matériels et les GPU.

Liste des publications

[DDGK10] A. Delévacq, P. Delisle, M. Gravel, and M. Krajecki. Parallel Ant Colony Optimization on Graphics Processing Units. In R.R. Hashimi H.R. Arabnia and A.M.G. Solo (Eds.), editors, *16th International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'10)*, pages 42–48. CSREA Press, Las Vegas, États-Unis, 12-15 Juillet 2010.

[DDGK13] A. Delévacq, P. Delisle, M. Gravel, and M. Krajecki. Parallel Ant Colony Optimization on Graphics Processing Units. *Journal of Parallel and Distributed Computing*, 73(1) : 52–61, 2013.

[DDK10] A. Delévacq, P. Delisle, and M. Krajecki. MAX-MIN Ant System on Graphics Processing Units. In *Proceedings of the 3rd International Conference on Metaheuristics and Nature Inspired Computing (META'10)*. Djerba, Tunisie, 27-31 Octobre 2010.

[DDK11] A. Delévacq, P. Delisle, and M. Krajecki. MAX-MIN Ant System parallèle sur processeurs graphiques (GPU). In *Rencontres francophones du Parallélisme (RenPar'20)*. Saint-Malo, France, 10-13 Mai 2011.

[DDK12a] A. Delévacq, P. Delisle, and M. Krajecki. Optimisation par Colonie de Fourmis parallèle sur architectures hybrides multi-coeur/multi-GPU. In *13e congrès annuel de la Société française de Recherche Opérationnelle et d'Aide à la Décision*. Angers, France, 11-13 Avril 2012.

[DDK12b] A. Delévacq, P. Delisle, and M. Krajecki. Parallel GPU implementation of Iterated Local Search for the Travelling Salesman Problem. In *Learning and Intelligent OptimizatioN (LION 6)*. Lecture Notes in Computer Science, Paris, France, 16-20 Janvier 2012.

[DDK12c] A. Delévacq, P. Delisle, and M. Krajecki. Parallelization strategies for Local Search algorithms on Graphics Processing Units. In *18th International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'12)*. Las Vegas, États-Unis, 16-19 Juillet 2012.

Bibliographie

[AD08] E. Alba and B. Dorronsoro. *Cellular Genetic Algorithms*. Operations Research/Computer Science Interfaces Series. Springer, 2008.

[AL03] E. Aarts and J.K. Lenstra. *Local Search in Combinatorial Optimization*. Princeton University Press, 2003.

[Alb02] E. Alba. Parallel Evolutionary Algorithms can achieve super-linear performance. *Information Processing Letters*, 82 :7–13, 2002.

[ALO07] E. Alba, G. Leguizamon, and G. Ordonez. Two models of parallel ACO algorithms for the minimum tardy task problem. *International Journal of High Performance Systems Architecture*, 1(1) :50–59, 2007.

[amd12] Advanced Micro Devices, Inc. *AMD OpteronTM 6200 Series Processor Quick Reference Guide*, 2012.

[And96] E.J. Anderson. Mechanisms for Local Search. *European Journal of Operational Research*, 88(1) :139–151, 1996.

[App06] D.L. Applegate. *The Traveling Salesman Problem : a computational study*. Princeton series in applied mathematics. Princeton University Press, 2006.

[ATD10] R. Arora, R. Tulshyan, and K. Deb. Parallelization of binary and real-coded Genetic Algorithms on GPU using CUDA. In *IEEE Congress on Evolutionary Computation*, pages 1–8. IEEE, 2010.

[Bar96] H.J.C. Barbosa. A Genetic Algorithm for min-max problems. In *Proceedings of the 1st international conference on Evolutionary Computation and its Applications*, pages 99–109. 1996.

[BEG04] M. Belal and P. El-Ghazawi. Parallel models for Particle Swarm Optimizers. *International Journal of Intelligent Computing and Information Sciences (IJICIS)*, 4(1) : 100–111, 2004.

[Ben92] J.L. Bentley. Fast algorithms for geometric Traveling Salesman Problems. *ORSA Journal on Computing*, 4(4) :387–411, 1992.

[BFM97] T. Back, D.B. Fogel, and Z. Michalewicz. *Handbook of Evolutionary Computation*. Taylor & Francis, 1997.

[BH93] R.S. Barr and B.L. Hickman. Reporting computational experiments with parallel algorithms : Issues, measures and experts' opinions. *ORSA Journal on Computing*, 5(1) :2–18, 1993.

[BHS97] B. Bullnheimer, R.F. Hartl, and C. Strauss. A new rank based version of the Ant System – a computationnal study. *Central European Journal for Operations Research and Economics*, Vol. 7 :25–38, 1997.

[BKS97] B. Bullnheimer, G. Kotsis, and C. Strauss. Parallelization strategies for the Ant System. In R. De Leone, A. Murli, P. Pardalos, and G. Toraldo, editors, *High Performance Algorithms and Software in Nonlinear Optimization*, volume 24 of *Applied Optimization*, pages 87–100. Kluwer, Dordrecht, 1997.

[BSARK12] R. Ben-Shalom, A. Aviv, B. Razon, and A. Korngreen. Optimizing ion channel models using a parallel Genetic Algorithm on graphical processors. *Journal of Neuroscience Methods*, 206(2) :183 – 194, 2012.

[BUW10] W. Bozejko, M. Uchronski, and M. Wodecki. Parallel hybrid metaheuristics for the flexible job shop problem. *Computers & Industrial Engineering*, 59(2) : 323–333, 2010.

[CB11] M. Czapinski and S. Barnes. Tabu Search with two approaches to parallel flowshop evaluation on CUDA platform. *J. Parallel Distrib. Comput.*, 71(6) : 802–811, 2011.

[CBFN11] M. Oliveira Junior C. Bastos-Filho and D. Nascimento. *Running Particle Swarm Optimization on Graphic Processing Units*, chapter 2, pages 47–68. InTech, 2011.

[CBM12] S. Cagnoni, A. Bacchini, and L. Mussi. OpenCL implementation of Particle Swarm Optimization : A comparison between multi-core CPU and GPU performances. In *EvoApplications*, pages 406–415. Springer, 2012.

[CBZ10] A. Choong, R. Beidas, and J. Zhu. Parallelizing Simulated Annealing-based placement using GPGPU. In *FPL*, pages 31–34. IEEE, 2010.

[CDJN11] S. Chen, S. Davis, H. Jiang, and A. Novobilski. CUDA-based Genetic Algorithm on Traveling Salesman Problem. In Roger Lee, editor, *Computer and Information Science 2011*, volume 364 of *Studies in Computational Intelligence*, pages 241–252. Springer Berlin / Heidelberg, 2011.

[CDM91] A. Colorni, M. Dorigo, and V. Maniezzo. Distributed optimization by ant colonies. In *Proceedings of the First European Conference on Artificial Life*, pages 134–142. F. Varela and P. Bourgine (Eds.), Elsevier Publishing, Paris, France, December 11-13, 1991.

[cg12] NVIDIA Corporation. *Cg 3.1 Reference Manual Release 3.1*, 2012.

[CGN+12] J.M. Cecilia, J.M. García, A. Nisbet, M. Amos, and M. Ujaldøn. Enhancing data parallelism for Ant Colony Optimization on GPUs. *Journal of Parallel and Distributed Computing*, 2012.

[CGU+11] J.M. Cecilia, J.M. García, M. Ujaldøn, A. Nisbet, and M. Amos. Parallelization strategies for Ant Colony Optimisation on GPUs. *The Computing Research Repository (CoRR)*, abs/1101.2678, 2011.

[Cle10] M. Clerc. *Particle Swarm Optimization*. John Wiley & Sons, 2010.

[CMRC02] V. Cung, S.L. Martins, C.C. Ribeiro, and C.Roucairol. Strategies for the parallel implementation of metaheuristics. In *Essays and Surveys in Metaheuristics*, pages 263–308. Kluwer Academic Publishers, 2002.

[CNA+12] J.M. Cecilia, A. Nisbet, M. Amos, J.M. García, and M. Ujaldón. Enhancing GPU parallelism in nature-inspired algorithms. *The Journal of Supercomputing*, pages 1–17, 2012.

[CP98] E. Cantú-Paz. A survey of parallel Genetic Algorithms. *Calculateurs paralleles*, 10, 1998.

[CR04] M. Craus and L. Rudeanu. Parallel framework for ant-like algorithms. In *Third International Symposium on Parallel and Distributed Computing (ISPDC/HeteroPar'04)*, pages 36–41, 2004.

[Cra05] T. Crainic. *Parallel Computation, Co-operation, Tabu Search*, pages 283–302. Metaheuristic Optimization Via Memory and Evolution : Tabu Search and Scatter Search. Kluwer Academic Publishers, 2005.

[Cro58] G.A. Croes. A method for solving Traveling Salesman Problems. *Operations Research*, 6 :791–812, 1958.

[CT10] T. Crainic and M. Toulouse. Parallel meta-heuristics. In M. Gendreau and J.-Y. Potvin, editors, *Handbook of Metaheuristics*, volume 146 of *International Series in Operations Research & Management Science*, pages 497–541. Springer US, 2010.

[CTZ05] D. Chu, M. Till, and A. Zomaya. Parallel Ant Colony Optimization for 3D protein structure prediction using the HP lattice model. In *19th IEEE international Parallel and Distributed Processing Symposium*, volume 7. IEEE Computer Society, 2005.

[cud11] NVIDIA Corporation. *CUDA : Computer Unified Device Architecture Programming Guide 4.1*, 2011.

[DG97] M. Dorigo and L.M. Gambardella. Ant Colony System : A cooperative learning approach to the Traveling Salesman Problem. *IEEE Transactions on Evolutionary Computation*, 1(1) :53–66, 1997.

[DGK+05a] P. Delisle, M. Gravel, M. Krajecki, C. Gagné, and W. L. Price. Comparing parallelization of an ACO : Message passing vs. shared-memory. In M.J. Blesa, C. Blum, A. Roli, and M. Sampels, editors, *Lecture Notes in Computer Science*, volume 3636, pages 1–11. Springer-Verlag Berlin Heidelberg, 2005.

[DGK+05b] P. Delisle, M. Gravel, M. Krajecki, C. Gagné, and W. L. Price. A shared memory parallel implementation of Ant Colony Optimization. In *6th Metaheuristics International Conference (MIC'2005)*, pages 257–264. Vienna, Autria, 2005.

[DGK09] P. Delisle, M. Gravel, and M. Krajecki. Multi-colony parallel Ant Colony Optimization on SMP and multi-core computers. In *Proceedings of the World Congress on Nature and Biologically Inspired Computing (NaBIC 2009)*, pages 318–323. IEEE, 2009.

[DHBL06] K. Doerner, R. Hartl, S. Benker, and M. Lucka. Parallel cooperative savings based Ant Colony Optimization - multiple search and decomposition approaches. *Parallel Processing Letters*, 16(3) :351–370, 2006.

[DKGG01] P. Delisle, M. Krajecki, M. Gravel, and C. Gagné. Parallel implementation of an Ant Colony Optimization metaheuristic with OpenMP. In *International*

Conference on Parallel Architectures and Compilation Techniques, 3rd European Workshop on OpenMP (EWOMP'01). Barcelona, Spain, 2001.

[DMC91] M. Dorigo, V. Maniezzo, and A. Colorni. Ant System : An autocatalytic optimizing process. Technical Report, n. 91-016 Politecnico di Milano, 1991.

[DMC96] M. Dorigo, V. Maniezzo, and A. Colorni. Ant System : Optimization by a colony of cooperating agents. *IEEE Transactions on Systems, Man, and Cybernetics - Part B : Cybernetics*, 26(1) :29–41, 1996.

[DPST03] J. Dréo, A. Pétrowski, P. Siarry, and E. Taillard. *Métaheuristiques pour l'optimisation difficile*. Edition Eyrolles, 2003.

[Dra10] B. Draskoczy. Fitness distance correlation and search space analysis for permutation based problems. In *EvoCOP*, pages 47–58. Springer, 2010.

[DS04] M. Dorigo and T. Stützle. *Ant Colony Optimization*. Bradford Books. MIT Press, 2004. I-XIV, 1-305 pp.

[ECB07] I. Ellabib, P. Calamai, and O. Basir. Exchange strategies for multiple Ant Colony System. *Information Sciences*, 177(5) :1248–1264, 2007.

[FK12] F. Fabris and R.A. Krohling. A co-evolutionary Differential Evolution algorithm for solving min-max optimization problems implemented on GPU using C-CUDA. *Expert Systems with Applications*, 39(12) :10324 – 10333, 2012.

[Fly66] M.J. Flynn. Very high-speed computing systems. In *Proceedings of the IEEE*, pages 1901–1909. Morgan Kaufmann Publishers Inc., December, 1966.

[FOW66] L.J. Fogel, A.J. Owens, and M.J. Walsh. *Artificial intelligence through simulated evolution*. Wiley, 1966.

[FT10] N. Fujimoto and S. Tsutsui. A highly-parallel TSP solver for a GPU computing platform. In *NMA*, pages 264–271. Springer, 2010.

[FWW07] K. Fok, T.T. Wong, and M.L. Wong. Evolutionary computing on consumer graphics hardware. *IEEE Intelligent Systems*, 22(2) :69–78, 2007.

[FZY10] Y. Feng, L. Zhao, and J. Yang. Tuning schema matching systems using parallel Genetic Algorithms on GPU. *International Journal of Modern Education and Computer Science*, 2(1) :48–56, 2010.

[GL97] F. Glover and M. Laguna. *Tabu Search*. Kluwer Academic Publishers Norwell, Boston, USA, 1997.

[Glo89] F. Glover. Tabu Search - part I. *ORSA Journal on Computing*, 1(3) :190–206, 1989.

[Glo90] F. Glover. Tabu Search - part II. *ORSA Journal on Computing*, 2(1) :4–32, 1990.

[gls06] The Khronos Group. *The OpenGL Shading Language V1.2*, 2006.

[GP02] G. Gutin and A.P. Punnen. *The Traveling Salesman Problem and Its Variations*. Combinatorial Optimization. Kluwer Academic Publishers, 2002.

[Gre08] F. Greco, editor. *Traveling Salesman Problem*. InTech, 2008.

[GZ06] A. Greß and G. Zachmann. GPU-ABiSort : optimal parallel sorting on stream architectures. In *IPDPS*. IEEE, 2006.

[HCRK11] Y. Han, K. Chakraborty, S. Roy, and V. Kuntamukkala. Design and implementation of a throughput-optimized GPU floorplanning algorithm. *ACM Trans. Design Autom. Electr. Syst.*, 16(3) :23, 2011.

[HHL12] C.-S. Huang, Y.-C. Huang, and P.-J. Lai. Modified Genetic Algorithms for solving fuzzy flow shop scheduling problems and their implementation with CUDA. *Expert Syst. Appl.*, 39(5) :4999–5005, 2012.

[HKM97] I. Hong, A. Kahng, and B. Moon. Improved large-step markov chain variants for the symmetric TSP. *Journal of Heuristics*, 3 :63–81, September 1997.

[Hol75] J.H. Holland. *Adaptation in Natural and Artificial Systems.* The University of Michigan Press, 1975.

[HS04] H. Hoos and T. Stützle. *Stochastic Local Search Foundations and Applications.* Morgan Kaufmann, Elsevier, 2004.

[ITT03] M. T. Islam, P. Thulasiraman, and R. K. Thulasiram. A parallel Ant Colony Optimization algorithm for all-pair routing in MANETs. In *17th international Symposium on Parallel and Distributed Processing.* IEEE Computer Society, 2003.

[JJL08] A. Janiak, W.A. Janiak, and M. Lichtenstein. Tabu search on GPU. *J. UCS*, 14(14) :2416–2426, 2008.

[JL07] P. Jetté and M. Lalonde. *Calculs Généraux Sur Processeurs Graphiques, GPGPU : Première Version.* Collection Scientifique et Technique. Centre de Recherche Informatique de Montréal, 2007.

[JM97] D.S. Johnson and L.A. McGeoch. *The Travelling Salesman Problem : A Case Study in Local Optimization*, pages 215–310. E.H.L. Aarts and J.K. Lenstra, editors, Local Search in Combinatorial Optimization. John Wiley & Sons, 1997.

[JP12] J. Jaros and P. Pospichal. A fair comparison of modern CPUs and GPUs running the Genetic Algorithm under the knapsack benchmark. In *EvoApplications*, pages 426–435. Springer, 2012.

[KE95] J. Kennedy and R. Eberhart. Particle Swarm Optimization. In *IEEE International Conference on Neural Networks*, volume 4, pages 1942–1948. IEEE, 1995.

[KGV83] S. Kirkpatrick, C.D. Gelatt, and M.P. Vecchi. Optimization by Simulated Annealing. *Science*, 220(4598) :671–680, 1983.

[KL70] B.W. Kernighan and S. Lin. An effective heuristic procedure for partitioning graphs. *The Bell System Technical Journal*, 49 :291–307, 1970.

[KPSA11] P. Krömer, J. Platos, V. Snásel, and A. Abraham. A comparison of many-threaded Differential Evolution and Genetic Algorithms on CUDA. In *NaBIC*, pages 509–514. IEEE, 2011.

[Lap92] G. Laporte. The Traveling Salesman Problem : An overview of exact and approximate algorithms. *European Journal of Operational Research*, (59) :231–247, 1992.

[LHPQ09] J. Li, X. Hu, Z. Pang, and K. Qian. A parallel Ant Colony Optimization algorithm based on fine-grained model with GPU-acceleration. *International Journal of Innovative Computing, Information and Control*, 5(11(A)) :3707–3716, 2009.

[Lin65] S. Lin. Computer solutions of the Traveling Salesman Problem. *Bell System Technical Journal*, 44 :2245–2269, 1965.

[LK73] S. Lin and B.W. Kernighan. An effective heuristic algorithm for the Travelling Salesman Problem. *Operations Research*, 21 :498–516, 1973.

[LLMT10] T.V. Luong, L. Loukil, N. Melab, and E. Talbi. A GPU-based Iterated Tabu Search for solving the quadratic 3-dimensional assignment problem. In *AICCSA*, pages 1–8. IEEE, 2010.

[LLTS12] C. Leung, P. Lam, P.W. Tsang, and W. Situ. A Graphics Processing Unit accelerated Genetic Algorithm for affine invariant matching of broken contours. *Signal Processing Systems*, 66(2) :105–111, 2012.

[LMS10] H. Lourenço, O. Martin, and T. Stützle. *Iterated Local Search : framework and applications*, pages 363–397. Handbook of metaheuristics. Springer, 2010.

[LMT10a] T.V. Luong, N. Melab, and E. Talbi. GPU-based island model for Evolutionary Algorithms. In *GECCO*, pages 1089–1096. ACM, 2010.

[LMT10b] T.V. Luong, N. Melab, and E. Talbi. Neighborhood structures for GPU-based Local Search algorithms. *Parallel Processing Letters*, 20(4) :307–324, 2010.

[LMT10c] T.V. Luong, N. Melab, and E. Talbi. Parallel hybrid Evolutionary Algorithms on GPU. In *IEEE Congress on Evolutionary Computation*, pages 1–8. IEEE, 2010.

[LMT11a] T.V. Luong, N. Melab, and E. Talbi. GPU-based multi-start Local Search algorithms. In *LION*, Lecture Notes in Computer Science, pages 321–335. Springer, 2011.

[LMT11b] T.V. Luong, N. Melab, and E. Talbi. GPU computing for parallel Local Search metaheuristic algorithms. *IEEE Transactions on Computers*, 99(PrePrints), 2011.

[LSOCCC+10] G.A. Laguna-Sánchez, M. Olguín-Carbajal, N. Cruz-Cortés, R. Barrón-Fernández, and J.A. Alvarez-Cedillo. Comparative study of parallel variants for a Particle Swarm Optimization. *Journal of Applied Research and Technology*, 7(3) :292–309, 2010.

[LTLL10] J. Li, H. Tan, X. Li, and L. Liu. A parallel Simulated Annealing solution for VRPTW based on GPU acceleration. In *KES IDT*, pages 201–208. Springer, 2010.

[Luo11] T.V. Luong. *Métaheuristiques parallèles sur GPU*. PhD thesis, Université de Lille, France, 2011.

[MBSD06] M. Manfrin, M. Birattari, T. Stützle, and M. Dorigo. Parallel Ant Colony Optimization for the Traveling Salesman Problem. In *Lecture Notes in Computer Science*, volume 4150, pages 224–234, 2006.

[MCC+10] L. Mussi, S. Cagnoni, E. Cardarelli, F. Daolio, P. Medici, and P.P. Porta. GPU implementation of a road sign detector based on Particle Swarm Optimization. *Evolutionary Intelligence*, 3(3-4) :155–169, 2010.

[MCD09] L. Mussi, S. Cagnoni, and F. Daolio. GPU-based road sign detection using Particle Swarm Optimization. In *ISDA*, pages 152–157. IEEE Computer Society, 2009.

[MDC11] L. Mussi, F. Daolio, and S. Cagnoni. Evaluation of parallel Particle Swarm Optimization algorithms within the CUDA architecture. *Inf. Sci.*, 181(20) : 4642–4657, 2011.

[MKQ+12] O. Maitre, F. Krüger, S. Querry, N. Lachiche, and P. Collet. EASEA : specification and execution of Evolutionary Algorithms on GPGPU. *Soft Comput.*, 16(2) :261–279, 2012.

[MLBT11] N. Melab, T.V. Luong, K. Boufaras, and E. Talbi. Towards ParadisEO-MO-GPU : A framework for GPU-based Local Search metaheuristics. In *IWANN (1)*, pages 401–408. Springer, 2011.

[MN92] P. Moscato and M.G. Norman. A "memetic" approach for the Traveling Salesman Problem implementation of a computational ecology for combinatorial optimization on message-passing systems. In *In Proceedings of the International Conference on Parallel Computing and Transputer Applications*, pages 177–186. IOS Press, 1992.

[MNC11] L. Mussi, Y.S.G. Nashed, and S. Cagnoni. GPU-based asynchronous Particle Swarm Optimization. In *GECCO*, pages 1555–1562. ACM, 2011.

[MO96] O. Martin and S. Otto. Combining Simulated Annealing with Local Search heuristics. *Annals of Operations Research*, 63 :57–75, 1996.

[MRR+53] N. Metropolis, A.W. Rosenbluth, M.N. Rosenbluth, A.H. Teller, and E. Teller. Equation of state calculation by fast computing machines. *Journal of Chemical Physics*, 21(6) : 1087–1092, 1953.

[MRS02] M. Middendorf, F. Reischle, and H. Schmeck. Multi colony ant algorithms. *Journal of Heuristics*, 8(3) :305–320, 2002.

[MWMA09] A. Munawar, M. Wahib, M. Munetomo, and K. Akama. Hybrid of Genetic Algorithm and Local Search to solve MAX-SAT problem using NVIDIA CUDA framework. *Genetic Programming and Evolvable Machines*, 10(4) :391–415, 2009.

[nvi10] NVIDIA Corporation. *TESLA C2050 / C2070 GPU Computing Processor*, 2010.

[nvi12] NVIDIA Corporation. *NVIDIA Corporation from super phones to super cars*, 2012.

[OL96] I.H. Osman and G. Laporte. Metaheuristics : A bibliography. *Annals of Operations Research*, 63(5) :511–623, 1996.

[OMYO11] M. Oiso, Y. Matsumura, T. Yasuda, and K. Ohkura. Implementing Genetic Algorithms to CUDA environment using data parallelization. *Technical Gazette*, 18(4) :511–517, 2011.

[ope12] Khronos Group. *OpenCL API 1.2 Reference Card*, 2012.

[OTB11] M.A. O'Neil, D. Tamir, and M. Burtscher. A parallel GPU version of the Traveling Salesman Problem. In *International Conference on Parallel and Distributed Processing Techniques and Applications*, pages 348–353, 2011.

[Pac11] P. Pacheco. *An Introduction to Parallel Programming.* Morgan Kaufmann. Elsevier Science, 2011.

[PAL11] M. Pedemonte, E. Alba, and F. Luna. Bitwise operations for GPU implementation of Genetic Algorithms. In *GECCO (Companion)*, pages 439–446. ACM, 2011.

[PDB12] F. Pinel, B. Dorronsoro, and P. Bouvry. Solving very large instances of the scheduling of independent tasks problem on the GPU. *Journal of Parallel and Distributed Computing*, 2012.

[PJS10] P. Pospichal, J. Jaros, and J. Schwarz. Parallel Genetic Algorithm on the CUDA architecture. In *EvoApplications (1)*, pages 442–451. Springer, 2010.

[PNC11] M. Pedemonte, S. Nesmachnow, and H. Cancela. A survey on parallel Ant Colony Optimization. *Appl. Soft Comput.*, 11(8) :5181–5197, 2011.

[PSJ10] P. Pospichal, J. Schwarz, and J. Jaros. Parallel Genetic Algorithm solving 0/1 knapsack problem running on the GPU. In *16th International Conference on Soft Computing MENDEL 2010*, pages 64–70. Brno University of Technology, 2010.

[RCCRT11] L.E. Ramírez-Chavez, C.A. Coello Coello, and E. Rodríguez-Tello. A GPU-based implementation of differential evolution for solving the gene regulatory network model inference problem. In *Proceedings of the Fourth International Workshop on Parallel Architectures and Bioinspired Algorithms, WPABA 2011*, pages 21–30, 2011.

[Rec65] I. Rechenberg. Cybernetic solution path of an experimental problem, 1965.

[Rei94] G. Reinelt. *The Traveling Salesman, Computational Solutions for TSP Applications*, volume 840 of *Lecture Notes in Computer Science*. Springer, 1994.

[RL02] M. Randall and A. Lewis. A parallel implementation of Ant Colony Optimization. *Journal of Parallel and Distributed Computing*, 62(2) :1421–1432, 2002.

[rom11] *Centre de Calcul de Champagne-Ardenne ROMEO*, 2011. URL `https://romeo.univ-reims.fr/`.

[SBPE10] N. Soca, J.L. Blengio, M. Pedemonte, and P. Ezzatti. PUGACE, a cellular Evolutionary Algorithm framework on GPUs. In *IEEE Congress on Evolutionary Computation (CEC)*, pages 1–8. IEEE, 2010.

[SC10] D. Sharma and P. Collet. GPGPU-compatible archive based stochastic ranking Evolutionary Algorithm (G-ASREA) for multi-objective optimization. In *PPSN (2)*, pages 111–120. Springer, 2010.

[Sch12] Christian Schulz. Efficient local search on the GPU-investigations on the vehicle routing problem. *Journal of Parallel and Distributed Computing*, 2012.

[SH97] T. Stützle and H. Hoos. Improvements of the Ant System : Introducing the MAX-MIN Ant System. In *Proceedings ICANNGA97 - Third Int. Conf. Artificial Neural Networks and Genetic Algorithms*. Springer-Verlag, 1997.

[SH00] T. Stützle and H. Hoos. MAX-MIN Ant System. *Future Generation Comp. Syst.*, 16(8) :889–914, 2000.

[SH01] T. Stützle and H. Hoos. *Analysing the run-time behaviour of Iterated Local Search for the Traveling Salesman Problem*, pages 21–43. Essays and surveys in metaheuristics. Springer, 2001.

[SP97] R. Storn and K. Price. Differential Evolution - a simple and efficient heuristic for global optimization over continuous spaces. *Journal of Global Optimization*, 11 :341–359, 1997.

[SSW10] A. Stivala, P.J. Stuckey, and A. Wirth. Fast and accurate protein substructure searching with Simulated Annealing and GPUs. *BMC Bioinformatics*, 11 :446, 2010.

[STT11] S. Solomon, P. Thulasiraman, and R.K. Thulasiram. Collaborative multi-swarm PSO for task matching using Graphics Processing Units. In *GECCO*, pages 1563–1570. ACM, 2011.

[Stü98a] T. Stützle. *Local Search Algorithms for Combinatorial Problems - Analysis, Improvements, and New Applications*. PhD thesis, Darmstadt University of Technology, Darmstadt, Germany, 1998.

[Stü98b] T. Stützle. Parallelisation strategies for Ant Colony Optimization. In A.E. Eiben, T. Bäck, H.-P. Schwefel, and M. Schoenauer, editors, *Proceedings of the Fifth International Conference on Parallel Problem Solving from Nature (PPSN V)*. Springer-Verlag, New York, 1998.

[Tal09] E. Talbi. *Metaheuristics - From Design to Implementation*. Wiley, 2009.

[TF09] S. Tsutsui and N. Fujimoto. Solving quadratic assignment problems by Genetic Algorithms with GPU computation : a case study. In *GECCO (Companion)*, pages 2523–2530. ACM, 2009.

[TF11] S. Tsutsui and N. Fujimoto. ACO with Tabu Search on a GPU for solving QAPs using move-cost adjusted thread assignment. In *GECCO*, pages 1547–1554. ACM, 2011.

[TRFR01] E. Talbi, O. Roux, C. Fonlupt, and D. Robillard. Parallel Ant Colonies for the quadratic assignment problem. *Future Generation Computer Systems*, 17(4) : 441–449, 2001.

[Tsu06] S. Tsutsui. *c*AS : Ant Colony Optimization with cunning ants. In *PPSN*, pages 162–171. Springer, 2006.

[VA95] M.G.A. Verhoeven and E.H.L. Aarts. Parallel Local Search. *Journal of Heuristics*, 1 :43–65, 1995.

[VA10] P. Vidal and E. Alba. A multi-GPU implementation of a Cellular Genetic Algorithm. In *IEEE Congress on Evolutionary Computation*, pages 1–7. IEEE, 2010.

[Wei11] R. M. Weiss. *GPU-Accelerated Ant Colony Optimization*, pages 325–340. Morgan Kaufmann, 2011.

[WMMA11] M. Wahib, A. Munawar, M. Munetomo, and K. Akama. Optimization of parallel Genetic Algorithms for NVIDIA GPUs. In *IEEE Congress on Evolutionary Computation*, pages 803–811. IEEE, 2011.

[WW09] M.L. Wong and T.T. Wong. *Implementation of Parallel Genetic Algorithms on Graphics Processing Units*, volume 187, pages 197–216. Springer Berlin / Heidelberg, 2009.

[WWW11] J. Wang, Z. Wu, and H. Wang. A novel Particle Swarm algorithm for solving parameter identification problems on graphics hardware. *Int. J. Comput. Sci. Eng.*, 6(1/2) :43–51, July 2011.

[Yan90] M. Yannakakis. The analysis of Local Search problems and their heuristics. In *STACS*, pages 298–311. Springer, 1990.

[YCP05] Q. Yu, C. Chen, and Z. Pan. Parallel Genetic Algorithms on programmable graphics hardware. In *ICNC (3)*, pages 1051–1059. Springer, 2005.

[ZC09] W. Zhu and J. Curry. Parallel Ant Colony for nonlinear function optimization with graphics hardware acceleration. In *Proceedings of the 2009 IEEE international conference on Systems, Man and Cybernetics*, pages 1803–1808. IEEE Press, 2009.

[Zhu11] W. Zhu. Nonlinear optimization with a massively parallel Evolution Strategy-pattern search algorithm on graphics hardware. *Appl. Soft Comput.*, 11(2) : 1770–1781, 2011.

[ZLG+09] C. Zhang, H. Li, Q. Guo, J. Jia, and I. Shen. Fast active Tabu Search and its application to image retrieval. In *IJCAI*, pages 1333–1338, 2009.

[ZT09] Y. Zhou and Y. Tan. GPU-based parallel Particle Swarm Optimization. In *IEEE Congress on Evolutionary Computation*, pages 1493–1500. IEEE, 2009.

[ZT10] Y. Zhou and Y. Tan. Particle Swarm Optimization with triggered mutation and its implementation based on GPU. In *GECCO*, pages 1–8. ACM, 2010.