# Parallélisation d'heuristiques d'optimisation sur les GPUs

Achraf Berrajaa

Normandie Université

# THESE

**Pour obtenir le diplôme de doctorat**

**Spécialité : Informatique**

**Préparée au sein de l'université du Havre en cotutelle avec l'université d'Oujda.**

# Parallélisation d'heuristiques d'optimisation sur les GPUs.


**Présentée et soutenue par**
**Achraf BERRAJAA**


| Thèse soutenue publiquement le 29/12/2018 devant le jury composé de | | |
|---|---|---|
| M. El Miloud Jaara | Professeur à l'université Mohammed Premier d'Oujda. | Président |
| M. Ahmed Elhilali Alaoui | Professeur à l'université Sidi Mohamed ben Abdellah de Fès. | Rapporteur |
| M. Abderrahman El Mhamedi | Professeur à l'université de Vincennes Paris 8. | Rapporteur |
| Mme. Souad El Bernoussi | Professeur à l'université Mohammed V de Rabat. | Examinateur |
| M. Ahmed Rachid | Professeur à l'université de Picardie d'Amiens. | Examinateur |
| M. Jaouad Boukachour | MC à l'université Le Havre Normandie. | Examinateur |
| M. El Mostafa Daoudi | Professeur à l'université Mohammed Premier d'Oujda. | Directeur de thèse |
| M. Abdelhamid Benaini | Professeur à l'université Le Havre Normandie. | Directeur de thèse |

**Thèse dirigée par M. Abdelhamid BENAINI et M. El Mostafa DAOUDI, laboratoire LMAH et LARI.**

UNIVERSITÉ LE HAVRE NORMANDIE

MIS

LMAH

# Remerciement

Le travail présenté dans cette thèse n'aurait été possible sans l'intervention et la contribution de plusieurs personnes. Qu'elles trouvent ici l'expression de ma profonde gratitude et de ma reconnaissance la plus sincère.

Mes remerciements vont en premier lieu à mes parents qui sont et ont toujours été tout pour moi. Sans eux, rien n'aurait été possible. Je leur remercie à travers cette thèse pour leur amour inconditionnel, leur soutien constant, leur attention sans faille et leurs encouragements qui m'accompagnent depuis toujours.

Au terme de ce travail, J'aimerais aussi particulièrement remercier mes deux directeurs de thèse pour la confiance qu'ils ont faite à mon égard : M. Abdelhamid BENAINI, pour la qualité de son encadrement, sa disponibilité permanente, ses suggestions, son regard critique et sa relecture attentive et M. El Mostafa DAOUDI, pour son encadrement, ses conseils pertinents et ses qualités humaines et scientifiques ;

Mes remerciements les plus vifs s'adressent également à ceux qui m'ont fait l'honneur de faire partie du jury de ma thèse :

- Monsieur Ahmed EL HILALI ALAOUI, Professeur à l'université Sidi Mohamed ben Abdellah de Fès ;
- Monsieur Abderrahman EL MHAMEDI, Professeur à l'université de Vincennes Paris 8 ;
- Monsieur El Miloud JAARA, Professeur à l'université Mohammed Premier d'Oujda ;
- Madame Souad EL BERNOUSSI, Professeur à l'université Mohammed V de Rabat ;
- Monsieur Ahmed RACHID, Professeur à l'université de Picardie à Amiens ;
- Monsieur Jaouad BOUKACHOUR, Maître de conférences à l'université du Havre.

Qu'il me soit enfin permis de remercier toute ma famille, mes amis et une pensée spéciale à tous ceux qui me sont chers et qui ne sont plus parmi nous.

# Parallélisation d'heuristiques d'optimisation sur les GPUs.

***Résumé.*** — Cette thèse, présente des contributions à la résolution (sur les GPUs) de problèmes d'optimisations réels de grandes tailles. Les problèmes de tournées de véhicules (VRP) et ceux de localisation des hubs (HLP) sont traités. Diverses approches et leur implémentions sur GPU pour résoudre des variantes du VRP sont présentées. Un algorithme génétique (GA) parallèle sur GPU est proposé pour résoudre différentes variantes du HLP. Le GA adapte son codage, sa solution initiale, ses opérateurs génétiques et son implémentation à chacune des variantes traitées. Enfin, nous avons utilisé le GA pour résoudre le HLP avec des incertitudes sur les données.
Les tests numériques montrent que les approches proposées exploitent efficacement la puissance de calcul du GPU et ont permis de résoudre de larges instances jusqu'à 6000 nœuds.

***Mots clés :*** GPU, problèmes d'optimisations, tournées de véhicules (VRP), localisation des hubs (HLP) et algorithme génétique.

***Abstract.*** — This thesis presents contributions to the resolution (on GPUs) of real optimization problems of large sizes. The vehicle routing problems (VRP) and the hub location problems (HLP) are treated. Various approaches implemented on GPU to solve variants of the VRP. A parallel genetic algorithm (GA) on GPU is proposed to solve different variants of the HLP. The proposed GA adapts its encoding, initial solution, genetic operators and its implementation to each of the variants treated. Finally, we used the GA to solve the HLP with uncertainties on the data.
The numerical tests show that the proposed approaches effectively exploit the computing power of the GPU and have made it possible to resolve large instances up to 6000 nodes.

***Keywords :*** GPU, optimization problems, vehicle routing (VRP), hub location (HLP) and genetic algorithm.

# Table des matières

# Introduction générale

Dans un passé non lointain, le calcul parallèle était considéré comme une poursuite exotique et se classait généralement comme une spécialité dans le domaine de l'informatique. Cette perception a profondément changé ces dernières années et a touché plusieurs domaines, notamment les télécommunications, l'industrie, le transport et la médecine. Ces derniers ont plutôt opté pour la puissance de calcul parallèle et notamment l'utilisation de nouvelles plate-formes de calculs basées sur les accélérateurs GPUs.

Depuis le lancement des GPUs au début de 2007, diverses applications industrielles ont tiré profit de ces accélérateurs ; les avantages se mesurent surtout en termes de performances en comparaison avec les performances des implémentations traditionnelles construites exclusivement sur les technologies mono ou multi CPU(s).

En effet, la logistique de transport est un exemple d'optimisation où l'usage des GPUs est intéressant. Les problèmes de la logistique de transport, de localisation des hubs ont été démontrer comme des problèmes NP-difficiles.
Certes, les heuristiques et les métaheuristiques sont utilisées pour résoudre de tels problèmes, mais elles restent insuffisantes pour trouver rapidement une bonne solution aux problèmes de grandes tailles.
Cependant, l'utilisation d'une méthode de résolution efficace sur les unités de traitement graphique moderne GPU peut atteindre cet objectif, comme le confirment les résultats présentés dans cette thèse.

Dans ce travail, nous avons essayé d'exploiter la puissance des GPUs pour résoudre efficacement des problèmes d'optimisations réels et volumineux. Les deux problèmes traités sont : les problèmes de tournées de véhicules (VRP) et ceux de localisation des hubs (HLP).

Dès qu'il s'agit d'applications réelles, la taille des problèmes devient considérable (quelques millions de points de distributions par exemple). Dans ce cas, les problèmes ne peuvent pas être résolus à l'aide de systèmes classiques (CPLEX par exemple). Nous sommes parmi les premiers à essayer de pallier cela par, l'usage des GPUs. Concrètement, nous avons proposé des approches parallèles et les avons implémenté sur GPU pour les deux problèmes cités précédemment et leurs variantes.

Quatre implémentations sur GPU sont proposées pour les variantes du VRP, la première se base sur l'heuristique de Clarke et Wright, pour résoudre le VRP classique [24]. Une version améliorée de cette implémentation a fait l'objet d'une autre publication mais cette fois-ci pour la variante multi dépôts [23].
La deuxième implémentation a concerné, la variante multi capacités VRP. Cette dernière, nous permet d'améliorer les solutions trouvées dans la littérature et à résoudre des instances non résolues [25].
La troisième implémentation [16] concerne la variante du VRP dynamique. Elle se base sur une heuristique simple qui insère en temps réel des requêtes dynamiques dans des routes déjà planifiées.
La quatrième implémentation [17] est fondée sur l'algorithme génétique, pour la résolution de grandes instances du VRP dynamique (jusqu'à 3000 clients).

Le deuxième problème traité est celui de la localisation des hubs (HLP), auquel nous avons proposé divers algorithmes génétiques chacun dédié à une variante du HLP [18, 19, 21, 22]. Chaque GA proposé adapte son codage, sa solution initiale, ses opérateurs génétiques et son implémentation à la variante HLP traitée. Pour les cinq variantes du HLP traitées ; la performance des implémentations est comparée aux meilleures solutions connues sur tous les benchmarks, sur des instances jusqu'à 1000 nœuds et sur des instances plus importantes jusqu'à 6000 nœuds générées par nous-même.

Enfin, nous nous sommes intéressés aux HLP avec des données incertaines (le flux transporté dans le réseau et les données du problème sont incertaines). Nous avons adapté le GA dédié au problème de localisation de p-médian hub avec allocation unique pour résoudre le problème des incertitude sur les demandes [20].
La fonction objectif minimise les coûts en se basant sur le modèle *min-max* dans chaque scénario. Notre implémentation sur GPU a permis de calculer des solutions efficaces même pour des instances larges (jusqu'à 6000 nœuds).

Le travail se compose de cinq chapitres :

Le chapitre 1, décrit le contexte général des problèmes traités. En effet, nous présentons une revue de la littérature sur les divers thèmes reliés à notre problématique. Il s'agit non seulement de présenter les approches et les méthodes de résolution les plus réputées pour résoudre les problèmes d'optimisation combinatoire, mais également de rappeler et de donner les concepts de base nécessaires à la compréhension du problème de tournées de véhicules, et celui de localisation des hubs avec ses variantes. Il s'agit enfin de présenter et de décrire la problématique de notre recherche.

Le chapitre 2, commence par présenter l'historique de l'évolution des unités de traitement CPUs et des GPUs pour les ordinateurs personnels. La suite est réservée à l'explication du principe, de la performance et de l'architecture des GPUs, puis à la description de l'architecture CUDA. Une conclusion est consacrée au final à passer en revue quelques domaines et applications ayant connu un succès retentissant en choisissant d'utiliser les GPUs et la technologie CUDA.

Le chapitre 3, présente les quatre implémentations parallèles proposées sur le GPU, pour résoudre les variantes du VRP. Chaque implémentation est présentée dans une section comme suit :
La première section présente l'implémentation pour résoudre le VRP classique unique et multi dépôts. La deuxième section est réservée à l'implémentation pour résoudre le multi capacités VRP. La troisième section présente, la conception et l'implémentation de la heuristique qui insère des requêtes dynamiques avec une ré-optimisation continue. La quatrième section a concerné l'algorithme génétique, pour résoudre le DVRP avec une ré-optimisation périodique.

Le chapitre 4, présente la résolution des problèmes de localisation des hubs. L'algorithme génétique a été développé avec différents codages, opérateurs génétiques et implémentations pour qu'il soit adaptable aux différentes variantes de ce problème. Cinq variantes du problème de localisation des hubs sont présentées dans ce chapitre. Ce dernier se décline en trois sections :
La première section présente l'implémentation pour résoudre le problème où le nombre des hubs à localiser est connu à l'avance et où, le nombre est une décision à prendre. La deuxième sections est réservée à l'implémentation de la variante où les hubs ont une capacité limitée et finalement la troisième section est consacrée à la variante multi affectations.

Au cours des dernières années et dans le cadre d'un processus de prise de décisions stratégiques, les problèmes d'optimisation combinatoire ont été étendus pour

gérer des données incertaines, ce qui donne une optimisation robuste.

Le chapitre 5, présente un contexte général de l'optimisation robuste. Nous détaillons par la suite la résolution du problème de localisation des hubs avec des demandes incertaines, en présentant l'implémentation parallèle sur GPU et les différents résultats obtenus pour ce problème.

Enfin, la thèse se conclut par les perspectives de nos recherches.

Chapitre 1

# Concepts de base pour les problèmes de transport et de localisation

## 1.1 Introduction

En général, un problème d'optimisation combinatoire consiste à trouver un point extrême d'une fonction à maximiser ou à minimiser, souvent appelée fonction objectif, sur un ensemble discret. Cette solution est alors appelée optimale. Trouver une solution optimale dans un ensemble discret et fini est une question facile en théorie. Nous devons juste essayer toutes les solutions et comparer leurs qualités pour voir la meilleure. Cependant, dans la pratique, la comparaison de toutes les solutions peut prendre beaucoup de temps qui est un facteur très important pour chercher la solution optimale et c'est pourquoi les problèmes d'optimisation combinatoire sont considérés comme des problèmes difficiles. En plus, comme l'ensemble des solutions réalisables est défini de manière implicite, il est parfois très difficile de trouver une solution réalisable.

Le système de distribution est un problème d'optimisation combinatoire, qui consiste à identifier et optimiser le coût de transport depuis le centre de fabrication jusqu'aux clients. Le problème de base de ce système de transport est connu sous le nom de "problème de routage" dans le domaine de la recherche opérationnelle. Dans les problèmes de routage, les clients peuvent être décrits par différentes caractéristiques, telles que : leurs demandes (marchandises, quantité, livraison et / ou enlèvement), leurs localisations dans le réseau, le service horaire, leurs disponibilités représentées par la fenêtre d'horaire, et le type de véhicules nécessaires pour satisfaire les demandes des clients. Un problème de routage est souvent modélisé par un graphe, où les nœuds représentent un ensemble de clients, les arcs représentent les

liens entre chaque paire de nœuds et parfois le nombre de véhicules disponibles au dépôt est limité. Un véhicule est exploité dans chaque tournée, commençant et se terminant au même nœud nommé dépôt, en satisfaisant un ensemble de demandes du client. Nous notons que chaque client doit être servi uniquement par un véhicule. L'objectif est de créer les tournées avec un coût minimal. Le problème du voyageur de commerce (TSP) et le problème de tournées de véhicules (VRP) sont les deux types de problèmes de routage les plus importants. Si la capacité du véhicule permet de visiter tous les clients, une seule "grosse" tournée est requise. Ce problème est connu sous le nom de problème du voyageur de commerce (TSP). Autrement dit, lorsqu'un véhicule n'est pas suffisant pour desservir l'ensemble des clients, plusieurs tournées doivent être effectuées. Le problème résultant est connu sous le nom de problème de tournées de véhicules (VRP).

D'autre part parmi les problèmes optimisation combinatoire les plus importants ce sont les problèmes de localisation des hubs (HLPs) ; des problèmes qui surviennent dans les réseaux de télécommunications et de transport où les nœuds envoient et reçoivent des marchandises (transmissions de données, passagers, courriers, etc.) via des installations spéciales ou des points de transbordement s'appellent des hubs. Les hubs "concentrateurs" consolident les flux à partir des nœuds d'origine et les redirigent vers des nœuds de destination, parfois via d'autres hubs. Les nœuds d'envoi et de réception dans de tels réseaux sont appelés non-hub (spokes en anglais). L'objectif est de localiser les hubs et d'affecter les non-hubs aux hubs de sorte que le coût de transport total soit minimisé.

Ce chapitre présente les concepts de base liés aux problèmes de tournées de véhicules et de localisation des hubs et leurs positions dans les problèmes de la logistique. Dans la section 1.3 nous présentons les méthodes les plus courantes, utilisées dans la littérature pour résoudre les problèmes d'optimisation combinatoire. Une définition plus formelle des problèmes de tournées de véhicules (VRP) et du problème de localisation des hubs (HLP) et des modèles classiques pertinents sont présentés respectivement dans les sections 1.4 et 1.5. Ces sections présentent également les variantes du VRP et du HLP avec leurs méthodes de résolution. La section 1.6 conclut ce chapitre.

## 1.2 Concepts de base

### 1.2.1 Logistique en générale

La logistique est une notion qui vient d'un mot grec "logistike" signifiant l'art du raisonnement et du calcul. Historiquement, la logistique trouve ses origines dans le domaine militaire où elle définit l'ensemble des techniques nécessaires aux activités de soutien (réapprovisionnement en armes, munitions, uniformes, chaussures, etc) et aux opérations militaires. Par la suite, cette notion s'est vue appropriée par le milieu industriel notamment avec la reconversion des spécialistes militaires en logistique dans les entreprises industrielles. Depuis, la logistique a évolué avec l'évolution du marché et des systèmes industriels. Actuellement, elle recouvre diverses fonctions et activités à savoir l'achat, l'approvisionnement, la production, la gestion des stocks, le transport et la distribution. Une telle diversité a été à l'origine d'un désaccord total entre les spécialistes sur la définition du concept "logistique", ce qui a donné lieu à une multitude de tentatives de définitions. Parmi celles-ci, nous retenons la définition officielle de la norme AFNOR (norme X 50-600). La logistique est une fonction *"dont la finalité est la satisfaction des besoins exprimés ou latents, aux meilleures conditions économiques pour l'entreprise et pour un niveau de service déterminé. Les besoins sont de nature interne (approvisionnement de biens et de services pour assurer le fonctionnement de l'entreprise) ou externe (satisfaction des clients). La logistique fait appel à plusieurs métiers et savoir-faire qui concourent à la gestion et à la maîtrise des flux physiques et d'informations ainsi que des moyens".*

Bien qu'il existe une assez vaste littérature sur les types de la logistique, nous n'en ferons ici qu'un bref survol de ceux qui sont très présents dans la littérature :

- *Logistique d'approvisionnement* : achat des matières premières, sélection de fournisseurs.
- *Logistique de production* : planification de la production. Elle est appelée également logistique interne.
- *Logistique de soutien* : organisation de tout ce qui est nécessaire pour rendre opérationnel un système industriel donné. Nous citons, par exemple, la prévision et l'entretien.
- *Logistique de distribution* : transport et acheminement.
- *Logistique inverse* : traitement des retours des produits, recyclage.
- *Logistique de service* : système bancaire, télécommunication.

L'objectif commun à toutes ces logistiques est d'atteindre une haute performance en optimisant leurs ressources et en réduisant leurs coûts.

Dans cette thèse, on va se focaliser sur la logistique de distribution dans laquelle on va traiter les deux problèmes parmi les plus importants dans ce domaine, le problème de tournées de véhicules et de localisation des hubs. Ces deux problèmes font partie d'une classe de problèmes de recherche opérationnelle et d'optimisation combinatoire NP-difficile.

### 1.2.2 Les problèmes NP-difficiles

La théorie de la complexité fournit une classification des problèmes selon leur difficulté. Les deux principales familles de problèmes sont la classe des problèmes P pour lesquels il existe un algorithme de complexité polynomiale permettant de résoudre tels problèmes ; tandis que la classe des problèmes NP vérifie en temps polynomial si la solution proposée si elle est réalisable ou pas.

La distinction entre les deux classes n'est aujourd'hui pas prouvée et se ramène au problème ouvert P = NP. L'opinion la plus courante chez les chercheurs en recherche opérationnelle est que P $\neq$ NP. Sous cette condition, on peut penser qu'il est peine perdue de vouloir résoudre des problèmes NP-difficiles de manière efficace. Bien qu'il n'existe pas de méthodes rapides pour les résoudre de manière exacte, il existe cependant des méthodes alternatives pour traiter ces problèmes.

D'une part, il y a les méthodes de résolution exacte, mais dont la durée d'exécution peut être exponentielle.

D'autre part, il existe des méthodes plus rapides, mais dont le résultat n'a pas la garantie d'être optimum, voire peut être parfois arbitrairement mauvais. La conception de méthodes de résolution est généralement un compromis entre le temps de calcul et la qualité des solutions. Se pose alors le problème de comparer les méthodes de résolution, selon leur qualité des solutions et selon le temps de calcul afin de résoudre le problème.

Les chercheurs différencient deux types d'approches selon leurs aspects théoriques ou empiriques. Les études théoriques permettent de calculer les bornes sur les performances des méthodes de résolution. Elles peuvent porter sur la qualité des solutions avec les algorithmes d'approximation à facteur constant, ou sur le temps d'exécution avec la complexité des algorithmes. Les études empiriques permettent d'évaluer le comportement pratique des méthodes de résolution.

La section suivante présente quelques approches utilisées pour résoudre ce type de problèmes d'optimisation combinatoire NP-difficiles.

## 1.3 Méthodes d'optimisation combinatoire

Dans la forme la plus générale, un problème d'optimisation combinatoire (également appelé optimisation discrète) consiste à trouver un ensemble discret optimal parmi l'un des meilleurs sous-ensemble (ou solutions) possibles. Le concept de meilleure solution est défini par une fonction objectif. Les problèmes d'optimisation combinatoire peuvent être considérés comme la recherche du meilleur élément d'un ensemble d'éléments discrets ; par conséquent, n'importe quel algorithme de recherche ou métaheuristique peut être utilisé pour les résoudre. Cependant, les algorithmes de recherche génériques ne peuvent garantir une solution optimale ni un fonctionnement rapide (en temps polynomial). Dans la suite, nous allons expliquer les méthodes de résolution les plus populaires.

Le premier groupe de méthodes de résolution est constitué par des méthodes exactes qui ont généralement des difficultés à traiter de grandes instances et peuvent nécessiter un temps de calcul long, même sur de petites instances. Pour cette raison, l'utilisation de méthodes approximatives est très utile pour atteindre une solution de bonne qualité pendant un temps d'exécution raisonnable. Ces méthodes approximatives peuvent être divisées en deux classes : heuristique et métaheuristique.

### 1.3.1 Méthodes exactes

Les méthodes exactes sont basées sur l'utilisation d'algorithmes qui permettent de trouver l'optimum global dans les champs d'optimisation combinatoire. Cependant, ils sont souvent coûteux en calcul.

Pour les problèmes NP-difficiles, l'énumération complète des solutions est généralement considérée comme une approche exacte mais la complexité reste exponentielle.

Deux méthodes majeures sont abordées afin d'énumérer les solutions :

- La programmation dynamique.
- La recherche arborescente.

#### La programmation dynamique

La programmation dynamique a été introduite par Bellman [14]. C'est une méthode très efficace pour résoudre les problèmes d'optimisation en les divisant en un ensemble de sous-problèmes, de manière récursive et en résolvant ces sous-problèmes de façon indépendante. Ce processus commence avec la plus petite partie du problème aux plus grands en stockant les résultats intermédiaires. La programmation

dynamique est basée sur une hypothèse simple, appelée le principe d'optimalité de Bellman : Toute solution optimale est construite sur des sous-problèmes, localement résolus à l'optimalité de Bellman. En pratique, cela signifie que nous pouvons déduire la solution optimale pour un problème en combinant des solutions optimales d'une série de sous-problèmes. La solution du problème commence par la résolution des sous-problèmes les plus petits et ensuite progressivement déduire les solutions globales du problème initial.

### La recherche arborescente

L'idée de la recherche arborescente repose sur l'énumération implicite de l'espace de recherche. Celui-ci est défini par des contraintes auxquelles le problème a été astreint, et aussi la représenter sous forme d'un arbre dont les sommets désignent des solutions. Parmi ses algorithmes les plus connus :

- **L'algorithme de séparation et évaluation :** connu également sous son appellation "branch and bound" a été proposé pour la première fois par Land et Doig [91]. Comme son nom l'indique, l'algorithme repose sur une recherche arborescente d'une solution optimale par séparations et évaluations, en présentant les solutions par un arbre d'états, avec des sommets et des feuilles. Le "branch and bound" est basé sur trois concepts : l'évaluation, la séparation et la stratégie de parcours.

  Une méthode naïve pour résoudre ce genre de problème consiste à énumérer toutes les solutions du problème, à calculer le coût pour chacun d'entre eux, puis à obtenir le minimum. Parfois, il est possible d'éviter d'énumérer des solutions déjà connues, en analysant les propriétés du problème. Aussi, on peut ne pas terminer la construction d'une solution non prometteuse, car celle-ci ne peut pas construire une solution optimale. En d'autres termes, il implique principalement la construction d'un arbre de recherche qui représente l'espace des solutions et l'élagage des branches inutiles de l'arbre qui contiennent des solutions non prometteuses ou infaisables.

  Le "branch and bound" peut trouver des solutions optimales pour VRP et HLP, mais en général le temps de résolution augmente lorsque la taille du problème augmente. Cela fonctionne bien pour les problèmes de petite taille, mais sinon, il peut générer de très grosses branches.

- **L'algorithme de séparation et coupe :** connu également sous son appellation "branch and cut". Cette méthode généralise celle de "branch and bound"

dans le sens où elle intègre un algorithme de coupes pour trouver la borne de chaque sous-problème. Le terme "branch and cut" a d'abord été introduit par Padberg et Rinaldi [112] pour résoudre le problème du voyageur de commerce (TSP). Pour plus de détails on cite Toth et Vigo, [139].

- **L'algorithme de branch and price :** a été proposé pour la première fois par Savelsbergh [123]. L'algorithme "branch and bound" avec des algorithmes de génération de colonnes sont appelés "branch and price". Un "branch and price" impliquent une exploration arborescente de la même manière que dans "branch and bound", mais au lieu d'avoir une simple résolution PL à chaque nœud, elle doit résoudre un PL avec génération de colonnes.

  La génération de colonnes est une méthode pour résoudre efficacement une grande programmation linéaire. elle est basée sur la décomposition de Dantzig et Wolfe [49] et décompose toutes les contraintes en deux problèmes : le problème principal et le sous-problème. Le problème principal est le problème original avec seulement un sous-ensemble de variables considérées. Le sous-problème est un nouveau problème créé pour identifier de nouvelles variables. La fonction objectif du sous-problème est de réduire le coût des nouvelles variables par rapport aux variables duelles actuelles.

  Le "branch and price" définit d'abord un problème principal en appliquant un relâchement des contraintes d'intégrité imposées lors de la descente de l'arbre du "branch and bound" et un sous-problème qui évalue chaque nouvelle colonne ou groupe de colonnes, ajouté au problème principal.

### 1.3.2 Méthodes approchées

La limite technologique actuelle, en termes de puissance de calcul des ordinateurs traditionnels, ne permet pas généralement la résolution optimale de grands problèmes dans des délais raisonnables. Cela est peut être l'une des raisons pour lesquelles, les méthodes approchées s'avèrent d'une grande utilité. Celles-ci, consistent à trouver des solutions quasi-optimales en un temps de calcul raisonnable, sans toutefois pouvoir en garantir l'optimalité.

#### Heuristique

Une heuristique est une méthode de calcul qui fournit rapidement (en temps polynomial) une solution réalisable, pas nécessairement optimale pour un problème d'optimisation NP-difficile.

Les heuristiques construisent progressivement une seule solution par une série de choix partiels et définitifs sans retour en arrière, ce qui signifie qu'ils ne contiennent pas de phase d'amélioration. En général, la construction d'une solution est réalisée par des décisions consécutives. Afin d'étendre la solution partielle, l'élément est sélectionné parmi différentes possibilités, en fonction d'un attribut qui optimise un critère. Généralement, chaque heuristique constructive est conçue spécifiquement pour un problème donné.

Nous reprenons la définition proposée par Duhamel et al : *Une heuristique est une méthode approchée dédiée à un problème et qui tente d'exploiter au mieux sa structure par des critères de décision déduits de la connaissance du problème* [53].

### Recherche locale

La recherche locale consiste à passer d'une solution à une autre solution proche dans l'espace de recherche jusqu'à ce qu'une solution considérée comme optimale soit trouvée, ou que le temps imparti soit dépassé. Le but de l'approche de recherche locale est d'améliorer une solution réalisable en explorant un voisinage de la solution actuelle. L'espace des voisinages est l'ensemble des solutions qui peuvent être obtenues par une simple transformation de la solution actuelle, en utilisant des mouvements prédéfinis. L'approche de recherche locale, recherche le premier voisin qui peut être obtenu par un mouvement d'amélioration si ce dernier est meilleur que la solution actuelle, la recherche locale explore le voisinage de la meilleure solution. Il s'arrête quand aucun voisin d'amélioration n'est possible.

### Métaheuristique

Le but de la métaheuristique est similaire à l'heuristique : obtenir de bonnes solutions dans un délai raisonnable. Les métaheuristiques sont généralement des algorithmes stochastiques itératifs, qui progressent vers un optimum global. Les métaheuristiques ont des structures qui peuvent théoriquement être appliquées à tout type de problème d'optimisation.

Nous reprenons la définition proposée par Duhamel et al : *Une métaheuristique est une méthode approchée générique dont le principe de fonctionnement repose sur des mécanismes généraux indépendants de tout problème* [53].

Dans la littérature, il existe deux grandes familles de métaheuristiques :

- Les méthodes basées sur l'exploration d'un voisinage sont parmi les métaheuristiques les plus classiques. L'idée est d'améliorer la solution actuelle trouvée dans l'espace de recherche à travers chaque itération en partant d'une solution initiale. On peut citer les méthodes les plus connues dans cette catégorie :

la méthode GRASP (Greedy Randomized Adaptive Search Procedure), le recuit simulé SA (Simulated Annealing), la recherche tabou TS (Tabu Search), la recherche locale itérative ILS (Iterative Local Search), la recherche du voisinage variable VNS (Variable Neighborhood Search).

- La deuxième famille de métaheuristique utilise une population de solution dans chaque itération en améliorant un ensemble de solutions, comme l'algorithme génétique GA (Genetic algorithm) et aussi comme l'algorithme de colonies de fourmis AC (Ant Colony).

**Greedy Randomized Adaptive Search Procedure (GRASP) :** GRASP a été introduit dans l'article de Thomas et Resende [137]. Cette métaheuristique produit une solution réalisable. Elle est exécutée plusieurs fois et la meilleure solution trouvée est conservée. Pour produire une solution, deux phases sont exécutées l'une après l'autre : la première consiste en une phase de construction de manière gourmande, suivie d'une phase de recherche locale pour améliorer la solution actuelle. GRASP est randomisé pour avoir une solution initiale différente à chaque itération.

**Simulated Annealing (SA) :** Le recuit simulé inspiré d'un procédé utilisé en métallurgie. Nous alternons dans les derniers cycles de refroidissement lent et de réchauffage (recuit) qui ont pour effet de minimiser l'énergie du matériau. Le recuit simulé a été développé par Kirkpatrick et al. [85]. Comparé à la recherche locale, le recuit simulé génère également une série de solutions mais le coût peut augmenter d'une solution à l'autre.

En partant d'une solution donnée, en la modifiant, nous obtenons une seconde solution. Soit il améliore le critère que l'on veut optimiser, ensuite on réduit l'énergie du système, soit il se dégrade. Si on accepte une solution améliorant le critère, on a tendance à chercher l'optimum au voisinage de la solution initiale. L'acceptation d'une "mauvaise" solution nous permet d'explorer une plus grande partie de l'espace de solution et d'éviter de tomber trop vite dans un optimum local.

**Tabu Search (TS) :** L'idée de la recherche tabou proposée par Glover [73], consiste à commencer à partir d'une solution donnée pour explorer le voisinage et choisir la solution dans ce voisinage qui minimise la fonction objectif. Il est essentiel de noter que cette opération peut conduire à augmenter la valeur de la fonction (dans un problème de minimisation). C'est le cas lorsque tous les points du voisinage ont une valeur plus élevée. En utilisant ce mécanisme, nous sortons d'un minimum local. Cependant, le risque survient lors de la prochaine étape, lorsque nous tombons dans le minimum local que nous venons d'échapper. Par conséquent, il est nécessaire de considérer une mémoire pour l'heuristique. Le mécanisme est d'interdire de revenir aux dernières solutions explorées. Les solutions déjà explorées sont stockées

dans une liste de tabous avec une taille donnée, qui est un paramètre ajustable de l'heuristique. L'algorithme est arrêté après un certain nombre d'itérations et renvoie la meilleure solution trouvée.

**Iterative Local Search (ILS) :** C'est une méthode définissant une modification des méthodes de recherche locale pour résoudre des problèmes d'optimisation discrets. Les méthodes de recherche locales peuvent être bloquées dans un minimum local, où aucun voisin améliorateur n'est disponible.
Une modification simple consiste à répéter les appels à la routine de recherche locale, chaque fois à partir d'une configuration initiale différente. C'est ce qu'on appelle la recherche locale répétée (ILS).
L'apprentissage implique que les minimaux locaux précédemment trouvés, est exploité pour produire de meilleurs points de départ pour la recherche locale. La recherche locale itérée est basée sur la construction d'une séquence de solutions localement optimales par : perturber le minimum local actuel et appliquer une recherche locale après avoir démarré à partir de la solution modifiée.

**Variable Neighborhood Search (VNS) :** La recherche du voisinage variable, proposée par Mladenovic et Hansen [102]. Il explore les voisinages éloignés de la solution actuelle, et passe de là à un nouveau si et seulement si une amélioration a été faite. La méthode de recherche locale est appliquée à plusieurs reprises pour obtenir des solutions dans le voisinage à des optimaux locaux. VNS a été conçu pour l'approximation de solutions de problèmes d'optimisation discrets et continus et selon ceux-ci, il vise à résoudre des problèmes de programmes linéaires, des problèmes de programmes entiers, des problèmes de programmes entiers mixtes, des problèmes de programmes non-linéaires, etc.

**Ant Colony (AC) :** Les algorithmes de colonies de fourmis s'inspirent du comportement des fourmis et constituent une famille de métaheuristiques d'optimisation. Proposé par Drigo M. [52], pour la recherche de chemins optimaux dans un graphe. Le premier algorithme était basé sur le comportement des fourmis cherchant un chemin entre leur colonie et une source de nourriture. L'idée originale est diversifiée pour résoudre une plus grande classe de problèmes et plusieurs algorithmes ont été développés, inspirés par divers aspects du comportement des fourmis. L'initialisation du graphe est faite en assignant un taux zéro de phéromones sur les arcs. Les fourmis sont représentées par des agents qui construisent la solution. Ils se déplacent dans le graphe en faisant des choix soumis à des probabilités sur les arcs à traverser. En effet, la construction de leur trajectoire est biaisée au profit d'arcs fortement marqués par leurs phéromones. Ensuite, en fonction de la valeur de la fonction objectif obtenue, les taux de phéromones sont mis à jour. Afin de choisir le meilleur chemin.

**Genetic Algorithms (GA) :** Les algorithmes génétiques appartiennent à la classe plus large des algorithmes évolutionnistes, proposés par Holland [80], qui génèrent des solutions aux problèmes d'optimisation en utilisant des techniques inspirées de l'évolution naturelle. Ils évoluent une population d'individus (solutions) par des phénomènes de reproduction et de mutation.

Le GA commence par une population initiale de solutions, représentée comme des chromosomes. Puis, à chaque itération, le croisement de deux individus parents de la population est fait, en utilisant une sélection. Le croisement de deux individus combine les caractéristiques de ces deux derniers pour générer deux nouveaux individus, appelé enfants. Des mutations peuvent survenir dans la création de la descendance, ce qui permet une diversification en évitant la convergence prématurée.

Ici on va présenter les grandes lignes de l'algorithme génétique :

- **Génération de la population initiale :** Initialement, une population de solutions individuelles est générée pour former une population initiale. Traditionnellement, deux modes de gestion de la population peuvent être utilisés. (1) Chaque itération crée un nombre d'enfants égal à la taille de la population initiale, et ce dernier est remplacé par la population d'enfants à l'itération suivante. (2) Chaque itération ne combine que deux parents, leurs enfants sont alors directement intégrés en remplaçant d'autres individus dans la population actuelle.

- **Évaluation de la solution (Fitness) :** Chaque génération de population subit une évaluation. L'évaluation d'un individu est évaluée en utilisant une fonction de coût (fonction objectif). Les valeurs de la fonction objectif indiquent le coût des solutions de la population.

- **Sélection :** Au cours de chaque génération successive, une partie de la population existante est sélectionnée pour élever une nouvelle génération. Un certain nombre de méthodes de sélection ont été développées pour identifier les individus pour la reproduction. Certaines méthodes de sélection évaluent l'aptitude de chaque solution et sélectionnent préférentiellement les meilleures solutions pour le croisement. D'autres méthodes évaluent seulement un échantillon aléatoire de la population et choisissent les individus. Parfois, les solutions sont sélectionnées de manière stochastique à partir de l'échantillon de sorte que les solutions moins adaptées ont également des chances de sélection.

- **Croisement :** Le croisement est le résultat obtenu lorsque deux chromosomes s'échangent des parties de leurs chaînes (leurs caractéristiques), pour donner

de nouveaux chromosomes. Croisement peut être simples ou multiples. Dans le premier cas, les deux chromosomes se croisent et s'échangent des portions d'ADN en un seul point. Dans le deuxième cas, il y a plusieurs points de croisement.

- **Mutation :** consiste à altérer un gène dans un chromosome selon un facteur de mutation. Ce facteur est la probabilité qu'une mutation soit effectuée sur un individu. Cet opérateur permet d'éviter une convergence prématurée de l'algorithme vers un extremum local.

Similaire à d'autres métaheuristiques, bien qu'elles ne garantissent pas une solution optimale à un problème donné, elles peuvent fournir de bonnes approximations dans un temps acceptable plutôt que d'utiliser une méthode exacte pour une solution optimale, qui serait intraitable par calcul pour les instances de grandes tailles.

## 1.4 Les problèmes de tournées de véhicules

Le problème du voyageur de commerce (TSP) est une catégorie importante de recherche opérationnelle et d'optimisation combinatoire qui consiste à définir la tournée d'un vendeur, à visiter toutes les villes d'un ensemble de villes prédéfinies et à retourner dans la ville de départ. L'objectif principal est de définir un ordre dans lequel chaque ville sera visitée une fois (la tournée est un cycle hamiltonien) en minimisant la distance de voyage. Le TSP est défini dans un graphe non orienté, évalué et complet. En ajoutant certaines contraintes ou hypothèses au TSP, de nombreux problèmes de routage pourraient être définis.

Le problème de tournées de véhicules (VRP) est une généralisation du TSP qui peut être décrit comme la construction de tournées pour plusieurs véhicules avec un coût minimal, d'un dépôt à un ensemble de points géographiquement distribués (clients, villes, magasins, entrepôts, écoles, etc.) afin de satisfaire les demandes requises dans le réseau et de retourner au dépôt lorsque la capacité du véhicule et / ou d'autres critères sont respectés (comme limite de temps ou de distance). Puisque le TSP (version simple du problème du routage) est un problème NP-difficile (Nemhauser et Wolsey [105]).

Du point de vue de complexité algorithmique, le VRP est classé NP-difficle [64]. La figure 1.1 montre un exemple illustré de VRP avec un dépôt et 25 clients. Cette solution propose cinq tournées afin de livrer tous les clients.

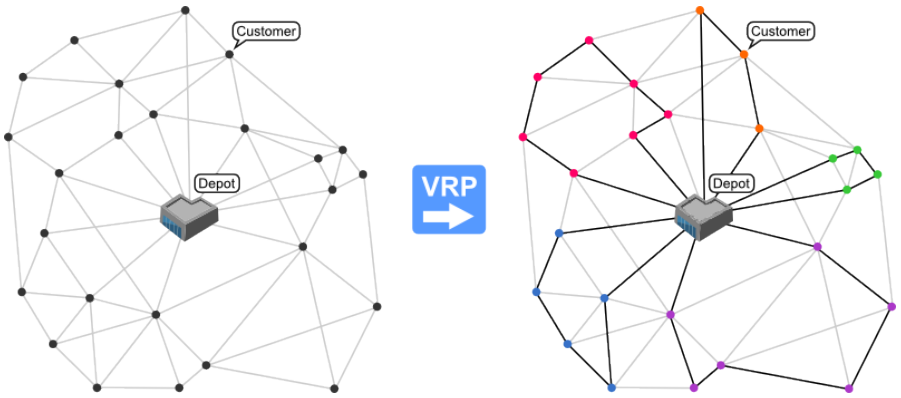



FIGURE 1.1 – Exemple du problème de tournées de véhicules.

Dans le VRP classique chaque solution réalisable devrait respecter trois contraintes majeures :

- Chaque ville est visitée exactement une seule fois, par un seul véhicule.
- Un véhicule commence et se termine au même dépôt et ne gère qu'une seule tournée.
- La quantité totale transportée lors de chaque tournée ne dépasse pas la capacité du véhicule.

Par conséquent, le VRP pourrait être une version généralisée de TSP avec plusieurs voyageurs qui seront appelés véhicules (tournée). Le but est de visiter un ensemble de clients géographiquement dispersés en utilisant une flotte de véhicules qui partent et retournent tous à un même dépôt. Dans chaque tournée créée, la somme des demandes des clients ne doit pas dépasser la capacité du véhicule. Si nous utilisons une flotte homogène, tous les véhicules offrent une capacité unique. Dans le VRP, nous considérons parfois une contrainte d'autonomie. Ce problème propose un temps de parcours maximal entre le départ du véhicule du dépôt et son retour au dépôt, et parfois chaque client possède un intervalle de temps dans lequel il doit être servi. Un sous-ensemble d'arcs du réseau de transport que les véhicules peuvent traverser et un sous-ensemble de clients qu'ils devraient être livrés par des véhicules, sont données comme des inputs du modèle.

Enfin, l'utilisation d'un véhicule nécessite un coût incluant une partie fixe et un coût du temps de service. L'objectif le plus commun pour le problème de tournées de véhicules est la minimisation du transport total, y compris le coût des tournées en fonction de la distance (temps des tournées ou temps de service) et du coût fixe de chaque véhicule usagé.

### 1.4.1 Formulation mathématique

Nous pouvons ainsi définir VRP classique comme un graphe $G = (V, E)$, avec $V = \{0, ..., n\}$ comporte le dépôt (nœud 0) et les $n$ clients. Au dépôt est basée une flotte de $m$ véhicules de capacité $C$. On note par $d_i$, la demande du client $i$ qui correspond à la quantité de produit qu'il faut lui livrer ou collecter, le coût $c_{ij}$ pour se déplacer entre les clients $i$ et $j$. Nous définissons également les variables binaires comme suit :

$$x_{ijk} = \begin{cases} 1 & \text{si le véhicule } k \text{ travérse le trajet } (i,j). \\ 0, & \text{sinon} \end{cases}$$

Nous présentons la formulation mathématique utilisée en programmation linéaire, adoptée par Crainic et Semet [48] :

$$Minimiser \sum_{i=1}^{n}\sum_{j=1}^{n} c_{ij} \sum_{k=1}^{m} x_{ijk} \tag{1.1}$$

Sous les contraintes :

$$\sum_{i=1}^{n}\sum_{k=1}^{m} x_{ijk} = 1 \quad \forall\ 1 \leq j \leq n \tag{1.2}$$

$$\sum_{j=1}^{n}\sum_{k=1}^{m} x_{ijk} = 1 \quad \forall\ 1 \leq i \leq n \tag{1.3}$$

$$\sum_{i=1}^{n}\sum_{l=1}^{n} x_{ilk} = \sum_{l=1}^{n}\sum_{j=1}^{n} x_{ljk} \tag{1.4}$$

$$\sum_{j=1}^{n} x_{0jk} = 1 \quad \forall\ 1 \leq k \leq m \tag{1.5}$$

$$\sum_{i=1}^{n} x_{i0k} = 1 \quad \forall\ 1 \leq k \leq m \tag{1.6}$$

$$\sum_{i=1}^{n}\sum_{j=1}^{n} x_{ijk} \leq C \quad \forall\ 1 \leq k \leq n \tag{1.7}$$

$$x_{ijk} \in 0, 1 \quad \forall\ 0 \leq i, j \leq n;\ 1 \leq k \leq m \tag{1.8}$$

La fonction objectif (1.1) cherche à minimiser la somme des coûts de toutes les tournées. Les contraintes (1.2) et (1.3) imposent que chaque client soit servi une et une seule fois ; la contrainte (1.4) assure la conservation dépôt. Les contraintes (1.5) et (1.6) assurent que chaque tournée commence et se termine au dépôt. La contrainte (1.7) garantie que la capacité de n'importe quel véhicule doit être respectée et inférieure ou égale à $C$ et enfin la contrainte (1.8) est une contrainte de binarité sur la

variable de décision $x_{ijk}$.

Ce VRP classique constitue une base modifiable selon les caractéristiques du problème étudié. Ces modifications dépendent principalement de quatre paramètres : la fonction objectif, les contraintes, la flotte des véhicules et la demande des clients.

- *La Fonction objectif :* Dans les problèmes de tournées de véhicules, l'objectif peut varier selon le problème étudié. Il consiste généralement en une fonction du coût à minimiser ou une fonction du profit à maximiser. Par exemple soit minimiser la distance totale des tournées, minimiser le temps total des tournées ou minimiser le nombre de véhicules.

- *les contraintes :* Les contraintes peuvent être diverses, retenons en quelques-unes : (1) La capacité du véhicule doit être limitée. (2) Le facteur temporel doit être pris en compte dans les opérations de pré-service (disponibilité des clients ou fenêtres horaires, préparation du matériel, formalités administratives,...), les opérations de service (collecte, livraison) et au cours du parcours des trajets.

- *La flotte :* La flotte de véhicules se différencie selon le critère d'homogénéité, elle est soit homogène lorsque tous les véhicules ont la même capacité, soit hétérogène si la flotte se divise à plusieurs types, chacun est caractérisé par une capacité et un coût de déplacement. Également, on peut distinguer entre les problèmes lorsque la flotte de véhicules est limitée si ces derniers sont disponibles en nombre limité. Chaque véhicule transporte une quantité limitée soit par la réglementation en vigueur, soit par les contraintes techniques et pratiques du problème.

- *La demande :* Dans les problèmes de tournées de véhicules, le besoin de chaque client se caractérise par sa demande en un ou plusieurs produits. Celle-ci peut être déterministe ou stochastique. De plus, elle peut être dynamique au sens où elle ne peut pas être connue à priori, mais elle apparaît plutôt pendant le déroulement des tournées. Le problème dynamique nécessite une résolution concurrente à l'exécution, contrairement à ceux dont l'évolution dans le temps est déterministe où la demande varie de façon aléatoire (stochastique) qui peuvent être résolus à priori par des approches probabilistes [125]. Dans un autre contexte, la demande peut être contrainte à (1) une fenêtre de temps de visite durant laquelle le client doit être servi, et (2) une précédence comme le cas du problème de collectes et livraisons.

### 1.4.2 Les variantes du VRP

Le VRP classique offre des tournées de livraison où chaque véhicule parcourt une tournée, chaque véhicule possède les mêmes caractéristiques et un seul dépôt. Le but des VRP est de minimiser le coût des véhicules exploités de manière à ce que chaque client soit rencontré exactement une fois par tournée. Nous considérons que chaque véhicule démarre et se retourne au même dépôt. Nous notons que chaque véhicule a une capacité et que la capacité des véhicules doit être respectée en cours de la tournée. Un état de l'art détaillé des variantes de VRP est donné dans le livre de Toth et Vigo [140], dans les travaux de Cordeau et al.[45] et Laporte et al. [93].

De nombreuses variantes de VRP ont été définies en élargissant ce problème de base, en ajoutant différentes contraintes et / ou objectifs. La littérature consacrée à ces problèmes est très vaste. Pour identifier les différentes variantes de problème de tournées de véhicules, les auteurs utilisent généralement des initialismes, dans lesquels différents préfixes et suffixes indiquent la présence de différentes hypothèses ou contraintes. Mais cette identification basée sur des initialismes est inefficace. Ramdane et al. [39] proposent une nouvelle notation et un nouveau système de classification pour identifier les problèmes de tournées de véhicules sans ambiguïté. Il décrit les problèmes traités par leurs hypothèses, contraintes et objectifs plutôt que par des initiales.

Dans le cas général, une tournée de véhicule commence par un dépôt et retourne au point de départ, mais il existe quelques cas comme des VRP ouverts où les véhicules terminent dans une autre ville comme un dépôt final (pour plus d'information pour Open VRP voir Sariklis et Powell [121]). Aussi on a multi dépôts VRP, où il y a plusieurs dépôts qui peuvent servir leurs clients, mais pas nécessairement avec des caractéristiques identiques (voir Mingozzi et Valletta [101]).

Concernant la contrainte de capacité, il y a le multi capacités VRP qui est un VRP dans lequel il a plusieurs types de véhicules avec différentes capacités [84].

Le VRPTW est une autre extension du VRP avec fenêtre de temps. Elle garantit le service à un client effectué pendant un intervalle de temps donné par les clients. Les fenêtres temporelles souples permettent des livraisons hors des limites moyennant avec un coût de pénalité (voir Tas et al. [134]). Par contre pour les fenêtres temporelles considérées comme strictes, nous ne sommes pas autorisés à livrer en dehors de l'intervalle de temps (voir Vidal et al. [145]).

Les problèmes de tournées de véhicules dynamiques (DVRP), également appelés "problèmes de tournées de véhicules en ligne ou en temps réel", sont apparus récem-

ment en raison des progrès des technologies de l'information et des communications. Dans les problèmes de tournées de véhicules dynamiques, les demandes peuvent être connues à l'avance avant le début de la journée de travail ; aussi des nouvelles demandes sont reçues au fur et à mesure de l'avance du temps et doivent être insérées de manière dynamique dans les tournées déjà planifiées alors que les véhicules ont déjà commencé leurs tournées [16].

### 1.4.3 Méthodes de résolution du VRP

Le nombre de méthodes pour résoudre le VRP introduit dans la littérature a augmenté rapidement au cours des dernières années. Selon la récente étude fournie par Hall [77], des milliers de grandes entreprises utilisent des logiciels du VRP. Selon l'état technique et les contraintes du VRP, nous pouvons trouver de nombreux travaux liés aux méthodes de résolution des VRPs (voir Toth et Vigo [140]). Elles ont commencé par des méthodologies exactes telles que la programmation linéaire, la programmation dynamique ou les algorithmes d'arborescente [92]. On peut citer Fischetti et al. [60] qui ont proposé des algorithmes utilisant "branch and bound" pour résoudre le VRP classique. Un algorithme exact de "branch and price" a été aussi proposé par Gutierrez-Jarpa et al. [76] pour résoudre le problème de tournées de véhicules multiples avec fenêtre de temps.

Comme autre approche exacte, Eilon et al. [54] a probablement proposé pour la première fois la programmation dynamique pour le VRP. Cette méthode a été utilisée pour les problèmes de VRP de très petites tailles (Rego et al. [119]) pour résoudre des problèmes avec 10 à 25 clients.
Aujourd'hui, les meilleures méthodes exactes pour les VRPs sont dominées par les approches polyédriques (Augerat et al. [11]), et plus généralement les méthodes "branch and cut".

Dans le travail de Tan [133], le problème du nombre de clients ($> 100$) ne peut être résolu par les approches exactes pendant un temps d'exécution raisonnable. Par conséquent, les chercheurs utilisent des heuristiques et des métaheuristiques pour résoudre ce type de problèmes à cause de la difficulté du problème (NP-difficile). Les méthodes exactes peuvent généralement résoudre efficacement des problèmes jusqu'à 50 clients. En 2003, une méthode de "branch and cut" en parallèle a pu résoudre un problème avec 100 clients, proposée par Ralphs, [117]. Une revue de la littérature des approches exactes du VRP est présenté dans les deux ouvrages de Toth et Vigo [140] et de Golden et al. [75].

En comparaison des méthodes exactes, les heuristiques et les métaheuristiques

seraient souvent plus compétitives pour les applications réelles, car les problèmes auxquels nous sommes confrontés aujourd'hui sont considérablement plus importants.
Les heuristiques de VRP peuvent être divisées en deux classes principales : les heuristiques classiques également appelées heuristiques simples et les nouvelles heuristiques ou métaheuristiques.

Parmi les heuristiques les plus simples et les plus efficaces, celle de Clarke et Wright [41]. C'est la première heuristique proposée pour résoudre le VRP classique. Cette méthode consiste à construire une solution triviale qui considére chaque client dans une tournée, ensuite calculer les gains $S_{ij}$ pour chaque client $i$ et $j$, et les trier par ordre décroissant, enfin parcourir la liste des gains pour fusionner des tournées qui ont le plus grand gain. La fusion des tournées ne s'effectue que si toutes les contraintes du problème sont vérifiées (capacités, etc) et l'algorithme s'arrête lorsqu'il n'y a plus de fusions améliorantes. Lorsque deux tournées (0, ..., i, 0) et (0, j, ..., 0) sont regroupées en une seule (0, ..., i, j, ..., 0), par conséquent le gain $S_{ij} = c_{i0} + c_{0j} - c_{ij}$ a été le plus grand. Ainsi cette heuristique peut réduire à la fois le coût total et le nombre de véhicules utilisés. De nombreuses améliorations de cette heuristique ont été proposées afin de réduire son temps d'exécution comme dans Paessens [113], ou d'effectuer une série de fusion en parallèle (Desrochers et Verhoog [50]).

La méthode d'insertion est une autre méthode constructive qui s'est avérée être une méthode populaire pour résoudre une variété de routage de véhicule. L'algorithme d'insertion se déroule en deux phases ; la première phase sélectionne les nœuds à insérer et la seconde phase sera appliquée pour l'insérer dans une tournée, Toth et Vigo [140].

Plus compliqué que les heuristiques constructives, il existe des heuristiques en deux phases : regroupement des clients comme première phase, puis routage "cluster-first route-second" ou premier routage, puis regroupement "route-first cluster-second". Le principe de "cluster-first route-second" consiste à répartir les clients dans un secteur séparé avec une demande totale proche de la capacité des véhicules, puis à construire une tournée pour chaque secteur. Le plus connu de ce type de méthode est le "Sweep Algorithm" (algorithme de balayage), qui utilise des secteurs angulaires. Cette méthode fonctionne bien si le dépôt est relativement central. Les origines de l'algorithme de balayage peuvent être remontées aux travaux de Gillett et Miller [71] pour le VRP classique, dans lesquels les sommets sont situés dans le plan euclidien.

Fisher et Jaikumar [62] proposent une phase de segmentation par sectorisation basée sur un problème d'affectation généralisé (GAP). En revanche, Beasley [13] a proposé une méthode "route-first cluster-second" qui fonctionne en sens inverse. Il s'agit cette fois-ci de générer une énorme tournée comme le TSP, il ne prend donc pas en compte la capacité des véhicules. Cette tournée est ensuite divisée en tournées réalisables en respectant la capacité des véhicules.

Concernant les métaheuristiques pour résoudre le VRP, de nombreuses métaheuristiques ont été proposées dans la littérature dont nous présentons les plus connues. Osman [111] a proposé un recuit simulé pour le VRP, cela fonctionne bien, mais il a été rapidement dépassé par les méthodes de recherche tabou. En 2005, Cordeau et Laporte [46] reportent dix recherches tabou plus éfficaces à cette époque. Les trois plus éfficaces sont l'algorithme parallel de Taillard [131], Taburoute de Gendreau et al. [67] et Granular Tabu Search (GTS) de Toth et Vigo [141]. Dans la littérature sur les VRPs, nous pouvons également mentionner d'autres méthodes éfficaces telles que la recherche locale itérative de Li et al. [94], "Adaptive Large Neighborhood Search" proposé par Hemmelmayr et al. [79] et algorithme génétique de Tasan et Gen [135].

## 1.5 Les problèmes de localisation des Hubs

Les problèmes de localisation des hubs (HLP) sont des problèmes d'optimisation combinatoire connus dans plusieurs domaines comme le transport et les télécommunications et ont une large gamme d'applications comme les systèmes de distribution postale, les réseaux de transport aérien ou terrestre. De nombreuses entreprises de livraison postale peuvent livrer des millions de colis par jour. Envoyer les colis directement de l'origine à la destination n'est ni économique ni pratique pour ces entreprises, car elles localisent plusieurs installations de transbordement, appelées hubs, afin d'agréger et de séparer les demandes entre chaque paire origine-destination. La structure en réseau de ces systèmes de transport basés sur la consolidation est principalement organisée en étoile.

Dans les réseaux en étoile, l'hypothèse est la suivante :
Les hubs sont entièrement connectés par des voies à haut volume et à faible coût permettant d'appliquer un facteur de réduction au coût de transport du flux entre une paire de hubs donnée. Une autre hypothèse de ces réseaux est que tout le flux internodal passe par au moins un hub et au plus deux. Globalement, le problème de localisation des hubs concerne la localisation des hubs sur le réseau et l'affectation des nœuds non-hubs aux hubs afin de minimiser le coût total du flux. Le trafic dans le réseau comprend la collecte (des nœuds d'origine aux hubs), le transfert (entre les

hubs) et la distribution (des hubs aux nœuds de destination). Les coûts de transport correspondants sont multipliés respectivement par $\chi$, $\alpha$ et $\delta$, où $\alpha$ est utilisé comme facteur d'actualisation pour réduire les coûts unitaires sur les arcs entre les hubs afin de refléter les économies d'échelle [29], $\alpha < \chi$ et $\alpha < \delta$.

Les réseaux en étoile ont des applications dans de nombreux domaines. Parmi les exemples courants, citons les compagnies aériennes [12], les systèmes de télécommunication [110], les entreprises de livraison de colis exprès [90], les entreprises de camionnage [136], les chaînes de magasins [3] et de nombreux autres domaines. De nombreuses études ont indiqué que la mise en place d'un réseau en étoile avait amélioré les performances du système de distribution. En raison de leurs multiples applications et de leur valeur économique, les HLPs ont fait l'objet de beaucoup d'attentions dans la littérature.

Le problème de localisation des hubs a de nombreuses variétés en fonction des contraintes et des variables de décision impliquées, comme la manière de sélectionner le nombre de hubs à localiser, la répartition des non-hubs entre hubs, l'existence des limites de capacité sur les hubs, etc. Des revues, synthèses et classifications sur des modèles et des méthodes utilisées dans la littérature sur différentes variantes du HLP peuvent être trouvées dans Alumur et Kara, [7], Campbell et O'Kelly [30] et Farahani et al. [59].

Avant de détailler les variantes du HLP ; la section suivante va présenter le modèle mathématique d'une variante, simple et qui représente la base de toutes les variantes, connu sous le nom "The Uncapacitated Single Allocation p-Hub Median Problem (USApHMP)".

### 1.5.1 Formulation mathématique

L'USApHMP consiste à choisir $p$ hubs emplacements à partir de l'ensemble des nœuds et d'affecter chaque non-hubs à un seul hub afin d'acheminer le trafic entre chaque paire de nœuds de telle sorte que le coût total du trafic dans le réseau soit minimal.

Notons que :

- $N$ l'ensemble des nœuds ; $i, j, k, l \in \{1, 2, \ldots, n\}$ ;
- $p$ le nombre de hubs à localiser ;
- $w_{ij}$ la quantité de flux originaire du nœud $i$ à $j$ ;
- $d_{ij}$ la distance entre le nœud $i$ et le nœud $j$ ;

- $O_i = \sum_j w_{ij}$ le flux total sortant du nœud $i$ ;
- $D_i = \sum_j w_{ji}$ le flux total entrant au nœud $i$ ;
- $C_{iklj}$ le coût de transport du nœud $i$ à $j$ ;

Puisque le coût de transport du flux du nœud $i$ à $j$ est $C_{iklj} = w_{ij}(\chi.d_{ik}+\alpha.d_{kl}+\delta.d_{lj})$ ; Le coût total du réseau est la somme de tous les $C_{iklj}$.

L'USApHMP est formulé par Ernst et Krishnamoorthy [56] comme suit :

$$minimise \sum_{i\in N}\sum_{k\in N} d_{ik}Z_{ik}(\chi O_i + \delta D_i) + \sum_{i\in N}\sum_{k\in N}\sum_{l\in N} \alpha d_{kl}Y^i_{kl} \tag{1.9}$$

Sous les contraintes :

$$\sum_{k\in N} Z_{kk} = p \tag{1.10}$$

$$\sum_{k\in N} Z_{ik} = 1 \qquad i \in N \tag{1.11}$$

$$Z_{ik} \leq Z_{kk} \qquad i,k \in N \tag{1.12}$$

$$\sum_{l\in N} Y^i_{kl} - \sum_{l\in N} Y^i_{lk} = O_i Z_{ik} - \sum_{j\in N} w_{ij}Z_{ik} \qquad i,k \in N \tag{1.13}$$

$$Z_{ik} \in \{0,1\} \qquad i,k \in N \tag{1.14}$$

$$Y^i_{kl} \geq 0 \qquad i,l,k \in N \tag{1.15}$$

La fonction objectif (1.9) minimise le coût total du transport prévu de la collecte, du transfert et de la distribution des flux entre tous les nœuds d'origine vers les nœuds de destination. La première partie de la fonction objectif sert à minimiser les coûts de transport entre les hubs et les non-hubs, et la seconde minimise les coûts de transport entre hubs. La contrainte (1.10) garantit la localisation exacte de $p$ hubs. La contrainte (1.11) impose que chaque non-hub soit affecté à un seul hub. La contrainte (1.12) garantit que les hubs sont établis pour chaque distribution / collecte, empêchant ainsi la transmission directe entre les nœuds non-hubs. La contrainte (1.13) est la contrainte de conservation du flux.

La figure 1.2 montre un exemple illustré de l'USApHMP avec $n$=7 nœuds et $p$=2 (localisation de 2 hubs). La solution sélectionne les nœuds 2 et 7 comme des hubs, et affecte les nœuds 1, 2, 4 et 6 au hubs 2 et les nœuds 3, 5 et 7 au hubs 7 (chaque hub est affecté à lui même).

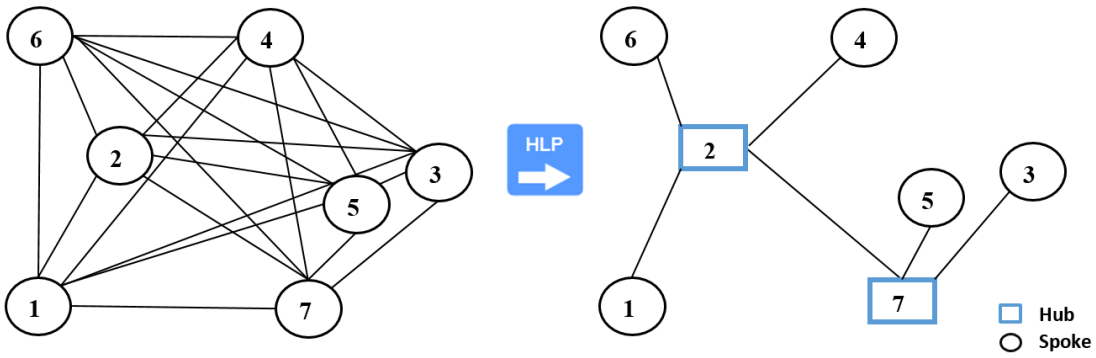



FIGURE 1.2 – Exemple de l'USApHMP.

Différentes variantes du problème de localisation des hubs ont été définies et classées selon :

- La maniére d'affectation : deux catégories principales, l'affectation unique, où chaque non-hub est affecté à un seul hub (Ilic et al. [81]), tandis que la variante multiple permet d'affecter les non-hub à plusieurs hubs (voir Kratica [86] et Stanimirovic [127]), pour consulter les modéles mathématiques, nous citons Boland et al. [27] et dans Stanimirovic [127].

- Le nombre de hubs : lorsque le nombre de hubs $p$ est donné, le problème est appelé problème de localisation p-hub médiane, sinon le problème est dit problème de localisation des hubs. Dans ce dernier cas, le nombre de hubs et leur localisation sont des décisions à prendre. Ceci est formulé en ajoutant un terme supplémentaire à la fonction objectif, visant à minimiser le coût de transport dans le réseau et aussi de minimiser le coût fixe de l'installation des hubs.

  Concernant le modèle mathématique du problème "The Uncapacitated Single Allocation Hub Location Problem (USAHLP)" est formulé comme suit :

$$minimise \sum_{i\in N}\sum_{k\in N} d_{ik}Z_{ik}(\chi O_i + \delta D_i) + \sum_{i\in N}\sum_{k\in N}\sum_{l\in N} \alpha d_{kl}Y^i_{kl} + \sum_{k\in N} F_k Z_{kk}. \quad (1.16)$$

  Sous les contraintes (1.11)-(1.15), avec $F_k$ est le coût d'installer le nœud $k$ comme hub.

- La capacité des hubs : lorsque les hubs ont des limites de capacité pour gérer le flux entre les nœuds le problème est dit HLP avec capacité voir Ernst and

Krishnamoorthy [58] et Stanimirovic [128]. Cela est formulé en ajoutant la contrainte de capacité :

$$\sum_{i \in N} O_i Z_{ik} \leq G_k Z_{kk} \qquad i, k \in N \tag{1.17}$$

avec $G_k$ est la capacité du hub k.

### 1.5.2 Les variantes du HLP

Dans la littérature, il y a un très grand nombre de variantes du HLP, dans cette section nous présentons seulement les variantes que nous allons traiter dans cette thèse.

Dans le problème de localisation des hubs nommé *"Single Allocation Hub Location Problem (SAHLP)"*, le nombre de hubs à localiser n'est pas connu à l'avance mais c'est une décision à prendre. Par contre dans *"Single Allocation p-Hub Median Problem (SApHMP)"* le nombre de hubs est connu à l'avance et présenté par le paramètre 'p'.

Pour chaque variante soit SAHLP ou SApHMP, on peut définir d'autre variantes en changeant le type d'affectation des non-hubs au hubs de single à multiple pour avoir *"Multiple Allocation Hub Location Problem (MAHLP)"* et *"Multiple Allocation p-Hub Median Problem (MApHMP)"* qui va permettre l'affectation d'un non-hub à plusieurs hubs.

De plus d'autres variantes sont gérées lorsque les hubs ont des limites de capacité ou non. Les hubs avec capacité peuvent gérer un flux limité (*Capacitated SApHMP* et *Capacitated SAHLP*), on peut les noter simplement par CSApHMP et CSAHLP. Tandis que les hubs sans capacité peuvent gérer n'importe quel débit de flux, ce qui donne (*Uncapacitated SApHMP* et *Uncapacitated SAHLP*), on peut les noter simplement par USApHMP et USAHLP.

Avec ces trois types de contraintes : (1) le nombre de hubs $p$ est soit connu ou inconnu, (2) la méthode d'affectation unique ou multiple et (3) selon la capacité des hubs, on peut combiner entre ces trois et générer plusieurs types de variantes dont dépend des besoins des entreprises et du marché commercial. Un exemple d'application du CSAHLP est celui des systèmes de distribution postale, dans lesquels un centre de tri (ou hub) regroupe des envois provenant de différents districts postaux et les achemine vers la destination généralement par d'autres centres [58]. Les centres de tri dans de tels systèmes ont des capacités, c'est-à-dire qu'ils peuvent

traiter une quantité maximale de flux de courrier à partir des points d'origine, et chaque courrier est acheminé par une seule voie (unique affectation).

### 1.5.3 Méthodes de résolution du HLP

L'USApHMP est un problème déjà démontré comme un problème NP-difficile [7, 109, 110]. De plus, dans l'USApHMP, le nombre de hubs est connu a priori et la contrainte d'affectation est unique. Dans la version USAHLP où le nombre de hubs est une décision, ou dans la version CSAHLP où les hubs ont des limites de capacité pour gérer le flux entre les nœuds. Cela rend les problèmes USAHLP, CSApHMP et CSAHLP plus difficiles [56, 58], de même pour les problèmes avec multi-affectations [27] qui sont tous démontrés que ce sont des problèmes NP-difficiles.

En raison de leur utilité et de leur importance économique, les versions du HLP ont fait l'objet d'une attention considérable de la part de la recherche, des méthodes exactes, des heuristiques et des métaheuristiques ont été proposées pour les résoudre. Certaines de ces méthodes incluent une formulation de programmation en nombre entier quadratique [109] et sa linéarisation [7].

O'kelly [109] a présenté la première formulation mathématique du problème USApHMP en tant que programme entier quadratique. Il a développé deux heuristiques et a reporté des résultats numériques pour les instances CAB (Civilian Aeronautics Board) avec 25, 20, 15 et 10 nœuds.

Ernst et Krishnamoorthy [56], ont développé une heuristique basée sur le recuit simulé (SA). Ils ont utilisé la limite supérieure du SA pour développer une méthode de solution pour trouver la borne inférieure du "branch and bound". Ils ont aussi reporté les résultats de l'ensemble d'instances AP (Australian Post) est un ensemble d'instances réelles représentant les flux de courrier en Australie.

Abdinnour-Helm et Venkataramanan [4] ont proposé un algorithme exact "branch and bound" et un algorithme génétique (GA) pour l'USAHLP, l'algorithme "branch and bound" proposé à résoudre des problèmes jusqu'à 15 nœuds, tandis que l'algorithme génétique résout efficacement l'ensemble d'instances CAB (25 nœuds).

Contreras et al. [43] ont présenté un algorithme de "branch and price" pour CSAHLP dans lequel une relaxation est utilisée pour obtenir la borne inférieure du problème ; Aykin [12] a présenté CSAHLP où les transferts directs d'origine-destination sont autorisés. Il a proposé un algorithme de "branch and bound" dans lequel les bornes inférieures sont obtenues par une relaxation lagrangean.

Boland et al [27] ont mis au point une méthode exacte de "branch and bound" pour résoudre l'UMAHLP en utilisant des algorithmes de prétraitement et de découpage. Ce sont les premiers qui obtiennent les meilleures bornes supérieures qui sont utilisées pour couper la taille de "branch and bound", mais cette approche ne donne des résultats que sur des instances de plus petites tailles. Les méthodes exactes peuvent généralement résoudre efficacement les problèmes mais seulement pour des petites instances.

Chen [37] a proposé une heuristique hybride pour résoudre l'USAHLP. Sa méthode est basée sur la combinaison d'une méthode qui cherche la limite supérieure du nombre de hubs à localiser et d'un recuit simulé avec une méthode recherche tabou, et elle a été testée sur les instances CAB et les instances AP jusqu'à 200 nœuds. Abyazi-Sani et Ghanbari [5] ont proposé une heuristique de recherche tabou pour résoudre l'USAHLP et ont reporté les résultats à la fois sur les instances CAB et l'ensemble d'instances AP jusqu'à 400 nœuds.

Pour résoudre l'USAHLP avec des métaheuristiques, une recherche tabou (TS) [124], un algorithme génétique (GA)[138] et une heuristique hybride combinant GA et TS [2] sont proposés pour résoudre cette variante. Même la méthode VNS est proposée par Ilic et al. [81] pour résoudre l'USApHMP. Ils ont reporté des résultats de larges instances AP jusqu'à 400 nœuds, des instances PlanetLab et des instances aléatoires jusqu'à 1000 nœuds.

Aussi pour résoudre CSAHLP, Kratica et al [88] et Erken [55] ont proposé des GAs. Ernst et Krishnamoorthy [58] ont également présenté les instances AP pour ce type de variante (CSAHLP), qui a depuis été utilisé par divers travaux de recherche.

Randall [118] applique l'algorithme de colonies de fourmis (AC) pour résoudre CSAHLP. Dans son travail, quatre variantes de l'algorithme AC basées chacune sur un choix de modélisation de construction différent et combinées avec la recherche de voisinage multiples ont été développées.

Kratica et al [88] ont également proposé un GA pour l'allocation multiple sans capacité (UMAHLP). Aussi Stanimirovic Z. [127] a développé une heuristique basée sur GA pour résoudre l'UMApHMP et il a fourni des résultats numérique pour les deux instances les plus connues CAB et AP pour justifier la performance de sa métaheuristique.

## 1.6 Conclusion

Ce chapitre a présenté un aperçu des concepts de base liés au VRP et au HLP. Nous avons commencé par un petit survol sur les principaux concepts relatif aux problèmes de transport et de localisation des hubs et leurs importances dans différents domaines. Des formulations mathématiques de base ont également été présentées, suivies par un bref rappel des variantes des problèmes et des méthodes classiques de résolution, des exactes, des heuristiques et des métaheuristiques sont proposées pour ces deux problèmes d'optimisation combinatoire.

Cependant, en comparant les trois dernières méthodes, il ressort que c'est souvent plus utile d'utiliser des heuristiques ou des métaheuristiques au lieu des méthodes exactes. Cela dit, il semble que le problème lié à la résolution des grandes instances dans un temps raisonnable, se pose surtout et souvent au niveau des applications actuelles des entreprises, dont le nombre de clients se compte pas des millions. Pour accélérer le calcul et augmenter la performance des méthodes proposées, plusieurs chercheurs ont orienté leurs recherches vers le parallélisme afin de profiter de la puissance de calcul parallèle, pour résoudre des problèmes d'optimisation combinatoire de grandes tailles dans un temps d'exécution raisonnable. C'est effectivement l'objectif de cette thèse.

Avant d'entrer dans le vif du sujet et présenter nos implémentations sur GPU, nous présenterons dans le chapitre suivant un contexte général sur la programmation parallèle sur GPU dans lequel nous expliquerons la performance, l'architecture des GPUs, et nous décrirons l'architecture CUDA. Nous terminerons finalement par montrer la réussite de certaines applications CUDA.

Chapitre 2

# La programmation parallèle sur GPU

## 2.1 Introduction

La programmation parallèle était toujours considérée comme étant un domaine spécifique de l'informatique. Cette pensée a beaucoup évolué ces dernières années. Les demandes en capacité de calcul n'ont cessé d'augmenter donnant lieu à de nombreuses applications scientifiques, techniques, militaires, médicales, etc.

Pour augmenter la puissance de calcul, la première solution était d'augmenter la fréquence du processeur mais cette solution est heurtée à plusieurs problèmes généralement liés à la surchauffe. La solution qu'apparait plus efficace est de faire coopérer plusieurs unités de traitement en multipliant les ressources matérielles pour diviser le temps de traitement. Cette solution, qui consiste à répartir les traitements sur ces unités, s'appelle le parallélisme. Dans ce contexte diverses architectures, dites parallèles sont apparues, supercalculateurs multi-processeur, multi-coeurs, grille de calcul et récemment les GPUs (Graphics Processing Unit). Ces architectures ont créé un nouveau domaine de l'informatique qui est le Calcul Haute Performance (HPC).

Ce chapitre se concentre sur les GPUs et sur l'architecture CUDA. L'unité de traitement graphique (GPU) est disponible sur chaque PC, ordinateur portable, ordinateur de bureau et poste de travail. Dans sa forme la plus élémentaire, la carte graphique génère des graphiques, des images et des vidéos 2D et 3D, des interfaces graphiques, des jeux vidéo, des applications d'imagerie visuelle. Le GPU moderne que nous décrivons ici est un multiprocesseur hautement parallèle, hautement multithreading. Pour fournir une interaction visuelle en temps réel avec des objets calculés via des graphiques, des images et des vidéos, le GPU possède une architecture graphique et informatique unifiée qui sert à la fois de processeur graphique program-

mable et de plate-forme de calcul parallèle évolutive pour faire du calcul scientifique, technique, médicale et même résoudre des problèmes d'optimisation combinatoire.

Dans la première section de ce chapitre, (section 2.2) ; nous introduisons l'historique de l'évolution des unités de traitement CPUs et des GPUs pour les ordinateurs personnels. Ensuite, la section 2.3 présente, une définition des systèmes hétérogènes utilisés actuellement et une présentation matérielle de l'architecture de base du GPU. La section 2.4 est réservée à l'architecture CUDA, où nous allons présenter l'organisation hiérarchique des processeurs, l'organisation hiérarchique des types de mémoire, la communication entre CPU et GPU et enfin le langage CUDA. Avant de conclure ce chapitre, la section 2.5 fait le point sur de nombreuses entreprises et sociétés qui ont connu un grand succès en choisissant de créer leurs applications avec CUDA C.

## 2.2 Evolution des CPUs et des GPUs

Le critère naturel de performance pour un calculateur scientifique, est la vitesse de calcul, c-à-d le nombre d'opérations arithmétiques réalisable par seconde. Cette vitesse est dépendante de la technologie des composants, qui peut elle-même se mesurer à priori par la fréquence des micro-processeurs.

Depuis trente ans, l'une des méthodes les plus importantes pour améliorer les performances des ordinateurs a consisté à augmenter la fréquence d'horloge du processeur. Les unités centrales de traitement (CPU) des premiers ordinateurs personnels du début des années 1980 était environ 1 MHz. Trente ans plus tard, la plupart des processeurs des ordinateurs ont une vitesse d'horloge comprise entre 1 GHz et 4 GHz, 1 000 fois plus rapide que l'horloge du premier ordinateur.

Ces dernières années et exactement en 2005, les constructeurs ont été contraints de rechercher des alternatives, car plusieurs limites fondamentales des circuits intégrés font qu'il n'est pas possible de se contenter d'augmenter infiniment la fréquence d'horloge d'un processeur. Les contraintes d'alimentation et de dissipation thermique ainsi que la limite physique de la taille des transistors, ont donc imposé la recherche d'autres solutions.

Les supercalculateurs améliorent également leurs performances depuis des dizaines d'années. Tout comme les CPUs des ordinateurs personnels, les performances des processeurs de ces supercalculateurs ont augmenté de façon vertigineuse. Cependant, outre les améliorations phénoménales de leurs vitesses, les constructeurs de ces machines ont également obtenu des gains de performances massives en multipliant le nombre de leurs processeurs. Ces améliorations apportées aux supercalculateurs

suggèrent une excellente question dans la recherche de puissance supplémentaire des ordinateurs personnels. Au lieu de se contenter d'augmenter les performances d'un unique cœur de traitement, pourquoi ne pas en mettre plusieurs ? De cette façon, les ordinateurs personnels pourront continuer à améliorer leurs performances sans devoir continuer à augmenter leurs fréquence d'horloge.

D'où la meilleure solution trouvée est de placer plusieurs processeurs de basses fréquences sur une même puce, ces processeurs sont appelés cœurs, et l'ensemble des cœurs d'une même puce sont appelés processeurs multi-cœur. Cette technique présente de nombreux avantages en termes de performances. L'intégration de plusieurs cœurs sur la même puce peut améliorer la vitesse de communication entre ces derniers.

En 2005, les principaux fabricants de processeurs ont commencé à proposer des processeurs avec deux cœurs de calcul au lieu d'un seul pour les ordinateurs personnels, ce qui fait qu'avec deux additionneurs on pourra aller deux fois plus vite qu'avec un seul, si on les fait travailler simultanément ; cette répartition des traitements sur ces unités, s'appelle le parallélisme. Au cours des années suivantes, ils ont suivi ce développement avec la sortie d'unités centrales de trois, quatre, six et huit cœurs. Cette tendance a induit une évolution du marché de l'informatique personnel.
Aujourd'hui, il est quasiment impossible d'acheter un ordinateur doté d'un CPU avec un seul cœur, même les CPUs d'entrée de bas de gamme et les moins performants sont livrés avec d'au moins deux cœurs.

En 2010, les principaux fabricants de CPU ont déjà réussi à lancer Intel produit : un micro-processeur avec 128 cœurs. Ce qui confirme bien que le traitement parallèle n'est plus l'apanage des supercalculateurs et que l'ère de l'informatique parallèle est arrivée.

Pendant ce temps, le traitement graphique subissait une révolution importante. Les graphismes sur un PC ont été effectués par un contrôleur VGA. Un contrôleur VGA était simplement un contrôleur de mémoire et un générateur d'affichage connecté à certaines DRAM.
Dans les années 1990, la technologie des semi-conducteurs a suffisamment progressé pour que davantage de fonctions puissent être ajoutées au contrôleur VGA. En 1997, les contrôleurs VGA commençaient à incorporer des fonctions d'accélération tridimensionnelles (3D).

En 2000, le processeur graphique à puce unique intégrait presque tous les détails du pipeline graphique de station de travail de haut de gamme traditionnel et par

conséquent, méritait un nouveau nom au-delà du contrôleur VGA. Le terme GPU a été inventé pour indiquer que le périphérique graphique était devenu un processeur.

Au fil du temps, les GPUs sont devenus plus programmables. De plus, les calculs sont devenus plus précis, passant de l'arithmétique indexée à un nombre entier, à une précision à virgule flottante simple et récemment à une précision double en virgule flottante. Les GPUs sont devenus des processeurs programmables parallèlement massifs avec des centaines de cœurs et des milliers de threads.

Les GPUs et leurs pilotes associés implémentent les modèles OpenGL et DirectX du traitement graphique en 2001. OpenGL est un standard ouvert pour la programmation graphique 3D disponible pour la plupart des ordinateurs. DirectX est une série d'interfaces de programmation multimédia Microsoft, y compris Direct3D pour les graphiques 3D. Son but était de fournir une méthode standardisée, indépendante des plate-formes, pour écrire des applications graphiques en 3D. Étant donné que ces interfaces de programmation d'application (API) ont un comportement bien défini, il est possible de générer une accélération matérielle efficace des fonctions de traitement graphique définies par les API.
C'est l'une des raisons (en plus de l'augmentation de la densité des appareils) qui explique pourquoi les nouveaux GPUs sont développés chaque année, ce qui double les performances de la génération précédente sur les applications existantes. Le doublage fréquent des performances du GPU permet de nouvelles applications qui n'étaient pas possibles auparavant de les résoudre.

Il n'y avait toujours pas moyen d'utiliser la performance de GPU qu'en passant par OpenGL ou DirectX. Ceci impliquait non seulement que les utilisateurs devaient continuer de traduire leurs traitements en problèmes graphiques, mais également qu'ils devaient encore écrire ces traitements dans un langage de shading comme GLSL d'OpenGL ou HLSL de Microsoft. Pour intéresser le maximum de développeurs, NVIDIA a donc choisi le langage C et C++ et lui a ajouté un nombre relativement restreint de mots clés, et a exploité certaines des fonctionnalités spéciales de l'architecture CUDA.
Quelques mois après le lancement de la GeForce 8800 GTX en 2007, NVIDIA mit à disposition un compilateur pour ce langage CUDA C, qui devient le premier langage conçu spécifiquement par un constructeur du GPU pour faciliter le développement de traitements généraux sur ses circuits graphiques. Désormais, les utilisateurs n'ont plus besoin d'apprendre les API graphiques, OpenGL ou DirectX, ni de transformer leurs problèmes en traitements graphiques. Le GPU moderne combine le traitement graphique et le calcul parallèle de manière novatrice pour permettre la mise en œuvre de nouveaux algorithmes graphiques et ouvre la voie à des applications de traitement

parallèle entièrement nouvelles sur des GPUs très performants.

Avec l'ajout de l'architecture CUDA aux capacités du GPU, il est désormais possible d'utiliser le GPU à la fois comme processeur graphique et comme processeur de calcul. L'architecture du processeur du GPU est exposée de deux manières : premièrement, en implémentant les API graphiques programmables, et deuxièmement, en tant que matrice de processeurs massivement parallèle programmable en C où C++ avec CUDA pour implémenter des applications dans différents domaines, pour faire des calculs scientifiques, de l'optimisation, résoudre des problèmes de transport et télécommunication etc.

Bien que le GPU soit sans doute le processeur le plus parallèle et le plus puissant sur un ordinateur personnel ou PC classique, il n'est certainement pas le seul processeur. Le CPU principalement est un compagnon au GPU massivement parallèle. Ensemble, ces deux types de processeurs constituent un système multiprocesseur hétérogène. Les meilleures performances pour de nombreuses applications proviennent de l'utilisation à la fois du CPU et du processeur graphique GPU. Les chapitres suivants vont nous aider à comprendre comment et quand répartir le travail entre ces deux processeurs. Nous présenterons de ce fait, propose nos propres implémentations GPU pour résoudre différentes variantes du VRP et du HLP.

## 2.3 Architectures système/GPU (vue matérielle)

Dans cette section, nous expliquons les architectures de systèmes hétérogènes couramment utilisées à nos jours et nous discutons aussi l'architecture interne de base du processeur graphique.

### 2.3.1 Architectures système hétérogène

Une architecture de système informatique hétérogène, utilisant un GPU et un CPU, peut être décrite à un haut niveau par deux caractéristiques principales : premièrement par le nombre de sous-systèmes fonctionnels, autrement dit le nombre de puces utilisés dans le système et leurs technologies d'interconnexion, deuxièmement, par les types de mémoires utilisées.

L'architecture des systèmes informatiques hétérogènes la plus simple et la plus utilisée, est caractérisée par un GPU et un CPU, chacun à leur sous-systèmes de mémoire respectifs.
Un processeur Intel et un GPU sont connectés souvent via un lien PCI-Express 2.0 à 16 voies pour fournir un taux de transfert maximal de 16 Go/s (crête de 8 Go/s dans chaque direction) pour transférer les données entre la mémoire du CPU et la mémoire du GPU. Aussi le GPU peut accéder à leur propre mémoire locale et à la mémoire du système CPU, en utilisant des adresses virtuelles.

Une autre architecture un peu coûteuse de ces systèmes, est une architecture avec mémoire unifiée (UMA - Uniform Memory Access), qui utilise uniquement la mémoire du CPU, en omettant la mémoire du GPU. Ces systèmes ont des GPUs à performances relativement faibles, car leurs performances obtenues sont limitées par la bande passante mémoire disponible du système et la latence accrue de l'accès à la mémoire.
Par contre, l'architecture système haute performance utilise plusieurs GPUs connectés entre eux, généralement deux à plusieurs fonctionnant en parallèle et connectés avec un CPU ou plusieurs CPUs pour rassembler les résultats finaux de chaque GPU. D'où un système hétérogène est bien équilibré s'il contient un nombre important de processeurs et s'il accède et utilise correctement les différents types de mémoires.
Le système connu sous le nom Titan à titre d'exemple et qui est classé deuxième parmi les 500 plus puissants supercalculateurs dans le monde est combiné du 299008 cœurs CPU et 18688 cartes GPUs.

### 2.3.2 Architecture de base du GPU

Les architectures GPU unifiées reposent sur un ensemble parallèle de nombreux processeurs programmables. Ils unifient le traitement de la géométrie, du pixel shader et le calcul parallèle sur les mêmes processeurs, contrairement aux anciens GPUs dotés de processeurs distincts dédiés à chaque type de traitement.

Comparés aux processeurs CPUs multi-cœurs, de nombreux GPUs de base ont un point de conception architecturale différent, centré sur l'exécution efficace de nombreux threads parallèles sur de nombreux cœurs de processeurs. En utilisant de nombreux noyaux (processeurs) plus simples et en organisant l'accès aux données par des groupes de threads. D'où l'architecture matérielle des cartes graphiques moderne, qui est généralement composée en deux grandes entités : Multiprocesseur et Mémoires.

Un tableau de processeurs GPU contient de nombreux cœurs de processeurs, généralement organisés en multiprocesseurs multithread. La figure 2.1 illustre un GPU avec un tableau de 112 cœurs de streaming processor (SP), organisé en 14 streaming multiprocessors (SMs) [106].

Chaque cœur de SP est multithread, gérant un ensemble de threads simultanés, le nombre de threads à exécuter dépend du type de GPU et sa version. Les processeurs se connectent à quatre partitions DRAM de 64 bits via un réseau d'interconnexion.

Chaque SM contient huit cœurs de SP, deux unités de fonction spéciales (SFU), des caches d'instructions et de constantes, une unité d'instructions multithread et une mémoire partagée (Shared memory). Il s'agit ici de l'architecture de base de Tesla implémentée par la carte graphique NVIDIA GeForce 8800. Elle présente une architecture dans laquelle les programmes graphiques traditionnels sont exécutés sur les SMs et sur leurs cœurs SP, ainsi les programmes informatiques parallèles exécutent sur les mêmes processeurs.

La figure 2.1 montre sept clusters de deux SM partageant une unité de texture et un cache de texture L1. L'unité de texture délivre les résultats filtrés au SM, avec un ensemble de coordonnées dans une texture. Comme les régions de filtrage du support se chevauchent souvent pour des demandes de texture successives, un petit mémoire cache de texture L1 est effectué pour réduire le nombre de demandes adressées au mémoire global. Un processeur se connecte aux caches de texture L2, aux mémoires DRAM externes (Global memory) et à la mémoire système du CPU via un réseau d'interconnexion du GPU.

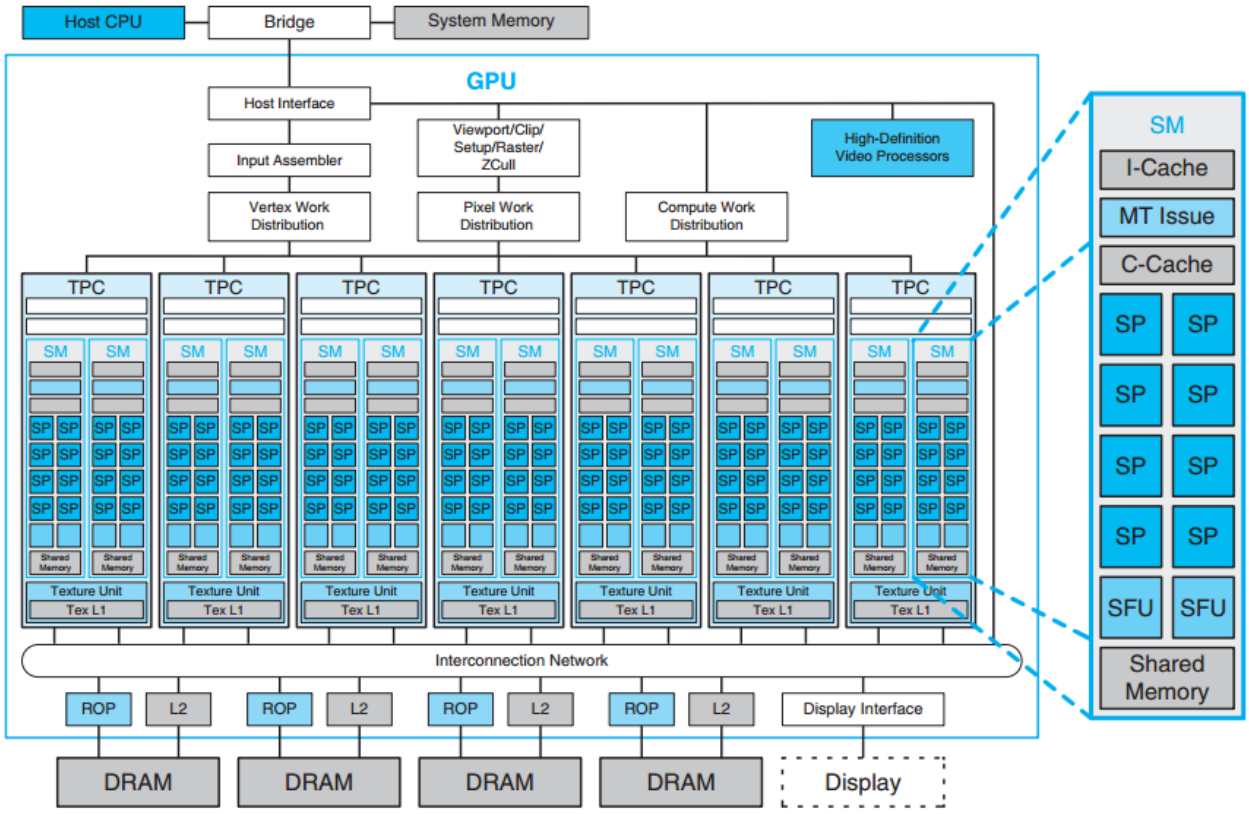



FIGURE 2.1 – Architecture GPU [106].

Le nombre de processeurs et le type de mémoires à utiliser peuvent être spécifiés par le développeur dans ses implémentations pour concevoir et évoluer des systèmes équilibrés. La performance des implémentations réalisée sous ces systèmes est influencée par ces choix. Afin d'exploiter l'architecture interne de base du GPU (matérielle), nous devons savoir comment ça fonctionne du côté programmation, l'architecture CUDA.

## 2.4 Architecture CUDA (vue logicielle)

CUDA (Compute Unifed Device Architecture) est un modèle de programmation parallèle évolutif et une plate-forme logicielle pour les GPUs proposé par NVIDIA, qui permettent au programmeur de contourner les interfaces graphiques, ou de faire du calcul scientifique et de les programmer simplement en C ou C++.

Le modèle de programmation CUDA a un style logiciel SPMD (single-program

multiple data), dans lequel un programmeur écrit un programme pour un thread qui est instancié et exécuté par plusieurs threads en parallèle avec différentes données. En fait, CUDA offre également la possibilité de programmer plusieurs cœurs de processeur, CUDA est donc un environnement permettant d'écrire des programmes parallèles dans système informatique hétérogène.

### 2.4.1 Organisation hiérarchique des processeurs

Au niveau du CPU et à cause du "toolkit" développé par NVIDIA, nous disposons des fonctions permettant de contrôler le GPU. Grâce à ces fonctions, le CPU est capable de démarrer l'exécution du code sur le GPU. Ce code qui est appelé depuis le CPU et lancé sur le GPU est appelé kernel.

L'architecture d'un GPU moderne est basée sur une grille programmable. Les différents fils d'exécution du kernel sont appelés threads. Les threads sont les unités de base de calcul que l'on peut lancer en parallèle sur le GPU lors de l'appel du kernel, son nombre est impressionnant, on peut atteindre les 8 millions sur une simple carte GeForce 9300 GE ; ces threads sont divisés en blocs. Les nombres de blocs et de threads par bloc sont spécifiés à l'appel du kernel.

Chaque bloc est affecté à un SM, d'où les threads d'un même bloc accèdent à la mémoire partagée (sherd memory) commune très rapide, la position d'un thread dans un bloc est repérée par ses coordonnées sur 2 ou 3 dimensions. Les blocs ne trouvant aucun multiprocesseur (SM) libre, sont mis en fils d'attente. Par exemple, si une grille de 6 blocs sera exécutée sur deux types de GPU : GPU1 avec 2 SM et GPU2 avec 3 SM. Chaque SM du GPU1 exécute séquentiellement 3 blocs et chaque SM du GPU2 exécute séquentiellement 2 blocs, voir figure 2.2. Un bloc est décroché d'un SM dès que tous ses threads ont fini leurs exécutions.

Les threads d'un bloc sont divisés en groupes appelés warps. Le nombre de thread résidant dans un SM peut être supérieur au nombre de SPs. Les threads sont alors ordonnancés temporellement entre les SPs. Le seul circuit d'instruction "fetch" d'un SM est celui qui gère l'exécution des threads sur différents SPs. Si seuls 8 SPs par SM sont disponibles, alors l'exécution des 32 instructions courantes d'un warp prend 4 cycles. Dans le cas de l'architecture Fermi, avec ses 32 SPs par SM, un seul cycle est nécessaire.

En résumé, le lancement d'un kernel sur le GPU crée une grille. Cette dernière peut être présentée sur 1, 2 ou 3 dimensions, et est divisée en blocs qui peuvent aussi être présenté sur 1, 2 ou 3 dimensions. Le nombre de bloc par grille et le nombre de

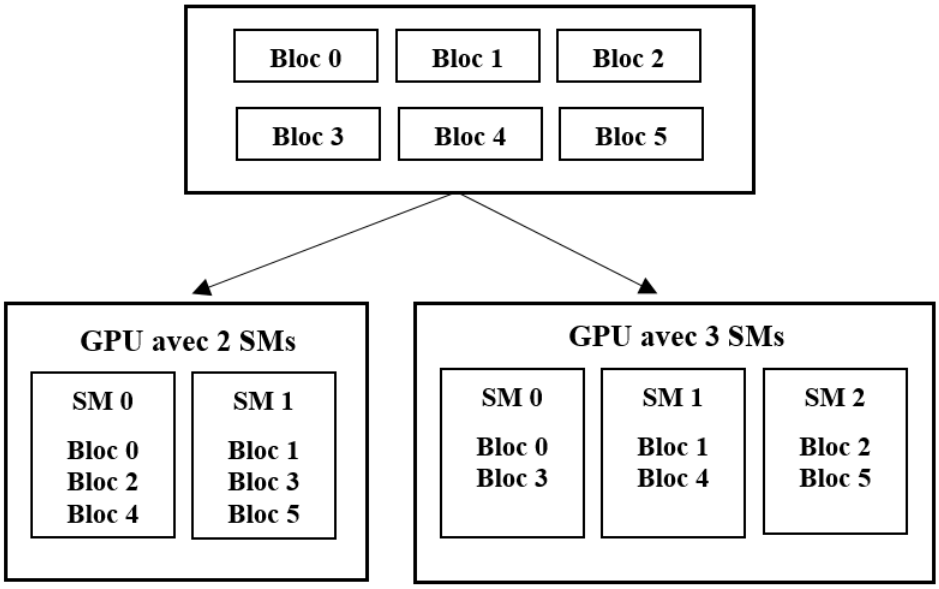



FIGURE 2.2 – Exemple de distributions des blocs aux SMs.

thread par bloc sont spécifiés à l'appel du kernel dans le côté du CPU.

### 2.4.2 Organisation hiérarchique des types de mémoire

Comme expliqué auparavant et afin d'exploiter la performance des GPUs sous l'architecture CUDA, il faut aussi connaitre les différents types de mémoires.

Selon le type considéré, les règles d'accès, la vitesse d'accès et la taille sont différentes. Ainsi pour les mémoires qui sont partagées par tous les threads (mémoire globale et mémoire constante) l'accès se fait en séquentiel par des warps.

Les types de mémoires les plus importants sont la mémoire partagée, la mémoire globale, la mémoire constante et les textures.

**La mémoire partagée : "Shared Memory" :** tous les threads d'un SM partagent cette mémoire et l'utilisent pour communiquer entre eux. L'accès est très rapide (2 cycles) car cette mémoire est sur la puce. En revanche, la mémoire partagée est de petite taille, actuellement de 48 Ko par SM, et les données qui y sont stockées ne sont valides que pour la durée de vie du bloc en cours d'exécution. Si plusieurs blocs sont assignés au même SM, ils sont exécutés indépendamment (mais séquentiellement) par le SM et la mémoire partagée est entièrement allouée au bloc

en cours d'exécution.

**La mémoire globale : "Global Memory" :** elle est partagée par tous les threads d'une application entière et persiste pendant les appels du bloc, tout au long de la vie de l'application. Elle est généralement beaucoup plus volumineuse que la mémoire partagée (1 à 4 Go) et accessible à partir de l'hôte. En revanche, la mémoire globale est hors puce et très lente, prenant des centaines de cycles d'horloge par accès (400 cycles).

**La mémoire constante : "Constante Memory" :** elle est utilisée pour stocker des données constantes. La mémoire constante est petite (64 Ko pour le GPU Quadro K2000) et l'accès est rapide par rapport à la mémoire globale (environ 20 cycles).

Les accès à la mémoire globale, partagée et constante se font via des demi-warps. Il est donc important d'éviter de stocker, dans la mémoire globale, des données qui doivent être consultées plusieurs fois par certains threads d'un bloc. Il est préférable de les copier dans la mémoire partagée (éventuellement dans une mémoire constante).

### 2.4.3 "deviceQuery" : pour connaître les détails du GPU

Le programme "deviceQuery" fournit par la SDK sample de CUDA, utilise des variables propres de CUDA pour donner directement un ensemble d'informations sur la carte graphique GPU. Cela nous permet notamment de connaître le nombre maximal de threads par blocs, la taille de différentes mémoires et la puissance de la carte graphique.

La figure 2.3 est le résultat de l'exécution de deviceQuery de notre carte Nvidia Quadro K2000. Les lignes suivantes sont intéressantes pour notre étude :

- Maximum number of threads per block : nombre maximum de threads par bloc (1024).
- Maximum dimension size of a thread block : taille maximale pour chaque dimension du bloc (nombre de threads dans chaque bloc) ($1024 \times 1024 \times 64$).
- Maximum dimension size of a grid size : taille maximale pour chaque dimension de la grid (nombre de blocs dans chaque grille) ($2147483647 \times 65535 \times 65535$).
- Total amount of global memory : capacité mémoire de la carte graphique (2G).
- Total amount of constant memory : capacité de la mémoire constante (64Ko).
- Total amount of shared memory per block : espace mémoire pour les échanges d'informations inter-bloc (taille du mémoire partagée) (48Ko).

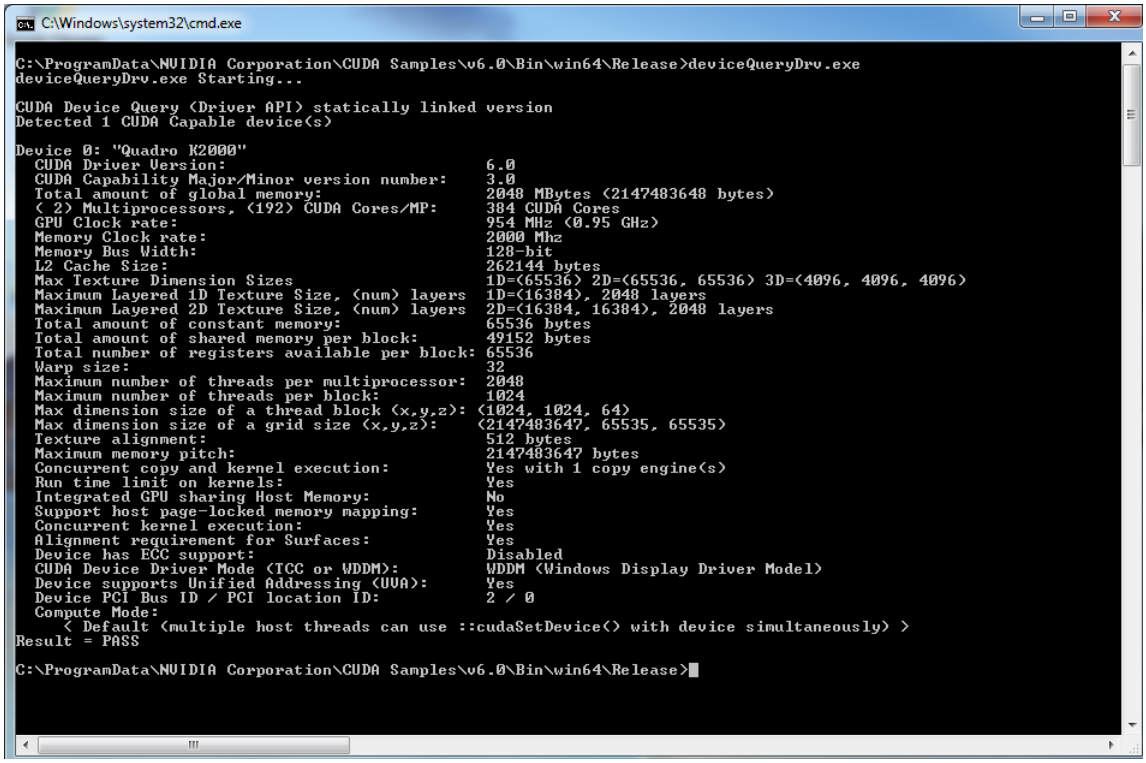



FIGURE 2.3 – Architecture de la carte Nvidia Quadro K2000 avec deviceQuery.

### 2.4.4 Communication entre CPU et GPU

Dans la plupart des cas, la carte GPU se connecte avec la carte mère sur le bus PCI Express et possède sa propre mémoire, c'est pour cela que des transferts de données doivent donc être effectués de la mémoire centrale (RAM CPU) vers la mémoire de GPU (RAM GPU). Ces transferts se font avec des dispositifs de copie.

Pour exécuter un programme sur GPU, la mémoire est allouée sur le CPU et sur le GPU, et on dispose d'instructions de transfert de CPU au GPU et de GPU au CPU. Les données sont chargées sur la RAM de CPU puis elles sont copiées sur la RAM de GPU, après la terminaison de traitement par le GPU, ce dernier retourne les résultats au CPU avec les mêmes mécanismes de copie. Ces transferts mémoire entre CPU et GPU peuvent être très couteux en terme de temps d'exécution [147].

### 2.4.5 Langage CUDA

La déclaration du kernel en langage CUDA se fait de la même manière du langage C, à l'exception faite du préfixe *__global__*, qui est utilisé pour spécifier que la fonction kernel sera appelée par le CPU et exécutée dans le GPU. Il y a aussi le préfixe *__device__*, qui est utilisé pour spécifier qu'une fonction sera appelée par le GPU et exécutée sur le GPU.

La définition de la fonction kernel est effectuée en utilisant la syntaxe suivante :
*__global__* void Kernel ($Arg_0$, $Arg_1$, ...)
{
Le code exécuter par les threads ... ;
}

L'appel au Kernel se fait en utilisant des chevrons comme suit :
Kernel $<<<$ $nombreDeBlocs, ThreadParBloc$ $>>>$ ($Arg_0$, $Arg_1$, ...) ;

La partie $<<<$ $nombreDeBlocs, ThreadParBloc$ $>>>$, s'agit de paramètres passés au système d'exécution afin de lui indiquer comment exécuter le kernel. En fait, le premier de ces deux nombres représente le nombre de blocs que nous voulons faire exécuter par le GPU. Tandis que le deuxième paramètre est le nombre de threads ou le nombre de copies dans chaque bloc. Si, par exemple, nous exécutions un Kernel avec $Kernel <<< 2, 1 >>>$, le système d'exécution créera deux blocs chacun lance un thread qui seront exécutés en parallèle, par contre si par exemple on lance un Kernel avec $Kernel <<< 1, 2 >>>$, le système d'exécution créera un seul bloc qui contient ou lance deux threads qui seront exécutés en parallèle.

Les variables *nombreDeBlocs* et *ThreadParBloc* sont du type *dim3*, type de 3 dimensions et spécifiques à CUDA. Les variables de ce type possèdent trois champs. Si seul le premier champ est affecté, la grille sera à une dimension. Si les deux premiers champs sont affectés, la grille sera de dimension deux, et finalement, si les trois champs sont affectés, la grille sera à trois dimensions.

Au cours de l'appel du kernel par le système d'exécution, dans le code kernel, des variables spéciales appelées "built-in" variables seront fixées pour chaque thread. Ces built-in variables sont :

- *gridDim :* contient la taille de la grille (le nombre de blocs).
- *blockDim :* contient la taille du bloc courent (le nombre de threads).
- *blockIdx :* contient l'identifiant du bloc courant.
- *threadIdx :* contient l'identifiant du thread courent.

Notons que toutes ces variables sont des structures contenant les champs x, y et z pour sélectionner la dimension.

Comme nous avons indiqué dans la partie communication entre CPU et GPU, les données sont échangées entre CPU et GPU à l'aide de *cudaMemcpy*, en utilisant soit l'option *cudaMemcpyHostToDevice* ou *cudaMemcpyDeviceToHost*.

## 2.5 Applications de CUDA

Depuis son lancement au début de 2007, de nombreuses entreprises et sociétés ont connu un grand succès en choisissant de créer des applications avec CUDA C, en utilisant des systèmes hétérogènes basés sur des CPU et des GPU. Les avantages en termes de performances sont souvent plus importants par rapport à celles des implémentations traditionnelles précédentes implémentées seulement sur les CPUs traditionnelles d'une manière séquentielle. Les éléments suivants ne représentent que quelques exemples d'utilisation réussis de CUDA C (Architecture CUDA).

- **Imagerie médicale :** Les algorithmes de traitement d'image médicale requièrent de grandes capacités de calcul. Avec l'accélération GPU, de nombreuses applications peuvent atteindre des performances comme celles de TechniScan [66].

- **Calculs financiers :** Les accélérateurs GPU offrent aux établissements de services financiers un avantage compétitif significatif en accélérant des applications de calculs financiers tel que les simulations de Monte Carlo. Le calcul des prix et des risques associés aux options les plus complexes et aux produits dérivés OTC s'effectue désormais en quelques secondes au lieu de plusieurs heures.

- **Science des données, analyse et base de données :** De plus en plus, des sociétés ont recours aux GPU pour analyser de gros volumes de données (Big Data) et ainsi prendre de meilleures décisions commerciales, en temps réel [34].

- **Apprentissage automatique :** Les chercheurs des domaines industriels et universitaires exploitent l'accélération GPU pour optimiser leurs procédures d'apprentissage et faire des découvertes révolutionnaires dans une grande variété de champs d'applications incluant la classification des images, l'analyse vidéo, la reconnaissance vocale et le traitement automatique des langues [89, 42].

*"Grâce aux GPU, les messages vocaux pré-enregistrés et les contenus multimédias peuvent êtes traités plus rapidement. Nous exécutons aujourd'hui nos algorithmes de reconnaissance jusqu'à 33 fois plus vite qu'avec une simple configuration CPU".* - Professeur Ian Lane, Université Carnegie-Mellon.

- **Chimie numérique :** Les processeurs graphiques GPU accélèrent aussi les procédures de chimie numérique et permettent aux chercheurs de faire de nouvelles découvertes. Par rapport aux CPU, les GPU permettent d'exécuter jusqu'à 5 fois plus vite les applications de dynamique moléculaire, de chimie quantique, de visualisation et de docking moléculaire [9].

- **Sciences météorologique et atmosphériques modélisation océanique et spatiale :** plusieurs applications de mécanique des fluides numériques exploitent la puissance des GPUs. Par exemple, des outils de modélisation météorologique et océanique tels que le modèle WRF (Weather Research and Forecasting model) et des simulations de tsunami s'exécutent maintenant beaucoup plus rapidement [100].

## 2.6 Conclusion

Dans ce chapitre, nous avons essayé de faire un parcours général sur l'historique de l'évolution des unités de traitement central CPU et des unités de traitement graphique GPU. Par ailleurs, nous nous sommes focalisé à expliquer à la fois le volet matériel incarné par l'architecture de base de GPU, et le volet dit "logiciel" qui est représenté par l'architecture CUDA. Dans le même contexte, nous avons essayé de mettre en exergue plusieurs éléments susceptibles de nous aider à apprendre aussi bien l'organisation hiérarchique des processeurs que les types de mémoires disponible sur un GPU, les performances d'un GPU, la communication entre le CPU et le GPU, ainsi que le langage CUDA. Et pour conclure, nous avons passé en revue quelques applications ayant connu un grand succès en choisissant d'adopter les GPUs et l'architecture CUDA, avant de s'en inspirer pour créer nos propres et différentes implémentations sur GPU, que nous allons présenter dans le chapitre suivant.

Chapitre 3

# Résolution des problèmes de tournées de véhicules sur GPU

## 3.1 Introduction

La plupart des méthodes de recherche utilisées dans la littérature pour résoudre les différentes variantes du VRP sont basées, sur des méthodes heuristiques ou des métaheuristiques pour répondre aux besoins de certaines entreprises de la logistique. Les applications réelles confrontées dans le domaine de la logistique sont aujourd'hui considérées plus importantes et peuvent compter plus de centaines de clients, et plus de vingt dépôts avec de nombreux véhicules. L'utilisation des simples CPU, avec des méthodes séquentielles ne peut pas répondre à ce besoin ; pour cette raison un nombre important de chercheurs ont orienté leurs recherches vers le parallélisme, afin de profiter de la puissance de calcul.

Dans la prochaine section, nous allons faire une revue de la littérature, où nous allons présenter certains travaux ayant traité des problèmes en exploitant des ressources du calcul haute performance, en utilisant les GPUs pour résoudre différentes variantes du VRP.

Comme déjà mentionné dans le premier chapitre, l'objectif de cette thèse est d'exploiter les nouvelles ressources du calcul haute performance GPU, afin de résoudre des variantes du VRP avec une grande qualité de solution et dans un temps raisonnable. Dans ce chapitre, nous allons présenter quatre implémentations sur GPU proposées pour résoudre des variantes du VRP. Dans la section 3.3, nous allons présenter une nouvelle implémentation parallèle sur GPU d'une heuristique basée sur l'algorithme de Clarke et Wright (CW) pour résoudre le VRP unique et multi dépôts.

La section 3.4 est réservée à l'étude de l'implémentation parallèle sur GPU pour la variante multi capacités VRP. Enfin, la section 3.5 et la section 3.6 sont consacrées à l'étude des implémentations sur GPU pour la variante dynamique VRP. La section 3.5 présente la conception et l'implémentation sur GPU d'une méthode simple (heuristique) qui insère, rapidement et efficacement, des requêtes dynamiques dans des tournées déjà planifiées avec une ré-optimisation continue, tandis que la section 3.6 présente, un algorithme génétique (métaheuristique) basé sur le GPU pour la résolution de grandes instances DVRP avec une ré-optimisation périodique. Finalement, la section 3.7 constitue une conclusion pour l'ensemble de ces travaux.

## 3.2 Revue de la littérature

Les recherches les plus récentes dans le domaine du calcul haute performance et du parallélisme sont presque toutes focalisées sur l'utilisation des GPUs ; et à cause de sa nouveauté seulement quelques implémentations qui résolvent le VRP sont déjà disponibles dans la littérature. On peut citer dans ce sens :

Cekmez et al. [35] présentent une implémentation de l'algorithme génétique sur GPU pour trouver une solution du TSP, aussi Fosin et al. [63] ont proposé des opérateurs de recherche local pour résoudre le TSP sur GPU.

Pour le VRP classique, on peut citer quelques travaux de Talbi et al., qui ont proposé une méthode génétique sur GPU [98], ainsi qu'une méthode de recherche locale [132]. Cette dernière a été aussi proposée par des contraintes sur GPU par Arbelaez et al. [10]. Diego et al. [51] proposent une stratégie de parallélisation pour résoudre le VRP sur GPU avec l'heuristique des colonies de fourmis.

D'autres approches sont conçues et présentées dans la littérature pour résoudre différentes variantes du VRP sur GPU, comme Li et al [95] qui proposent un algorithme de recuit simulé en parallèle basé sur GPU, pour résoudre un VRP à grande échelle avec des fenêtres de temps. Uthayopas et al. [143] ont développé un logiciel rapide pour résoudre le problème de collecte et de livraison avec des fenêtres de temps en utilisant un cluster GPU. Szymon et Dominik [130] résolvent l'optimisation discrète multi-critères de VRP avec contrainte de distance, en utilisant la recherche tabou parallèle sur GPU. Campeotto et al. [31] présentent une implémentation sur GPU très intéressante de la recherche de grand voisinage pour résoudre des problèmes d'optimisation avec des contraintes.

## 3.3 Problèmes de tournées de véhicules unique / multi dépôt(s)

### 3.3.1 Description de l'approche

Dans cette section, nous présentons une nouvelle implémentation parallèle sur GPU, d'une heuristique basée sur l'algorithme de Clarke et Wright (CW) pour résoudre le problème de tournées de véhicules unique et multi dépôts. À notre connaissance, il s'agit de la première implémentation sur GPU, d'une telle classe d'analyses heuristiques qui résolvent le VRP sur GPU.

En effet, l'implémentation calcule en parallèle une solution initiale en une seule étape puis, de manière itérative, elle améliore les coûts de chaque paire de tournées voisines en parallèle. Cette approche consiste tout d'abord, à partitionner l'ensemble des clients en secteurs en fonction de la capacité du véhicule pour avoir un ensemble de tournées sous la forme d'un ensemble de TSPs. Chaque tournée sera ensuite construite en parallèle indépendamment des autres tournées en utilisant l'algorithme CW. Enfin, chaque paire de tournées voisines seront améliorés en parallèle [24].

Cette section se décline comme suite : nous présentons la technique de partitionnement en secteurs, l'algorithme parallèle, son implémentation sur GPU et nos résultats expérimentaux. Les résultats expérimentaux obtenus sous CUDA montrent que l'implémentation proposée exploite efficacement le parallélisme du GPU. Nous adaptons enfin notre approche dans l'objectif de résoudre la variante multi dépôts [23].

### 3.3.2 Partitionnement en secteurs

Chaque client est définie par ses coordonnées polaires $(\theta_i,\ \rho_i)$ avec le dépôt en (0,0). Les clients sont numérotées en fonction de leur angle par rapport à l'axe des abscisses, l'angle $\theta_i \leq \theta_{i+1}$, pour chaque $1 \leq i \leq n$, de sorte que le client 1 soit le client dont l'angle $(\theta = \theta_1)$ est le plus petit par rapport à l'axe des abscisses. Dans le cas où $\theta_i = \theta_{i+1}$, alors les clients sont numérotés en augmentant le rayon $\rho_i$. Une étude plus générale de la partition en secteurs peut-être trouvée dans [82].

La figure 3.1 montre un exemple de numérotation. Dans cet exemple, on numérote les clients en fonction de son angle par rapport à l'axe des abscisses, les clients 4 et 5 ont le même angle $\theta$ et le client 4 a le numéro le plus petit car il est le plus proche du dépôt. Après cette numérotation, nous divisons l'ensemble des clients en secteurs où la demande de chaque secteur est inférieure ou égale à la capacité C du véhicule.

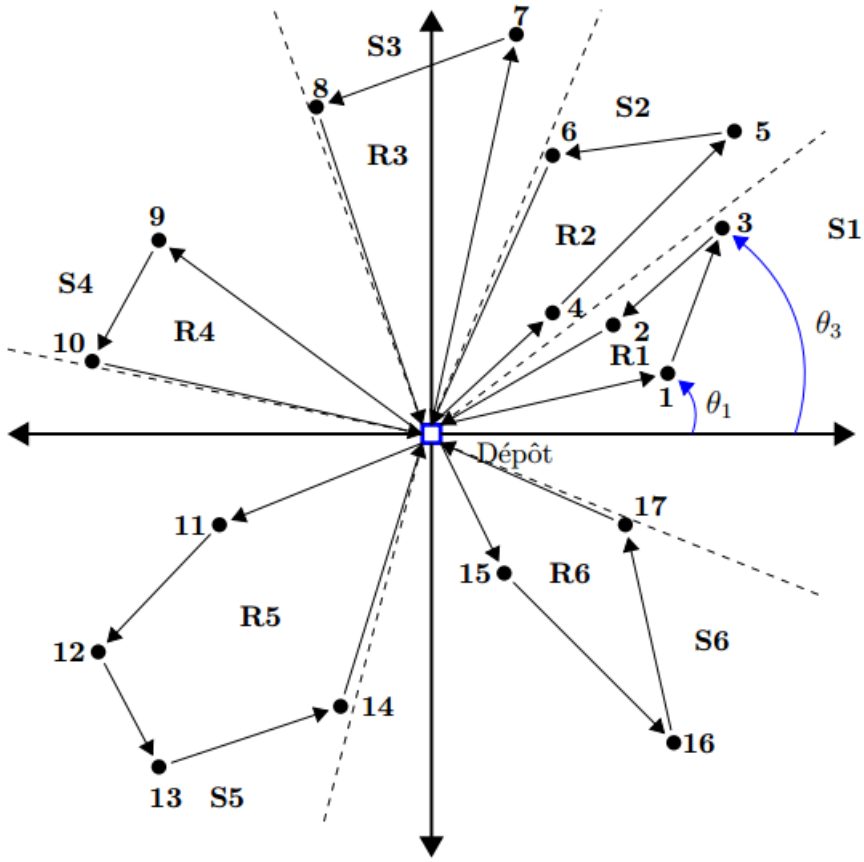



FIGURE 3.1 – Exemple de numérotation et de partitionnement en secteurs.

Soient $S_1, S_2, ..., S_m$ les secteurs, tel que $S_1 = \{0, 1, ..., r_1\}$, ..., $S_i = \{0, r_{i-1}+1, ..., r_i\}$. Nous déterminons le secteur $S_i$ comme l'ensemble des clients $\{0, r_{i-1} + 1, ..., r_i\}$ où $r_i$ est l'entier unique satisfaisant : $\sum_{k=r_{i-1}+1}^{r_i} d_k \leq C < \sum_{k=r_{i-1}+1}^{r_i+1} d_k$ pour $1 \leq i < m$ avec $r_0 = 0$ et $\sum_{k=r_{m-1}+1}^{r_m} d_k \leq C$ pour $i = m$. Avec, $s_i = r_i - r_{i-1}$ est le nombre de clients du secteur $S_i$. Chaque secteur est alors traité comme un TSP indépendant et résolu par un processus du TSP qui produit une seule tournée. Pour cette étude, nous utilisons l'heuristique CW de Clarke et Wright [41] pour construire les tournées indépendamment. Il est clair que cette partition initiale en secteurs a une influence sur la solution finale obtenue pour le VRP global.

La figure 3.1 montre un exemple de partition des clients en 6 secteurs : $S_1 = \{1, 2, 3\}$, $S_2 = \{4, 5, 6\}$, $S_3 = \{7, 8\}$, $S_4 = \{9, 10\}$, $S_5 = \{11, 12, 13, 14\}$ et $S_6 = \{15, 16, 17\}$. Pour cet exemple, nous avons supposé que : $d_1 + d_2 + d_3 \leq C < d_1 + d_2 + d_3 + d_4$, $d_4 + d_5 + d_6 \leq C < d_4 + d_5 + d_6 + d_7$, $d_7 + d_8 \leq C < d_7 + d_8 + d_9$,

$d_9 + d_{10} \leq C < d_9 + d_{10} + d_{11}$, $d_{11} + d_{12} + d_{13} + d_{14} \leq C < d_{11} + d_{12} + d_{13} + d_{14} + d_{15}$ et $d_{15} + d_{16} + d_{17} \leq C$.

### 3.3.3 Algorithme parallèle proposé

Chaque secteur est traité comme un TSP indépendant résolu par une seule tournée obtenue par le processus CW. Nous construisons ces tournées en parallèle, la tournée $R_i$ pour le secteur $S_i$, $i = 1, ..., m$, en utilisant l'algorithme CW pour trouver le meilleur ordre des clients de chaque tournée $R_i$. $m$ est le nombre de secteurs obtenus et c'est la limite supérieure du nombre de véhicules utilisés pour résoudre ce problème.

Soit $CW(S)$ la fonction qui construit la tournée R pour le secteur S. Cette fonction sera appelée en parallèle, pour que $S = S_1, ..., S_m$ produise m tournées $R_1, ..., R_m$. Les tournées $R_i$ et $R_{i+1}$ étant voisines, ($R_1$ et $R_m$ sont également voisines), les déplacements des clients de $R_i$ à $R_{i+1}$ ou de $R_{i+1}$ à $R_i$ pourraient réduire le coût de ces deux tournées. De même, les permutations des clients entre $R_i$ et $R_{i+1}$ pourraient réduire le coût de ces deux tournées. Ce processus de déplacement / permutation des clients entre $R_i$ et $R_{i+1}$, $i = 1, ..., m$, est itéré afin de réduire le coût global du VRP. Ces opérations se font entre paires ($R_1$, $R_2$), ($R_3$, $R_4$), ..., ($R_{m-2}$, $R_{m-1}$) en parallèle. Supposons que m soit impair alors que ($R_2$, $R_3$), ($R_4$, $R_5$), ..., ($R_{m-1}$, $R_m$) sont aussi des paires de tournée voisines, nous appliquons le même processus aux paires ($R_2$, $R_3$), ($R_4$, $R_5$), ..., ($R_{m-1}$, $R_m$). Enfin, nous appliquons alternativement ces deux étapes afin de réduire le coût global du VRP en réduisant la somme des coûts de chaque paire de tournées voisines.

#### Amélioration de la solution VRP

Le concept k-opt [92, 114] peut être appliqué à des ensembles de k tournées en supprimant les clients d'une tournée et en les insérant dans une autre pour minimiser la distance parcourue. Dans cette implémentation, nous faisons cela entre des paires de tournées adjacentes $R_i$ et $R_{i+1}$ et nous définissons deux fonctions de base *move()* qui déplace un client et *swap()* qui permutent deux clients. Nous appelons *Opt(R,R')* la fonction qui consiste à déplacer ou à permuter des clients à partir des tournées R et R'.

La fonction $Improve(VRP)$ (ou $Improve(R_1, ..., R_m)$) qui améliore la solution initiale $(R_1, ..., R_m)$, est définie comme suit :

**Algorithme 1** : Improve(VRP) = Improve($R_1, ..., R_m$) [24]

**si** m est impair **alors**
  **tant que** il y a une amélioration du coût global **faire**
    1. en parallèle Opt($R_1$, $R_2$), Opt($R_3$, $R_4$), ..., Opt($R_{m-2}$, $R_{m-1}$).
    2. en parallèle Opt($R_2$, $R_3$), Opt($R_4$, $R_5$), ..., Opt($R_{m-1}$, $R_m$).
  **fin tant que**
**fin si**
**si** m est pair **alors**
  **tant que** il y a une amélioration du coût global **faire**
    1. en parallèle Opt($R_1$, $R_2$), Opt($R_3$, $R_4$), ..., Opt($R_{m-1}$, $R_m$).
    2. en parallèle Opt($R_2$, $R_3$), Opt($R_4$, $R_5$), ..., Opt($R_{m-2}$, $R_{m-1}$).
  **fin tant que**
**fin si**

Dans ce qui suit, nous précisons les deux opérations de la recherche locale, utilisées dans la fonction opt (fonctions move() et swap()) :
Soit $R = (u_1, ..., u_k)$, notons que la demande(R)= demande($u_1, ..., u_k$) = $d_{u_1}$ + ... + $d_{u_k}$, et coût(R)= coût($u_1, ..., u_k$) = $c_{u_1u_2}$, + ... + $c_{u_{k-1}u_k}$.
Un déplacement d'un client $u \in R$ vers $R'$ est effectué si l'opération est rentable, c-à-d si le coût total des nouvelles tournées est inférieur à l'ancien c-à-d :
coût($R - \{u\}$) + coût($R' + \{u\}$) < coût($R$) + coût($R'$) ; avec la vérification de la contrainte de capacité. La figure 3.2 montre comment le déplacement de $u$ de $R$ à $R'$ se réalise : (a) avant le déplacement et (b) après le déplacement. Cela revient à supprimer $u$ de $R$ ($R - \{u\}$) et l'insérer dans $R'$ ($R' + \{u\}$).

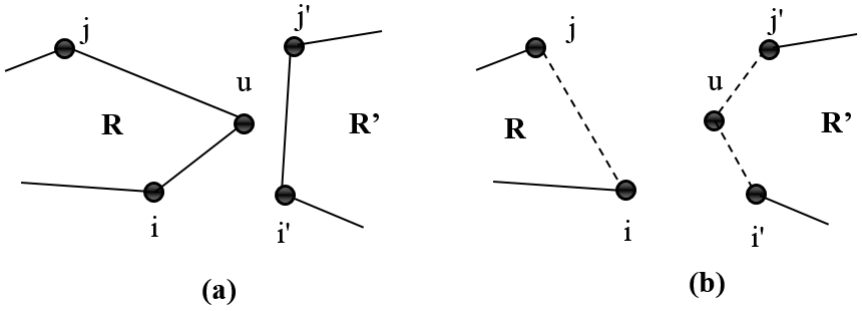



FIGURE 3.2 – Déplacement de $u$ de $R$ à $R'$.

Une permutation de $u \in R$ et de $u' \in R'$ est effectuée si l'opération est rentable, c-à-d si le coût total des nouvelles tournées est inférieur à l'ancien, c-à-d :
coût($R - \{u\} + \{u'\}$) + coût($R' - \{u'\} + \{u\}$) < coût($R$) + coût($R'$), avec la vérification de la contrainte de capacité. La figure 3.3 montre comment la permutation

de $(u, u') \in R \times R'$ est réalisé : (c) avant la permutation et (d) après la permutation.

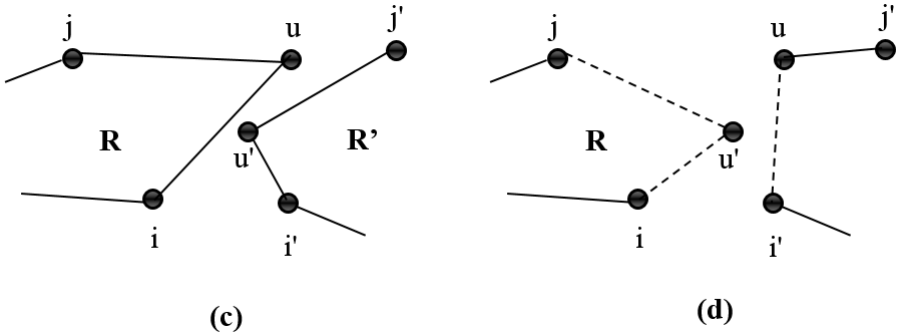



FIGURE 3.3 – Permutation de $(u, u') \in R \times R'$.

Enfin, on définit la fonction $SolveOneDepot(VRP)$ qui résout le VRP avec un unique dépôt, il se compose de deux étapes :

1. pour chaque i, $1 \leq i \leq m$, calculer en parallèle $R_i$=CW($S_i$).
2. lancer en parallèle Improve($R_1, ..., R_m$).

### 3.3.4 Implémentation sur GPU

#### Implémentation de CW() sur GPU

Soit $s_i$ le nombre de clients du secteur $S_i$, $1 \leq i \leq m$. Les données nécessaires pour calculer la tournée $R_i$, sont le bloc diagonal $s_i \times s_i$ de la matrice de distance $D$ composée des colonnes $r_i + 1, ..., r_{i+1}$. Les tournées $R_i = CW(S_i)$, pour $1 \leq i \leq m$ sont calculées en parallèle.
Chaque $R_i$ est calculée sur un bloc de $s_i \times s_i$. Comme, dans CUDA, nous ne pouvons définir que des blocs de threads de tailles uniformes, nous utilisons des blocs de $s \times s$ threads où $s = max(s_i)$, en impliquant que $n \leq sm$. Enfin, nous exécutons le kernel $CW(S_i)$ sur m blocs notés par $B_1, ..., B_m$, chacun avec $s \times s$ threads. En utilisant la syntaxe CUDA, les blocs, la grille et le kernel CW() sont définis comme suit :

```
dim3 dimBlock(s,s) ; dim3 dimGrid(m,1) ;
CW<<<dimGrid,dimBlock>>>(Si) ;
```

$S_i$ est représenté par la diagonale de bloc $s_i \times s_i$ de la matrice de distance $D$.

#### Amélioration de la solution initiale sur GPU

L'opération de base de cette fonction est $move(u)$ de $R$ vers $R'$ ou de $swap(u, u')$ entre $R$ et $R'$. Comme toutes ces opérations de base sont effectuées sur un bloc

de $s \times s$ threads, elles sont exécutées en une seule étape. Plus précisément, chaque $Opt(R_i, R_{i+1})$, $1 \leq i < m$, est exécuté sur un bloc $B_i$ avec la mémoire partagée $M_i$ et nécessite toutes les distances entre les clients $u$ et $u'$ des tournées $R_i$ et $R_{i+1}$, qui sont stockés dans les blocs $D_i$ et $D_{i+1}$ de la matrice D, comme indiqué sur la figure 3.4. Ainsi, $D_i$ et $D_{i+1}$ doivent être stockés dans la mémoire partagée $M_i$ du bloc $B_i$. Maintenant, si un client $u$ est déplacé de $R_i$ à $R_{i+1}$, alors nous devons mettre à jour les mémoires partagées $M_{i-1}$, $M_i$ et $M_{i+1}$ comme indiqué sur la figure 3.5. De même, un échange de $u$ et $u'$ entre $R_i$ et $R_{i+1}$ nécessite une mise à jour de $M_{i-1}$, $M_i$ et $M_{i+1}$, comme indiqué dans la figure 3.5.

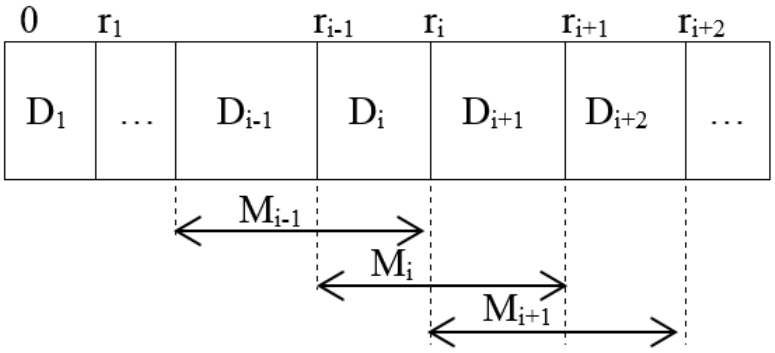



FIGURE 3.4 – Répartition des données sur la mémoire partagée.

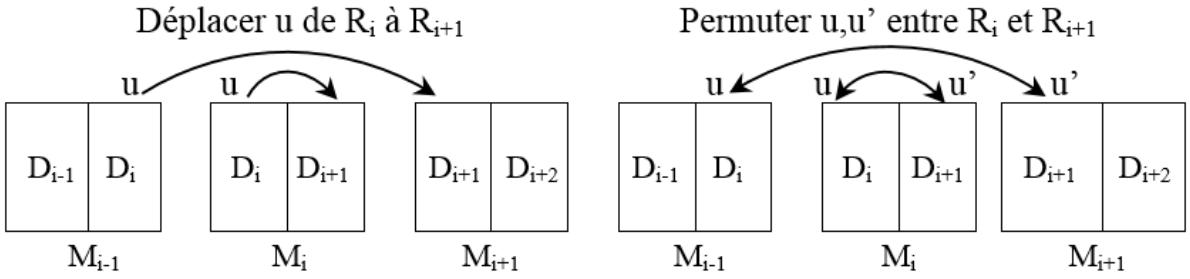



FIGURE 3.5 – Mise à jour de la mémoire partagée lors du déplacement / permutation.

En utilisant la syntaxe CUDA, les blocs, la grille et le kernel Opt() sont définis comme suit :

```
dim3 dimBlock (s, s) ; dim3 dimGrid (m, 1) ;
Opt <<< dimGrid, dimBlock >>> (Di, Di+1) ;
```

Enfin, le kernel $SolveOneDepot(VRP)$ est exécuté sur GPU en deux étapes séquentielles :

1. Pour tout i, $1 \leq i \leq m$, calculer en parallèle $R_i = CW(S_i)$ // sur $m$ blocs, chacun de $s \times s$ threads.
2. Lancer en parallèle Improve($R_1, ..., R_m$), // sur $m$ blocs chacun de $s \times s$ threads.

### 3.3.5 Résultats expérimentaux

Dans un premier temps, on va se focaliser sur la performance du calcul sur GPU. Cette partie montre l'évolution du temps d'exécution de l'implémentation présentée ci-dessus sur le CPU et sur le GPU. Nous avons utilisé Nvidia GeForce, 1 Go, avec un SM de 48 cœurs CUDA (1,17 GHz) et i3 avec 4 cœurs (2,4 GHz) en tant que CPU. L'accélération optimale, ou ce matériel, ne peut pas la dépasser est six. Nos algorithmes sont implémentés sous CUDA V.6.0.

Pour les tests, nous avons fixé la capacité C des véhicules ; les données ($c_{ij}$, $d_i$) sont générées aléatoirement pour un nombre de clients variant entre 5 et 120 (limité par la capacité de la mémoire partagée du GPU). Pour comparer les implémentations CPU et GPU, nous avons utilisé les mêmes données ($c_{ij}$, $d_i$ et $C$) et nous avons résolu le VRP respectivement sur CPU et sur GPU. Les transferts de données entre CPU et GPU ne sont effectués qu'au début du programme $SolvingOneDept()$ (avant les kernels $CW()$ et $Improve()$). Par conséquent, nos temps d'exécution n'incluent pas le temps de transfert de données entre le CPU et le GPU.

La figure 3.6, montre les temps d'exécution du CPU et du GPU pour le kernel $CW()$. Les temps obtenus montrent que le facteur d'accélération augmente en fonction du nombre de clients $n$ et qu'il est proche de 5 pour $n = 120$.
La figure 3.7, montre les temps d'exécution du CPU et du GPU pour le kernel $Improve()$. Ici, le facteur d'accélération est proche de 3 pour $n = 120$.
Finalement, la figure 3.8, montre les temps d'exécution du CPU et du GPU pour le problème VRP avec un unique dépôt. Les temps obtenus montrent que le facteur d'accélération est proche de 3. Il est clair que l'augmentation de la taille des données (nombre de clients) pourrait augmenter le facteur d'accélération.

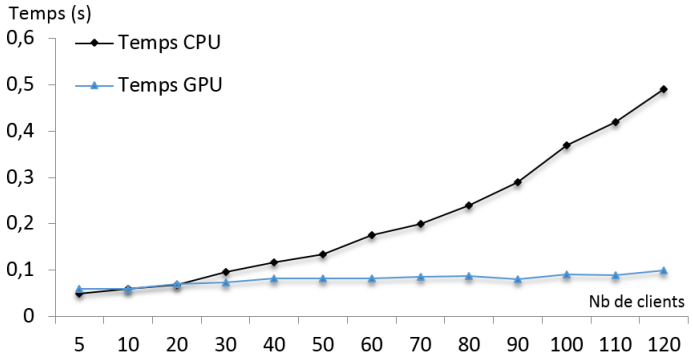



FIGURE 3.6 – Temps d'exécution de l'implémentation CW() sur CPU et sur GPU.

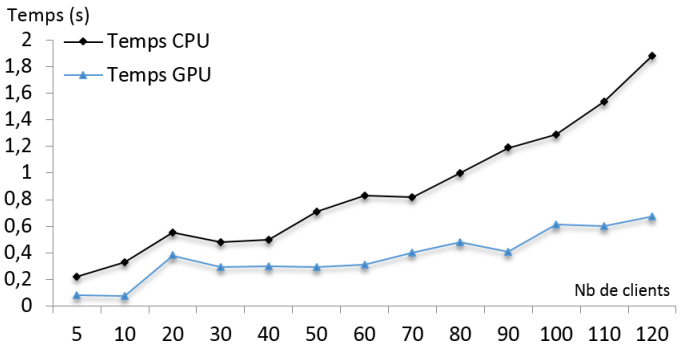



FIGURE 3.7 – Temps d'exécution de l'implémentation Improve() sur CPU et sur GPU.

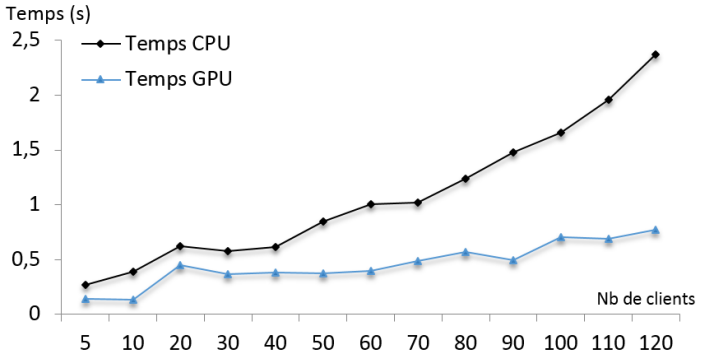



FIGURE 3.8 – Temps d'exécution de l'implémentation SolveOneDepot() sur CPU et sur GPU.

### 3.3.6 VRP avec multi dépôts

Dans le problème de tournées de véhicules avec unique dépôt, plusieurs véhicules quittent un seul dépôt et doivent retourner à cet endroit après avoir effectué les tournées assignées. Le VRP avec multi dépôts (MDVRP) est une généralisation du VRP dans lequel plusieurs véhicules partent de plusieurs dépôts et retournent à leur dépôt d'origine à la fin de leurs tournées. L'objectif de MDVRP est de minimiser la somme de toutes les longueurs des tournées.

Le MDVRP est défini comme suit : Étant donné les $d$ dépôts et $n$ clients qui doivent être desservies à partir de ces dépôts, le problème consiste à la fois :

(a) d'affecter chaque client à un dépôt

(b) de trouver l'ensemble des tournées qui résolvent le problème.

Dans notre cas, après avoir assigné des clients à des dépôts, nous résolvons en parallèle des VRPs de base indépendants, noté par $VRP_1, ..., VRP_d$, chacun associé à un dépôt : $VRP_k$ est associé au dépôt k.

- La première étape (a) consiste à affecter des clients à leurs dépôts les plus proches. Pour chaque client $u$, calculer la distance ($u$, $depot_k$), pour chaque k, $1 \leq k \leq d$ et assigner $u$ au dépôt le plus proche noté par $depot_j$ où :

$$distance(u, depot_j) = min_k(distance(u, depot_k)).$$

Nous obtenons donc des VRPs indépendants, $VRP_1, ..., VRP_d$. Chaque $VRP_k$ est représenté par sa matrice de distances notée par $H_k$. Cette étape d'affectation peut être facilement parallélisée.

- L'étape suivante (b) résout ces $d$ VRPs en appelant en parallèle le kernel $SolveOneDepot(VRP_k)$ pour $1 \leq k \leq d$. Comme nous l'avons présenté ci-dessus, la fonction $SolveOneDepot(VRP_k)$ est exécutée sur $m_k$ bloque chaque bloc de $s_k \times s_k$ thread, où $m_k$ est le nombre de secteurs (ou tournée) dans le $VRP_k$ et $s_k$ est le nombre maximum de clients dans les secteurs du $VRP_k$. Ainsi, le VRP multi dépôts est exécuté sur $d \times m$ blocs de $s \times s$ threads, où $m = max_k(m_k)$ et $s = max_k(s_k)$.

Avec la syntaxe CUDA, cela peut être défini comme suit :

```
dim3 dimBlock (s, s) ; dim3 dimGrid (d,m) ;
SolveOneDepot <<< dimGrid, dimBlock >>> (H_k) ;
```

La figure 3.9 montre un exemple avec 3 dépôts, 60 clients, l'affectation des clients aux dépôts et la solution obtenue. Les temps d'exécution en parallèle de $VRP_1$

(celui avec 24 clients), $VRP_2$ (avec 7 clients) et $VRP_3$ (avec 29 clients) sont $0,069s$, $0,026s$ et $0,083s$ ; le temps d'exécution de l'étape d'affectation est $0,028s$. Le temps d'exécution parallèle du MDVRP est donc égal à $0,11s$ ($\approx 0,083 + 0,028$).

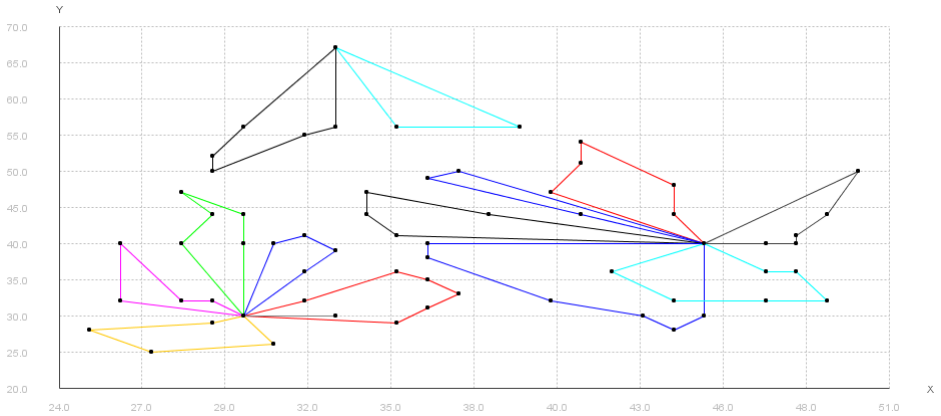



FIGURE 3.9 – Exemple d'une solution du MDVRP.

Nous citons qu'il existe d'autres approches pour attribuer des clients aux dépôts (voir [33, 72] pour plus de détails). En outre, une solution de mauvaise affectation entraînera des tournées d'un coût total plus élevé.

Avoir de bons résultats dépend non seulement de la qualité de l'algorithme d'affectation mais également de l'instance du problème à résoudre. En effet, pour notre approche, le temps d'exécution parallèle du MDVRP égale au temps maximum d'exécution parallèle des $VRP_k$ plus le temps de l'étape d'affectation. Ainsi, notre implémentation parallèle est efficace dans les instances où les clients sont uniformément situés autour des $d$ dépôts (c-à-d. produits $d$ dépôts chacun contient $\frac{n}{d}$ clients) et les temps parallèles d'exécution des $d$ VRPs sont identiques. Ceci est illustré par l'exemple de la figure 3.10.

L'instance montré dans la figure 3.10 est obtenu à partir de celui de la figure 3.9, en déplaçant le deuxième dépôt au plus bas. Nous obtenons 3 VRPs de 20 clients chacun. Les temps d'exécution parallèles obtenus sont $0,059s$ pour $VRP_1$ et $VRP_3$, $0,049s$ pour $VRP_2$, le temps de l'étape d'affectation est $0,028s$, d'où le temps d'exécution en parallèle du MDVRP est $0,087s$ ($\approx 0,059 + 0,028$), qui est bien réduit par rapport au temps de l'instance du problème présenté dans figure 3.9.

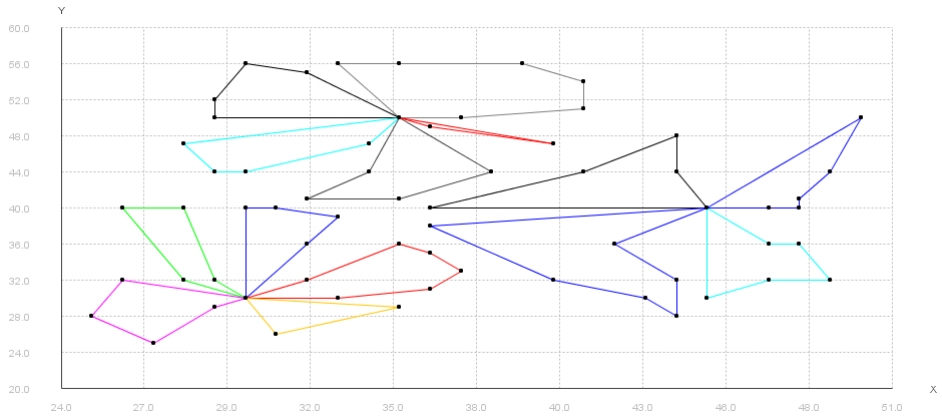



FIGURE 3.10 – Une autre instance du MDVRP de la figure 3.9.

## 3.4 Problèmes de tournées de véhicules multi capacités

### 3.4.1 Description de l'approche

Dans cette section, nous présentons une nouvelle implémentation parallèle sur GPU d'une heuristique pour résoudre le problème de tournées de véhicules multi capacités. Cette implémentation implique deux types de décisions : (1) la sélection d'un mélange de véhicules parmi les types de véhicules disponibles dans le dépôt, (2) en précisant la tournée de chaque véhicule sélectionné. L'implémentation GPU proposée calcule en parallèle une solution initiale (tournées), puis calcule en parallèle tous les cas possibles pour obtenir les véhicules les plus appropriés à utiliser. Enfin, on utilise la procédure pour améliorer le coût de toutes les paires de tournées voisines. Afin de mettre en évidence les performances de notre approche, nous avons comparé nos résultats avec les résultats obtenus de la part d'Ochi [107] et de Karagul [84]. Nous avons également résolu des instances générées aléatoirement. Les résultats expérimentaux obtenus par notre implémentation GPU, surpassent les autres implémentations en termes de temps d'exécution et de qualité des solutions. Cela signifie que notre algorithme est bien adapté à la puissance de calcul du GPU et que notre implémentation exploite efficacement la puissance du GPU. Cela nous permettra d'améliorer les solutions trouvées dans la littérature et aussi de résoudre des instances non résolues jusqu'à présent.

Dans la suite de cette section, nous présentons l'algorithme parallèle proposé pour résoudre le VRP multi capacités, son implémentation sur GPU et enfin, nos résultats expérimentaux [25].

### 3.4.2 Algorithme parallèle proposé

Dans cette section, nous sommes intéressés par le VRP multi capacités, où la flotte est composée de différents types de véhicules. Chaque véhicule de type $k$ a une capacité $C_k$ et est associé à un coût fixe $F_k$.
Une tournée réalisable $R^k = (u_1, u_2, ..., u_t)$, avec $u_1 = u_t = 0$ et $u_1, u_2, ..., u_t$ c'est un simple circuit dans le graphe $G$ et $k$ est le type du véhicule utilisé pour parcourir cette tournée. Notons que, la demande totale des clients visités dans $R^k$ ne dépasse pas la capacité du véhicule $C_k$. c-à-d $\sum_{h=2}^{t-1} d_{u_h} \leq C_k$. Le coût de la tournée $R^k$ est la somme des coûts des arcs constituant la tournée (distance parcourue) plus le coût fixe du véhicule qui lui est associé, c-à-d coût$(R^k) = F_k + \sum_{h=0}^{t-1} c_{u_h u_{h+1}}$. Comme dans Karagul [84] et dans Ochi [107], nous supposons que le nombre de véhicules de chaque type est illimité et que la capacité des véhicules et les types de coûts satisfont : $C_1 \leq C_2 \leq ... \leq C_m$ et $F_1 \leq F_2 \leq ... \leq F_m$.

Pour trouver une solution réalisable avec un coût efficace à ce problème, nous devons concevoir un algorithme qui teste le maximum de solutions possibles dans des délais raisonnables. Le concept et la puissance du GPU peuvent atteindre cet objectif. Nous proposons donc un algorithme qui teste un maximum de solutions réalisables en parallèle. Les trois principales étapes de notre algorithme parallèle sont les suivantes :

**Étape 1 :** Partitionner le VPR multi capacités en sous-problèmes indépendants (Ns secteurs). Chacun d'entre eux est considéré comme un problème de voyageur de commerce (TSP) et suit à la construction d'une solution initiale en parallèle, on peut utiliser n'importe quel algorithme qui résout le TSP. Pour cela, nous avons préféré utiliser l'algorithme de Clarke et Wright, noté CW déjà utilisé dans la résolution du VRP unique et multi dépôts dans la section précédente.

**Étape 2 :** Calculer toutes les solutions réalisables avec différents types de véhicules en parallèle pour chaque secteur. Autrement dit, pour chaque secteur construire une tournée ou plusieurs en testant tous les types de véhicules un par un en parallèle, puis sélectionner le type de véhicule avec le coût minimal pour visiter les clients de ce secteur.

**Étape 3 :** Améliorer la solution en utilisant le principe 2-Opt que nous avons adapté au concept de GPU.
Dans ce qui suit, nous détaillerons chacune de ces étapes :

#### Partitionnement du VRP à Ns secteurs

Les clients sont numérotés de la même manière comme dans l'implémentation du VRP classique, en augmentant l'angle $\theta_i \leq \theta_{i+1}$, pour chaque $1 \leq i \leq n$, de sorte que le client 1 soit le client dont l'angle $(\theta = \theta_1)$ est le plus petit par rapport à l'axe

des abscisses. Concernant le partitionnement des clients nous divisons l'ensemble des clients en secteurs, de telle sorte que la demande de chaque secteur soit inférieure à la capacité maximale $C_m$. Le choix de cette capacité n'est pas aléatoire, en effet avec cette capacité nous allons nous assurer que le secteur est saturé avec le maximum de clients.

Soient $S_1, S_2, ..., S_{Ns}$ les secteurs, et $S_i = \{0, r_{i-1} + 1, ..., r_i\}$ où $r_i$ est l'entier unique satisfaisant : $\sum_{k=r_{i-1}+1}^{r_i} d_k \leq C_m < \sum_{k=r_{i-1}+1}^{r_i+1} d_k$ pour $1 \leq i < Ns$ avec $r_0 = 0$ et $\sum_{k=r_{Ns-1}+1}^{r_{Ns}} d_k \leq C_m$ pour $i = Ns$.

Chaque $S_i$ peut être servi en une tournée avec un véhicule de capacité $C_m$, et $S_i \cup \{r_i+1\}$ ne peut pas être servi en une tournée par aucun véhicule. Par conséquent, nous pouvons utiliser n'importe quel algorithme de TSP pour construire une tournée qui sert $S_i$ au coût minimum. Dans notre cas, nous utilisons l'algorithme CW que nous avons adapté à l'implémentation GPU.

### Construction des tournées pour chaque secteur

Chaque secteur est un ensemble de clients où la somme de leurs demandes ne dépasse pas la capacité maximale des véhicules $C_m$. Nous avons utilisé l'algorithme CW, ce dernier sera appelé en parallèle pour que $S = S_1, ..., S_{Ns}$ puissent construire $Ns$ tournées initiales $R_1, ..., R_{Ns}$. Rappelons que l'algorithme CW est basé sur le calcul des gains pour combiner deux clients dans la même tournée, si ça va optimiser le coût. Initialement, chaque client est considéré comme faisant l'objet de visite séparée. L'implémentation parallèle du CW sur GPU est déjà expliquée dans la sous-section 3.3.4. Chaque secteur est considéré comme un TSP indépendant résolu par une seule tournée avec un véhicule de type $m$. Nous construisons ces tournées $R_i^m = CW(S_i)$ pour $i = 1, ..., Ns$, en parallèle.

### Trouver le véhicule le plus approprié à chaque tournée

Nous avons partitionné le problème en Ns-TSP en utilisant la capacité $C_m$, puis nous avons utilisé l'algorithme CW en parallèle pour obtenir une solution initiale.

Nous allons maintenant calculer pour chaque tournée toutes les possibilités d'utilisation de tous les types de véhicules. Nous faisons cela en parallèle en exploitant le parallélisme sur GPU. En effet, chaque $R_i^m$ est traitée indépendamment des autres tournées, ce qui fait que les Ns tournées seront donc traitées en parallèle. Précisément, soit la tournée $R_i^m = (u_1, ..., u_{ti})$, puis pour chaque type de véhicule $k$, nous partitionnons $R_i^m$, en commençant par $u_1$ à $u_{ti}$ et inversement en commençant par $u_{ti}$ jusqu'à $u_1$, en sous-tournées qui peuvent être servies par des véhicules de type $k$. Enfin, nous sélectionnons le type $k^*$ qui minimise le coût$(R_i^k)+ F_k$.

**Algorithme 2** : Trouver le véhicule le plus approprié pour $R_i^m = \{u_1, ..., u_{ti}\}$ [25]

**pour tout** $k, 1 \leq k \leq m$ **en parallèle faire**
  capacité $= 0$ ; sous-tournée $= \emptyset$ ;
  **pour** $j = 1$ jusqu'à $ti$ **et pour** $j = ti$ jusqu'à 1 **faire**
  **si** capacité + $d_{u_j} \leq C_k$ **alors**
    ajouter $u_j$ au sous-tournée ;
    capacité = capacité $+d_{u_j}$ ;
  **sinon**
    créer une nouvelle sous-tournée et ajouter $u_j$ à cette sous-tournée.
    capacité $= d_{u_j}$ ;
  **fin si**
  **fin pour**
  $cout_k = \sum$(coût(sous-tournée de $R^k$)+ $F_k$)
**fin pour**
trouver $k^*$ tel que $cout_{k^*} = min_k\{cout_k\}$

À la fin de cette étape, la tournée $R_i^m$ est divisée en une série de sous-tournées $R_{i1}^{k^*}, ..., R_{it}^{k^*}$, toutes servies avec un véhicule de type $k^*$. Nous trouvons en parallèle le meilleur type de véhicule pour chaque $R_i^m$, pour tous les $1 \leq i \leq Ns$. Pour simplifier les notations, notons $R_1^{k1}, ..., R_t^{kt}$, les tournées obtenues à partir de la solution initiale $R_1^m, ..., R_{Ns}^m$. (Avec $Ns \leq t$).

### Amélioration de la solution initiale

Nous adaptons le *move*() et *swap*() utilisé déjà dans notre implémentation précédente du VRP classique à la variante multi capacités. Un déplacement des clients de $R_i$ à $R_{i+1}$ ou de $R_{i+1}$ à $R_i$ pourrait réduire le coût de ces deux tournées. De même, les permutations de clients entre $R_i$ et $R_{i+1}$ pourraient réduire le coût de ces deux tournées. Ce processus de déplacement / permutation de clients entre des paires de tournées adjacentes est répété afin de minimiser le coût global du VRP multi capacités. Ces opérations se font entre paires ($R_1$, $R_2$), ($R_3$, $R_4$), ..., ($R_{t-2}$, $R_{t-1}$) en parallèle et entre ($R_2$, $R_3$), ($R_4$, $R_5$), ..., ($R_{t-1}$, $R_t$) puisque ces paires sont aussi des tournées voisines. Nous appliquons alternativement ces deux étapes. Rappelez vous que $Opt(R, R')$ est la fonction qui consiste à déplacer / permutation des clients entre les tournées $R$ et $R'$. L'amélioration de la solution $(R_1, ..., R_t)$ suit la même implémentation précédente (voir Algorithme 1).

L'adaptation des méthodes *move*() et *swap*() pour la variante multi capacités est comme suit :

**Déplacer un client de $R^k$ à $R'^{k'}$**

Le client $u$ peut être déplacé de $R^k$ à $R'^{k'}$ s'il y a un $k1$ (type de véhicule) tel que $demande(R'+u) = demande(R') + d_u \leq C_{k1}$ et si l'opération est rentable, c-à-d que le coût total des nouvelles tournées est inférieur à l'ancien, à savoir :

$$\text{coût}(R-u)+F_{k2} + \text{coût}(R'+u)+F_{k1} < \text{coût}(R)+F_k + \text{coût}(R')+F_{k'}$$

où $k2$ est le type de véhicule utilisé pour servir la tournée $(R-u)$, il faut que $k2 \leq k$. La tournée $(R-u)$ est obtenue à partir de $R$ en remplaçant le chemin $i-u-j$ par $i-j$ dans $R$ et le coût$(R-u)$ = coût$(R)$ - $c_{iu} - c_{uj} + c_{ij}$.

La tournée $(R'+u)$ est obtenue à partir de $R'$, en insérant $u$ dans $R'$ à l'endroit qui minimise le coût de la tournée. Autrement dit, si $u$ est inséré entre $i'$ et $j'$ alors l'arc $i'-j'$ de $R'$ sera remplacé par le chemin $i'-u-j'$ et le coût$(R'+u)$ = coût$(R')$ - $c_{i'j'} + c_{i'u} + c_{uj'} = min_{i,j \subset R'}\{$ coût$(R')$ - $c_{ij} + c_{iu} + c_{uj}\}$.

**Permutation de deux clients u et u' entre $R^k$ et $R'^{k'}$**

La permutation des clients $u$ et $u'$ entre $R$ et $R'$, se fait s'il existe deux types de véhicules $k1$ et $k2$ tels que la $demande(R-u+u') \leq C_{k1}$ et la $demande(R'-u'+u) \leq C_{k2}$, et si l'opération est rentable c-à-d que le coût total des nouvelles tournées est inférieur à l'ancien :

$$\text{coût}(R-u+u') + F_{k1} + \text{coût}(R'-u'+u) + F_{k2} < \text{coût}(R) + F_k + \text{coût}(R') + F_{k'}$$

$u$ est remplacé par $u'$ dans $R$ et $u'$ est remplacé par $u$ dans $R'$ ($u$ et $u'$ peuvent être insérés dans $R$ et $R'$ à des endroits plus appropriés).

### 3.4.3 Implémentation sur GPU

**Aperçu de la mise en œuvre**

L'implémentation comprend les cinq étapes suivantes :

1. DataCopyToGPU() ; // copier la matrice de distance D sur GPU.
   // D(0 :n, 0 :n), les demandes d(1 : n), la capacité C(1 : m) et les coûts fixes F(1 : m).
2. PartitionIntoSectors() ; // Partitionner le VRP en Ns Secteurs (Ns-TSP).
3. FindInitialSolution() ; // Construire Ns tournées, en utilisant l'algorithme CW.
4. FindBestVehicle() ; // Trouver le type de véhicule le plus approprié à chaque tournée.

5. ImproveSolution() ; // Améliorer la solution en utilisant 2-$Opt()$.

Dans ce qui suit, nous expliquons chacune de ces étapes.

1. Nous utilisons la fonction classique CUDA $cudaMemcpy()$ pour transférer les données du CPU au GPU. Cependant, la programmation parallèle avec CUDA nécessite une méthodologie précise et elle convient de prendre en compte les différentes mémoires du GPU [24, 132]. Ici, la matrice $D$ est distribuée dans la mémoire partagée de chaque bloc, comme nous avons déjà vu dans l'implémentation précédente. Les demandes, la capacité et les coûts fixes sont dupliqués dans chaque mémoire partagée.
2. La construction du secteur est intrinsèquement séquentielle et est effectuée par le CPU.
3. Nous utilisons CW pour trouver en parallèle les Ns tournées pour les Ns secteurs $R_i = CW(S_i)$, $i = 1, ..., Ns$. Chaque $R_i$ est calculée sur un bloc de $s_i \times s_i$ threads ; où $s_i$ est le nombre de clients du secteur. Puisque, dans CUDA, nous ne pouvons définir que des blocs de threads de taille uniforme, nous utilisons des blocs de $s \times s$ threads, avec $s = max(s_i)$. Les données nécessaires au calcul de $R_i$ sont les colonnes $r_i, ..., s_i + r_i$, les capacités $C_1, ..., C_m$ et les coûts fixes $F_1, ..., F_m$.
   En utilisant la syntaxe CUDA, les blocs, la grille et le kernel CW() sont définis comme suit :
   `dim3 dimBlock(s,s) ; dim3 dimGrid(Ns,1) ;`
   `CW<<<dimGrid,dimBlock>>>(`$S_i$`) ;`
4. Chaque tournée $R_i$ est traitée indépendamment des autres tournées sur un bloc de $s \times s$ threads. Par conséquent, les Ns tournées sont traitées en parallèle sur Ns blocs.
   Justement, soit $R_i = (u_1, ..., u_{ti})$ alors pour chaque type $k$ de véhicule on partitionne $R_i$, en commençant par $u_1$ à $u_{ti}$ et en commençant par $u_{ti}$ jusqu'à $u_1$, en sous-tournées $R_{i1}, ..., R_{it}$ qui peuvent être servies par des véhicules de type $k$. Enfin, nous sélectionnons le type le plus approprié $k^*$ qui minimise le coût$(R_{i1}) + F_{k^*} + ... +$ coût$(R_{it}) + F_{k^*}$.
   Un bloc de $m$ threads est utilisé pour trouver $k^*$ pour chaque tournée $R_i$. Puisque Ns blocs chacun de $m$ threads sont utilisés pour trouver le type de véhicule qui minimise le coût pour les Ns tournées $R_i$. Ainsi, avec la syntaxe CUDA, les blocs, la grille et le kernel $FindBestVehicle()$ sont définis comme suit :
   `dim3 dimBlock(m,1) ; dim3 dimGrid(Ns,1) ;`
   `FindBestVehicle<<<dimGrid,dimBlock>>>(`$R_i$`) ;`

5. Rappelons que les procédures $Opt(R, R')$ basées sur les opérations de déplacement / permutation sont effectuées entre des paires adjacentes de tournées $R$ et $R'$.
   L'opération *move*() est lancée sur un bloc de $s \times m$ threads, chaque thread traite un client de la tournée $R$ qui peut être déplacé vers la tournée $R'$ voisine. Nous utilisons la deuxième dimension du bloc pour trouver le type de véhicule approprié pour les tournées après le déplacement.
   De même, l'opération de permutation est lancée en 3 dimensions de $s \times s \times m$ threads. Un thread de coordonnées ($i$, $j$, $k$) traite le client $i$ de $R$ et du client $j$ de $R'$ et d'un véhicule de type $k$ noté par $swap(i, j)$. La fonction $Opt(R, R')$ s'exécute en parallèle et calcule le gain de chaque opération de déplacement / permutation et choisit le plus grand profit.
   Comme les deux opérations nécessitent les mêmes données, nous lançons chaque tournée dans un bloc et chaque opération dans une dimension du bloc (déplacer dans la première dimension et permuter dans la deuxième dimension).
   Par conséquent, ce traitement est effectué sur une grille à deux dimensions où chaque dimension du bloc lance un ensemble de threads sous la forme de trois dimensions, mais chaque opération utilise uniquement le nombre de threads nécessaire à son traitement.

```
dim3 dimBlock(s,s,m) ; dim3 dimGrid(Ns,2) ;
Opt<<<dimGrid,dimBlock>>>() ;
```

   Comme nous l'avons déjà vu, chaque $Opt(R_i, R_{i+1})$, $1 \leq i \leq Ns$, est exécuté en bloc avec mémoire partagée et nécessite toutes les distances entre les clients $u$ et $u'$ de ces deux tournées qui sont stockés dans les blocs $D_i$ et $D_{i+1}$ de la matrice D comme indiqué sur la figure 3.4.
   Donc $D_i$ et $D_{i+1}$ doivent être stockés dans la mémoire partagée $M_i$ du bloc $B_i$. Maintenant, si $u$ est déplacé de $R_i$ à $R_{i+1}$, nous devrions mettre à jour les mémoires partagées $M_{i-1}$, $M_i$ et $M_{i+1}$. De même, pour une permutation de $u$ et $u'$ entre $R_i$ et $R_{i+1}$ nécessite une mise à jour à ces matrices.

### 3.4.4 Résultats expérimentaux

L'implémentation GPU proposée est testée dans le cadre des 12 instances cités par Golden et al. [74], et ses résultats sont comparés à ceux d'Ochi [107] et de Karagul [84] (les repères bien connus pour ce problème). Comme le montre le tableau ci-dessous, notre algorithme offre des meilleures solutions (en termes de coûts) pour la plupart de ces instances.

Tableau 3.1 – Comparaison de la qualités des solutions des 12 instances proposées par Golden.

| Instance | Borne Inférieur | Ochi | Karagul | Sol GPU |
|---|---|---|---|---|
| 3 | 961.03 | 1088.70 | 999.20 | **983.51** |
| 4 | 6437.30 | 7324.70 | 7324.7 | 7408.5 |
| 5 | 1007.10 | 1153.00 | 1097.4 | **1053.03** |
| 6 | 6516.50 | 7031.40 | 7031.40 | 7112.1 |
| 13 | 2406.40 | 2670.70 | 2680.20 | **2539.6** |
| 14 | 9119.00 | 9214.40 | 9214.40 | 9423.1 |
| 15 | 2586.40 | 2800.10 | 2861.20 | **2745.02** |
| 16 | 2720.40 | 3063.80 | 2899.00 | **2890.7** |
| 17 | 1734.50 | 2088.90 | 1954.10 | **1934.0** |
| 18 | 2369.70 | 2992.40 | 2986.50 | **2603.1** |
| 19 | 8661.80 | 9599.20 | 9824.80 | **9702.1** |
| 20 | 4039.50 | 4459.10 | 4498.90 | **4375.2** |

Dans la suite, nous comparons les temps d'exécution de l'implémentation sur CPU et sur GPU. Nous avons utilisé la même configuration matérielle adoptée pour le VRP classique dans la section précédente. D'après les tests, l'accélération optimale, pour ce matériel, ne peut pas dépasser 8 pour ce problème. Dans la figure 3.11, nous comparons les temps d'exécution du CPU et du GPU pour chaque problème de Golden. Les résultats obtenus montrent que les temps d'exécution sont réduits sur le GPU pour tous les problèmes étudiés et que le facteur d'accélération atteint 8 pour l'instance numéro 18.

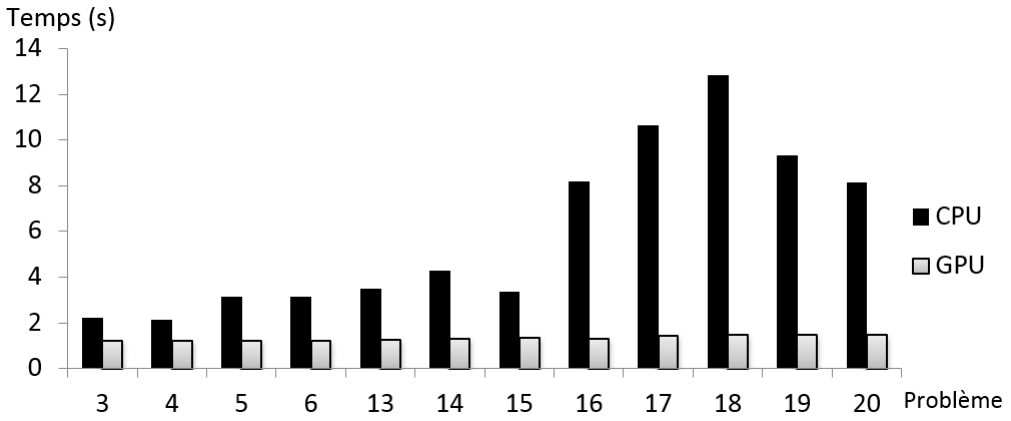



Figure 3.11 – Temps d'exécution des 12 instances sur CPU et sur GPU.

La figure 3.12 donne les temps d'exécution du CPU par rapport au GPU pour des instances aléatoires. Dans ces instances, les coordonnées des clients sont générées aléatoirement et le nombre de types de véhicules est choisi comme suit : Les problèmes avec un nombre de clients inférieur à 50 utilisent 3 types de véhicules. Les problèmes avec un nombre de clients compris entre 50 et 80 utilisent 7 types de véhicules et les problèmes avec un nombre de clients supérieur à 80 utilisent 9 types de véhicules.

Ici, le facteur d'accélération est supérieur à 8 pour $n = 120$ et cela signifie que notre algorithme est adapté à la puissance du GPU. De toute évidence, l'augmentation de la taille des données (nombre de clients et nombre de types de véhicules) pourrait augmenter le facteur d'accélération.

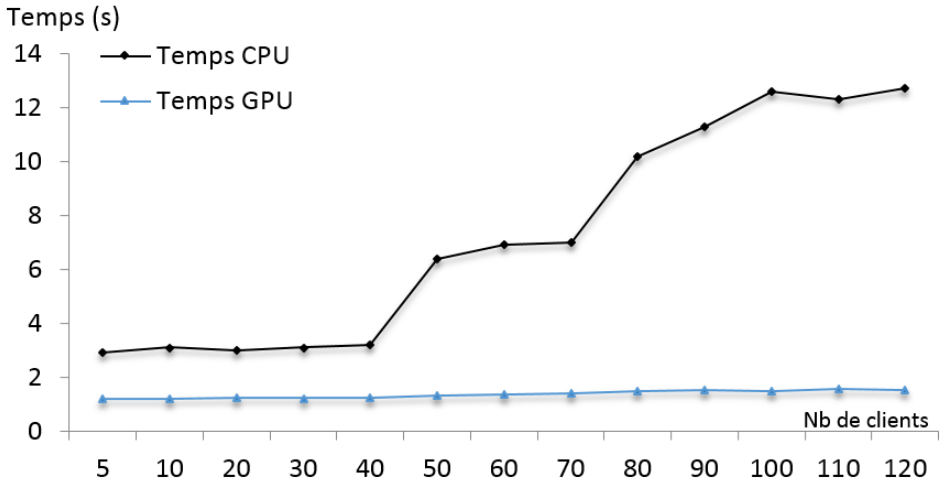



FIGURE 3.12 – Temps d'exécution des instances aléatoires sur CPU et sur GPU.

## 3.5 Problèmes de tournées de véhicules dynamique en temps réel

Dans le problème de tournées de véhicules dynamique (DVRP), les nouvelles demandes sont reçues au fil du temps et doivent être incorporées dynamiquement dans les tournées déjà planifiées, tandis que les véhicules ont déjà commencé leurs tournées. Deux approches principales sont utilisées pour résoudre le DVRP comme indiqué dans Abbatecola et al. [1].

i) L'approche de ré-optimisation périodique donne un premier plan de routage initial optimal (ensemble de tournées) avant le début de la journée de travail. Une fois le transport commence, une procédure d'optimisation résout périodiquement un problème statique correspondant à l'état actuel, chaque fois que les données disponibles changent.

   Généralement, l'horizon de planification est divisé en tranches de temps (périodes) et le DVRP est résolu en résolvant un VRP statique à chaque tranche de temps. Hanshar [78] utilise cette approche et résout chaque VRP par un algorithme génétique.

ii) La ré-optimisation continue où l'optimisation est continuellement réalisée tout au long de l'horizon temporel, tandis qu'une mémoire adaptative stocke des informations sur les bonnes solutions.

   Chaque fois que les données disponibles changent, une procédure de décision regroupe les informations de la mémoire pour effectuer une ré-optimisation. Gendreau et Potvin [68] et d'autres auteurs ont adopté cette approche pour résoudre le DVRP.

Aussi nous avons résolu le DVRP avec les deux approches, cette section est réservée à résoudre le DVRP avec une heuristique simple qui insère, rapidement et efficacement, des requêtes dynamiques dans des tournées déjà planifiées avec une ré-optimisation continue. Tandis que dans la prochaine section (section 3.6) nous proposons un algorithme génétique basé sur le GPU pour la résolution de grands DVRP avec une approche de ré-optimisation périodique.

### 3.5.1 Description de l'approche

Le VRP dynamique nécessite de prendre des décisions aussi rapidement que possible parce que chaque seconde compte. Cela nécessite des méthodes de résolution avec une haute efficacité de calcul et sans diminution de la qualité de la solution déjà planifiée. Dans cette partie, nous montrons encore que l'utilisation des GPUs modernes peut contribuer à atteindre ces objectifs et nous présentons dans ce qui suit la conception et l'implémentation sur GPU d'une méthode simple qui insère en temps réel des requêtes dynamiques dans des tournées déjà planifiées [16]. Les résultats expérimentaux confirment que la mise en œuvre proposée exploite efficacement le parallélisme et atteind les objectifs fixés pour cette variante de VRP.

### 3.5.2 Présentation du VRP dynamique

Soient $R_0, ..., R_{m-1}$ la solution d'un VRP (ensemble de tournées) avec $n$ clients $1, ..., n$ et un dépôt. Pour simplifier la présentation, nous considérons que la tournée voisine de $R_i$ est la tournée $R_{(i+1) mod\ m}$ comme on a déjà expliqué dans les sections

précédentes. On peut définir un grand ensemble de tournées comme voisinage de $R_i$ et cela ne changera pas le principe de notre méthode. Soit Td le tableau des demandes des clients ordonné de la même manière que les clients sur les tournées : Td = ((demande (premier client en $R_0$),..., demande (dernier client en $R_0$), ..., (demande (premier client en $R_{m-1}$), ..., demande (dernier client en $R_{m-1}$)).

La problématique du DVRP est d'insérer, dans un délai minimal, de nouveaux clients dans les tournées déjà planifiées sans avoir replanifié les parties non visitées des tournées et sans diminution de la qualité de la solution déjà obtenue.

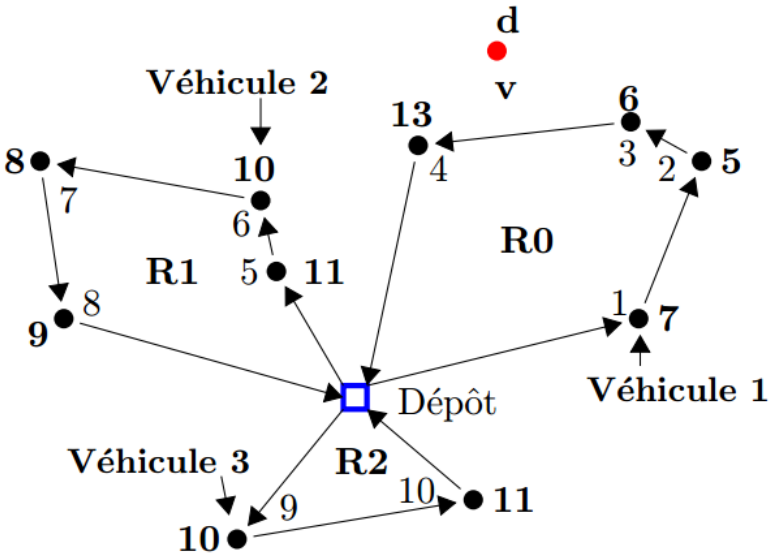



FIGURE 3.13 – Exemple de la solution initiale.

Un simple exemple est montré dans la figure 3.13 où $n = 10$, $m = 3$, $R_0$ = (0,1,2,3,4,0), $R_1$ = (0,5,6,7,8,0), $R_2$ = (0,9,10,0), $Td$ = ((7,5,6,13), (11,10,8,9), (10,11)) et $C = 40$ la capacité des véhicules. Notons que les numéros de nœuds sont écrits à l'intérieur des tournées et les demandes à l'extérieur. Les flèches continues représentent les trois tournées prévues par le répartiteur avant que les véhicules ne quittent le dépôt.

Les positions des véhicules au moment de la réception de la nouvelle demande dynamique ($v$, $d$) sont représentées sur cette figure (nœud 1 de $R_0$, nœud 6 de $R_1$ et nœud 9 de $R_2$).

### 3.5.3 Principe de la méthode

Le principe de la méthode est le suivant : Soit $R_{i^*}$ la tournée la plus proche à la nouvelle requête dynamique ($v$, $d$), où $v$ est le nouveau client et $d$ est sa demande. L'indice $i^*$ est tel qu'il y a un arc ($u^*$, $w^*$) dans $R_{i^*}$ non encore visité tel que : coût$(u^*, v, w^*) = min_i$ coût$(u, v, w)$ pour tous ($u$, $w$) dans $R_i$ non encore visité.

- Si $demande(R_{i^*}) + d \leq C$, insérer $v$ dans $R_{i^*}$ entre $u^*$ et $w^*$ et retirer l'arc ($u^*$, $w^*$) de $R_{i^*}$, c-à-d enlever l'arc ($u^*$, $w^*$) et connecter $u^*$ à $v$ et $v$ à $w^*$. Les autres tournées restent inchangeables.

- Si $demande(R_{i^*}) + d > C$ alors déplacer les derniers clients de $R_{i^*}$ qui provoquent la violation de la capacité du véhicule vers la tournée $R_{(i^*+1) mod\ m}$. Plus précisément, soit $R_{i^*} = (0, u_1, ..., u_r, 0)$ déplacer alors le chemin $(u_k, ..., u_r)$ vers $R_{(i^*+1) mod\ m}$ où $k$ est l'indice du clients tel que :
  $demand(0, u_1, ..., u_{k-1}) + d \leq C < demand(0, u_1, ..., u_k) + d$.
  De même, déplacer le chemin composé des derniers clients de $R_{(i^*+1) mod\ m}$ qui provoquent la violation de la capacité du véhicule à $R_{(i^*+2) mod\ m}$, c-à-d si $R_{i^*+1} = (0, u'_1, ..., u'_{r'}, 0)$ puis déplacer le chemin $(u'_{k'}, ..., u'_{r'})$ vers $R_{(i^*+2) mod\ m}$ où $k'$ est l'indice du clients dans $R_{i^*+1}$ tel que :
  $demand(u_k, ..., u_r) + demand(0, u'_1, ..., u'_{k'-1}) \leq C <$
  $demand(u_k, ..., u_r) + demand(0, u'_1, ..., u'_{k'})$, etc.
  En partant de $i = i^*$, répéter ce processus de déplacement des clients de $R_i$ à $R_{(i+1) mod\ m}$ jusqu'à atteindre $i^* - 1$. Notons que nous ne replanifions pas les tournées, mais seulement nous supprimons simplement la dernière partie de $R_i$ et l'insérons dans $R_{(i+1) mod\ m}$ pour certains indices des tournées et cela se fait très rapidement.

- En fin de compte, si les clients déplacés de $R_{(i^*-2) mod\ m}$ à $R_{(i^*-1) mod\ m}$ ne causent pas de violation de capacité, insérer-les dans $R_{(i^*-1) mod\ m}$, sinon renommer $R_i$ comme $R_{i+1}$ pour tous les $i > i^*$ et créer une nouvelle tournée pour les clients qui provoquent une violation de capacité en $R_{(i^*-1) mod\ m}$ (cette indexation préserve le concept de relations de voisinage entre tournées).

Ces trois cas sont illustrés sur trois figures où la demande dynamique ($v$, $d$) est insérée dans la solution initiale de la figure 3.13 (les véhicules sont positionnés dans les nœuds 1, 6 et 9). Nous supposons que $R_0$ est la route la plus proche à la nouvelle requête dynamique ($v$, $d$). Sur la figure 3.14, nous avons supposé que la demande $d = 7$ et l'arc ($u^*$,$w^*$) = (3,4). Puisque la $demande(0, 1, 2, 3, 4) + d = 38 < C$ alors $v$ est inséré entre 3 et 4.

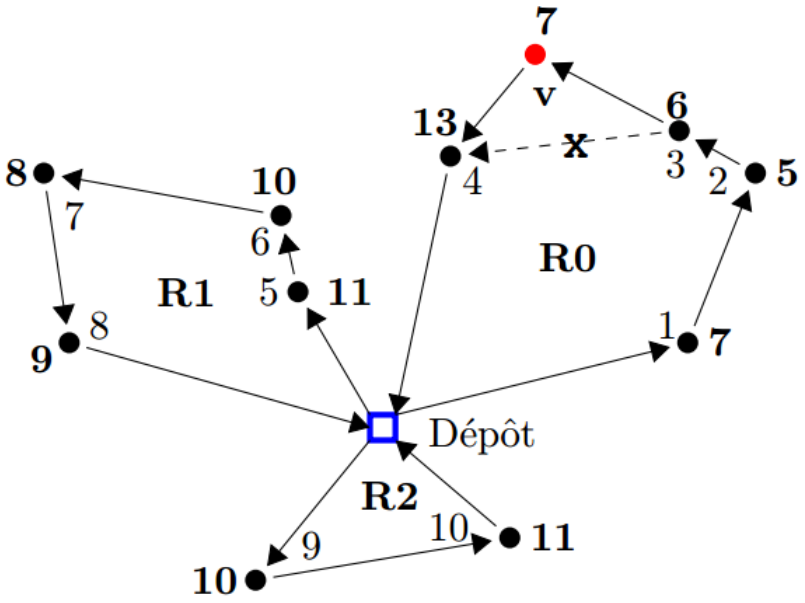



FIGURE 3.14 – Insérer (v, d) sans déplacement des clients.

Pour la figure 3.15, nous avons supposé que $d = 15$ qui implique que $demande(0, 1, 2, 3)+ d = 33 < C < demande(0, 1, 2, 3, 4) + d = 46$. Par conséquent, le client 4 doit être déplacé à $R_1$.

Depuis la $demande(4)+demande(0, 5, 6) = 34 < C < demande(4)+demande(0, 5, 6, 7, 8) = 51$, les clients 7 et 8 sont déplacés de $R_1$ et insérés dans $R_2$ sans déplacer les derniers clients de $R_2$ puisque $demande(7, 8) + demande(9, 10) = 38 < C$. Sur la figure 3.16, nous avons supposé que $d = 15$ et $d_8 = 15$. Dans ce cas, $R_0$ devient $(0, 1, 2, 3, v, 0)$, $R_1$ devient $(0, 5, 6, 4, 0)$ et $R_2$ devient $(0, 9, 8, 7, 0)$ et une nouvelle route $R_3 = (0, 10, 0)$ est créé, car $demande(0, 7, 8, 9, 10) > C$.

Notons que le choix de la tournée la plus proche $R_{i*}$ pour insérer $(v, d)$ n'est pas toujours bon et il peut produire des solutions de mauvaise qualité. Comme la méthode est conçue pour être implémentée sur le GPU, nous calculons en parallèle les $m$ solutions obtenues en insérant $(v, d)$ dans $R_0$, dans $R_1$, ..., dans $R_{m-1}$ et nous sélectionnons celle avec le coût minimal. Toutes ces solutions sont calculées parfaitement en parallèle, donc le temps de calcul de toutes les solutions est le même que le temps de calcul de l'une d'entre elles. Notons que :

- il est clair que cette approche est inefficace en séquentielle, mais elle est intéressante pour une mise en œuvre parallèle.

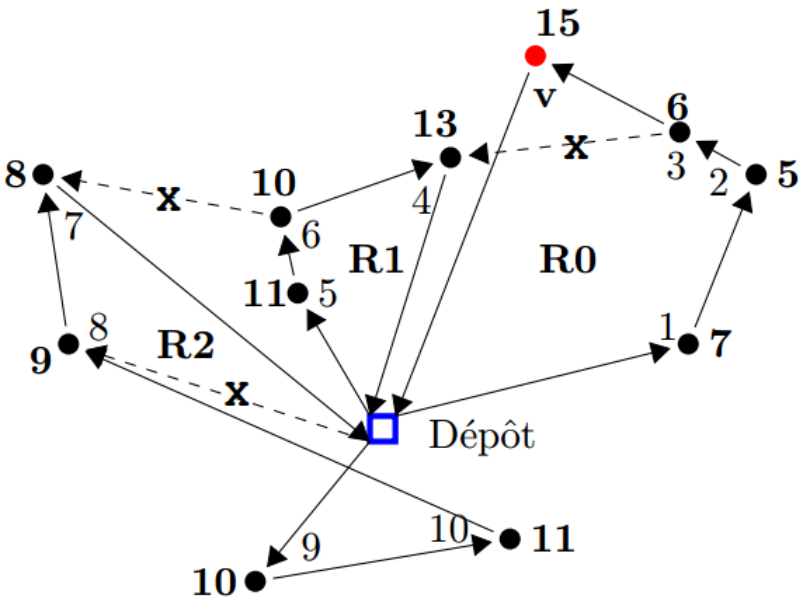



FIGURE 3.15 – Insérer (v, d) avec déplacement des clients.

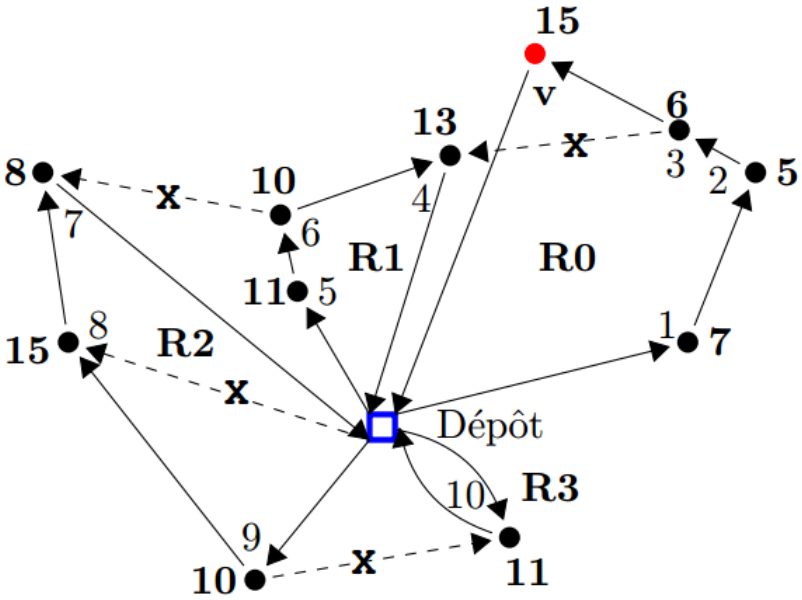



FIGURE 3.16 – Insérer (v, d) avec la création d'une nouvelle tournée.

- Cette approche peut facilement gérer plusieurs capacités (utiliser $C_k$ au lieu de $C$), multi dépôts, fenêtres temporelles et autres contraintes (pour cela déplacer les clients de $R_{i^*}$ à la tournée voisine de $R_{i^*}$) qui provoquent une violation de la capacité du véhicule, des fenêtres temporelles et d'autres contraintes.

### 3.5.4 Insertion d'une demande dynamique

Soit $L_{ki}$, $0 \leq k, i < m$, le dernier chemin des clients (sans le dépôt) qui sera déplacé de $R_i$ à $R_{(i+1)mod\ m}$, si $(v,\ d)$ est inséré dans $R_k$. Ainsi, le chemin $L_{ki}$ doit être supprimé de $R_i$ et inséré dans $R_{(i+1)mod\ m}$.

Supprimer $L_{ki}$ de $R_i$ consiste à connecter le dernier nœud de $R_i \backslash L_{ki}$ au dépôt. Insérer $L_{ki}$ dans $R_{(i+1)mod\ m}$ consiste à trouver $(u^*,\ w^*)$ qui vérifier :
$min$ coût$(u, L_{ki}, w)$ - coût$(u, w)$, pour tout $(u,\ w)$ non encore visité dans $R_{(i+1)mod\ m,k} \backslash L_{k,(i+1)mod\ m}$ et insérer le chemin $L_{ki}$ entre les nœuds $u^*$ et $w^*$.

Insérer $(v,\ d)$ dans $R_k$ consiste à trouver $(u^*,\ w^*)$ le minimum des coût$(u,\ v,\ w)$ - coût$(u,\ w)$ pour tous $(u,\ w)$ dans $R_k$ non déjà délivré et en fin connectant $u^*$ à $v$ et $v$ à $w^*$. (voir l'algorithme 3).

**Algorithme 3** : Algorithme parallèle InsertDynamicRequest($v$, $d$) [16].

**pour tout** $k, 1 \leq k \leq m$ **en parallèle faire**
  en parallèle calculer $L_{ki}$ ;
  **pour tout** $i, 1 \leq i \leq m$ **en parallèle faire**
    **si** $i = k$ **alors**
      retirer $L_{ki}$ de $R_k$ ;
      insérer ($v$,$d$) dans $R_k$ ;
    **sinon** $\{i \neq k\}$
      retirer $L_{ki}$ de $R_i$ ;
      insérer $L_{k,(i-1) mod\ m}$ dans $R_i$
    **fin si**
    $R_{ik}$ = résultat après la suppression de $L_{ki}$ et l'insertion de $L_{k,(i-1) mod\ m}$ dans la tournée $R_i$ ;
  **fin pour**
  **si** $demand(R_{(k-1) mod\ m}) > C$ **alors**
    **pour tout** $i, i > k$ **faire**
      renommer $R_i$ comme $R_{i+1}$ ;
    **fin pour**
    créer une nouvelle tournée composé de dépôt et de nœuds qui provoquent une violation de la capacité de $R_{(k-1) mod\ m,k}$.
  **fin si**
  calculer $k^*$ qui vérifier $min_k \ \sum_{i=0}^{m}$ coût($R_{ik}$) ;
  **pour tout** $i, 0 \leq i < m$ **faire**
    $R_i = R_{ik^*}$ ;
  **fin pour**
  mettre à jour $Td$ et la matrice de distance $D$ selon $k^*$(pour la prochaine insertion).
**fin pour**

### 3.5.5 Insertion de plusieurs demandes dynamiques

Si plusieurs requêtes dynamiques ($v_1$, $d_1$), ..., ($v_{nr}$, $d_{nr}$) doivent être insérées en même temps, une idée consiste à les insérer ensemble chacune dans la tournée la plus proche. Ensuite, déplacer des parties des tournées à l'origine de violations du capacité sur les tournées voisines. Cependant, les tests que nous avons effectué montrent que cette approche est efficace en séquentiel mais ne produit pas toujours des solutions de bonne qualité. Par conséquent, nous insérons plusieurs requêtes dynamiques les unes après les autres dans un ordre spécifique.

L'ordre d'insertion de ces demandes a un impact sur les performances de la solution résultante. En effet, si on commence par insérer ($v_1$, $d_1$) dans la solution initiale ($R_0$, ..., $R_{m-1}$), on obtient alors une nouvelle solution ($R_{10}$, ..., $R_{1,m-1}$) (éventuellement $R_{1m}$). Puis à partir de cette dernière solution, on insère ($v_2$, $d_2$) qui donne une solution ($R_{20}$, ..., $R_{2,m-1}$) (éventuellement $R_{2m}$, $R_{2,m+1}$) et ainsi de suite. Ou, si on commence par insérer ($v_2$, $d_2$) dans la solution initiale ($R_0$, ..., $R_{m-1}$), on obtient alors une autre solution ($R'_{10}$, ..., $R'_{1,m-1}$) (éventuellement $R'_{1m}$). Ensuite, l'insertion ($v_1$, $d_1$) dans cette dernière solution donne une autre ($R'_{20}$, ..., $R'_{2,m-1}$) (éventuellement $R'_{2m}$, $R'_{2,m+1}$) et ainsi de suite. Les coûts de ces solutions peuvent être radicalement différents, comme nous l'avons constaté lors de certaines expériences. La question est maintenant c'est quoi l'ordre à suivre pour faire les insertions de ces requêtes dynamiques afin d'obtenir des solutions de qualité ?

Puisque toutes ces solutions peuvent être calculées en parallèle sur le GPU, le temps de calcul en parallèle de toutes les solutions est le même que le temps nécessaire en séquentielle pour calculer une seule solution obtenue en insérant ($v_i$, $d_i$) pour n'importe quel client $v_i$. Donc, l'approche adoptée pour insérer ($v_1$, $d_1$), ..., ($v_{nr}$, $d_{nr}$) dans ($R_0$, ..., $R_{m-1}$) est résumée dans l'exemple suivant (notons que cette approche est inefficace en séquentiel mais convient à la méthodologie du parallélisme et produit des bonnes solutions) :

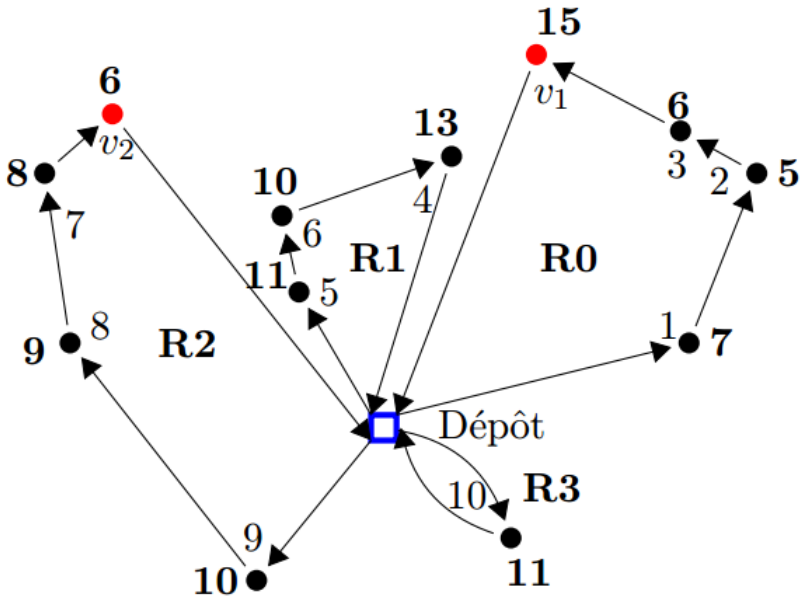



FIGURE 3.17 – L'insertion du ($v_1$, $d_1$) puis l'insertion du ($v_2$, $d_2$).

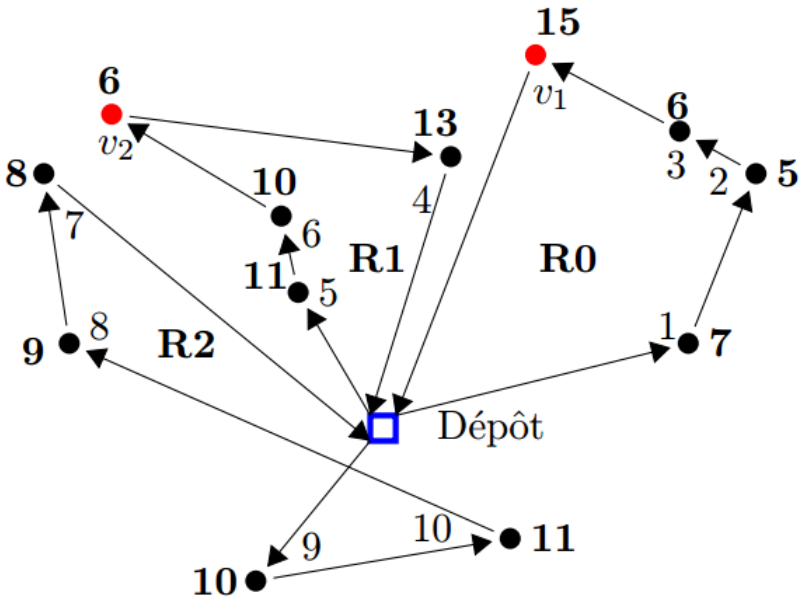



FIGURE 3.18 – L'insertion du ($v_2$, $d_2$) puis l'insertion du ($v_1$, $d_1$).

La figure 3.17 montre comment insérer deux requêtes dynamiques à partir de la solution initiale de la figure 3.13. Nous commençons par insérer ($v_1$, $d_1$) puis insérer ($v_2$, $d_2$)), alors nous obtenons la solution représentée dans la figure 3.17. En parallèle, on commence par insérer ($v_2$, $d_2$) puis on insère ($v_1$, $d_1$), alors on obtient la solution représentée dans la figure 3.18. Ainsi, la solution adaptée est celle avec le coût minimal (voir algorithme 4).

**Algorithme 4** : Algorithme parallèle InsertSeveralDynamicRequests($v_1$, $d_1$), ..., ($v_{nr}$, $d_{nr}$) [16].

**Données:** solution(0) = solution initiale du VRP ;
  $requête\_vk = \{1, ..., nr\}$ ;
  **pour** $k, 1 \leq k \leq nr$ **faire**
    **pour tout** $i \in requête\_vk$ **en parallèle faire**
      calculer solution($k$,$i$) = insérer($v_i$,$d_i$) dans solution(k-1) ;
      sélectionner $i^*$ tel que coût($solution(k, i^*)$) = $min_i$ coût($solution(k, i)$) ;
      solution($k$)=solution($k$,$i^*$) ;
      $requête\_vk = requête\_vk \backslash \{i^*\}$ ;
    **fin pour**
  **fin pour**

Après l'insertion d'une nouvelle demande dynamique, le nombre de tournées augmente d'au plus par un et le nombre de clients par tournée augmente d'au plus par $s$ ; où $s$ est le nombre maximum de clients par tournée. Ainsi, après avoir inséré $nr$ nouvelles demandes, le nombre de tournées est respectivement limité par $m+nr$. Tandis que le nombre maximal de clients par tournée est limité par $(1+nr)*s$. Ces paramètres sont utilisés dans l'implémentation GPU suivante.

### 3.5.6 Implémentation sur GPU

Dans la section 3.3 nous avons présenté comment résoudre le VRP sur GPU. La solution obtenue consiste en $m$ tournées $R_0, ..., R_{m-1}$ calculées en parallèle sur $m$ blocs chacun de $s \times s$ threads ; où $s$ est le nombre maximum de clients par tournée. Nous présentons ici comment insérer une demande dynamique $(v,\ d)$ dans cette solution initiale en sachant que certains clients ont déjà été visités. Pour ce faire, nous partitionnons le GPU sur $m+1$ blocs $B_i$ chacun est de $s \times m * s$ threads.

Le bloc $B_i$, $0 \leq i < m$, stocke dans sa mémoire partagée $M_i$, les données nécessaires au traitement de la tournée $R_i$ et effectue en parallèle tous les calculs liés à $R_i$. Plus précisément, les tâches suivantes sont effectuées en parallèle par chaque bloc $B_i$ :

- Calculer en parallèle les listes $L_{ki}$ et supprime $L_{ki}$ de $R_i$ pour tout $0 \leq i < m$. Pour cela, il est nécessaire de stocker le tableau ordonné des demandes $Td = (Td_1, ..., Td_m)$ dans $M_i$ (rappelons que $Td_j$ est le sous-tableau ordonné des demandes de $R_j$, $0 \leq j < m$).

- Trouver en parallèle $(u^*,\ w^*)$ qui atteint le minimum : coût$(u, L_{k(i-1)mod\ m}, w)$ - coût $(u, w)$. Avec $(u,\ w)$ n'a pas déjà été visité dans $R_i \backslash L_{ki}$ et insère $L_{k(i-1)mod\ m}$ entre les nœuds $u^*$ et $w^*$ (si $i = k$, alors $L_{k(i-1)mod\ m} = (v)$). Cela nécessite de stocker la matrice de distance de bloc $D_i = c_{uu'}$ avec $u$ dans $R_i$ et $0 < u' < n$, et toutes les distances entre les nœuds dans $L_{k(i-1)mod\ m}$ et $R_i$. Comme les nœuds dans $L_{k(i-1)mod\ m}$ appartiennent à $R_{(i-1)mod\ m}$, nous stockons les blocs $D_{(i-1)mod\ m}$ et $D_i$ dans la mémoire $M_i$.

- Calculer et écrire le coût$(R_{ik})$ dans la mémoire globale du GPU ; où $R_{ik}$ est la tournée obtenue après la suppression de $L_{ki}$ de $R_i$ et l'insertion de $L_{k(i-1)mod\ m}$ dans $R_i$.

Un bloc spécial (bloc maître) calcule l'indice $k^*$ qui $minimise \sum_{i=0}^{m}$ coût$(R_{ik})$, pour $1 \leq k < m$. Chaque bloc $B_i$ lit $k^*$ et mettre à jour ses données locales pour l'insertion suivante : il met $R_i = R_{ik^*}$ et met à jour $Td$, $D_{i-1}$ et $D_i$ en fonction de l'indice $k^*$ (ce qui indique que la requête dynamique $(v, d)$ est insérée dans la tournée $R_{k^*}$).

Chaque bloc $B_i$, $0 \leq i < m$, est défini par $s \times m * s$ threads et la grille est définie par $m$ blocs. Ainsi, avec la syntaxe CUDA, les blocs et la grille sont déclarés comme suit :
`dim3 dimBlock(s,m*s) ; dim3 dimGrid(m,1) ;`

Le kernel $InsertDynamicRequest((v, d))$ est lancé en invoquant
`InsertDynamicRequest <<< dimGrid, dimBlock >>> (v, d) ;`
qui exécute ce kernel sur chaque thread des $m$ blocs $B_i$.

Maintenant, pour insérer au plus $nr$ requêtes dynamiques, les blocs et la grille doivent être définis comme suit :
`dim3 dimBlock ((1 + nr) * s, (n + nr) * s) ;`
`dim3 dimGrid (m + nr, 1).`

Le kernel insère plusieurs requêtes dynamiques est lancé par :
`InsertSeveralDynamicRequest <<< dimGrid, dimBlock >>>`
`((`$v_1$`, `$d_1$`), ..., (`$v_{nr}$`, `$d_{nr}$`)).`

### 3.5.7 Résultats expérimentaux

Nous avons implémenté notre algorithme pour insérer plusieurs demandes dynamiques sous deux versions ; une sur le CPU et l'autre sur GPU. Nous comparons les deux versions en terme de temps de calcul et en terme de qualité des solutions. Nous avons utilisé la même configuration matérielle utilisée pour les implémentations précédentes.

Nous avons proposé nos propres instances pour tester l'efficacité de notre implémentation sur GPU, pour cela nous avons fixé la capacité $C$ des véhicules, alors que la distance $(u, v)$ et la demande$(u)$ sont générées aléatoirement pour $n$ varie de 10 à 30. Tandis que, le nombre de requêtes dynamiques à insérer varie de 1 à 100 (limité par la taille de la mémoire partagée du bloc $B_i$). Les transferts de données entre le CPU et le GPU ne se font qu'au début et à la fin du programme. Par conséquent, les temps d'exécution mesurés n'incluent pas le temps des transferts de données entre le CPU et le GPU.

Pour la comparaison des temps séquentiels et parallèles, le programme séquentiel insère chaque requête dynamique dans les tournées les plus proches (pas sur toutes les tournées, puis sélectionne la meilleure solution) et insère plusieurs requêtes dynamiques les unes après les autres dans l'ordre d'arrivée. Cependant, le programme parallèle détermine le meilleur ordre d'insertion de ces requêtes et le meilleur moyen d'insérer une requête dynamique comme expliqué précédemment. Par conséquent, la qualité de la solution obtenue par le programme parallèle est meilleure que celle obtenue par le programme séquentiel pour la même instance.

Par exemple, nous avons reporté dans la figure 3.19, les temps d'exécutions des implémentations sur CPU et sur le GPU de la solution initiale de la figure 3.13. À partir de cette solution initiale, nous avons calculé le temps nécessaire pour insérer de 1 à 100 nouvelles demandes dynamiques. Suite à ces mesures, il est clair que l'implémentation sur GPU surpasse celle sur CPU à la fois en termes de temps d'exécution (jusqu'à 35 fois plus rapide) et en qualité.

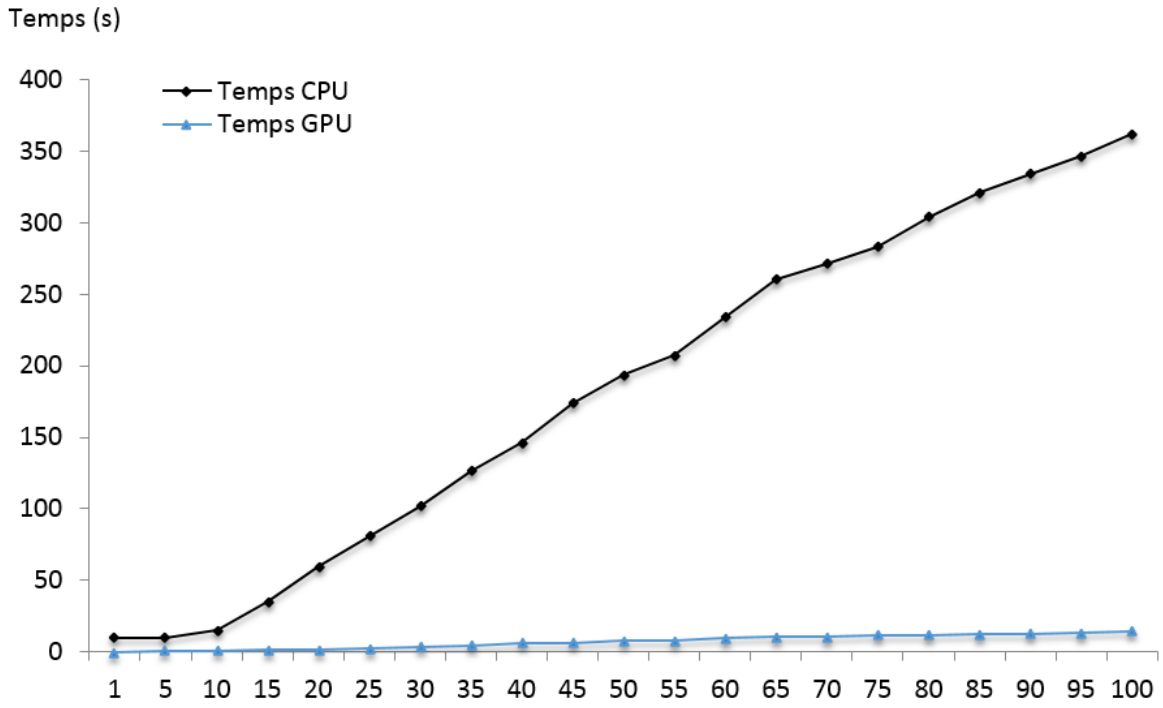



FIGURE 3.19 – Temps d'exécution de l'implémentation sur CPU et sur GPU.

Nous avons conçu une interface, en utilisant google map et des pages Web pour suivre l'évolution des véhicules (une ré-optimisation continue). Cela nous permet d'afficher et de présenter les solutions de manière réaliste (pour un éventuel portage de cette application sur les smartphones). La figure 3.20 montre un exemple simple de cette interface où les tournées sont affichées avec des couleurs différentes.

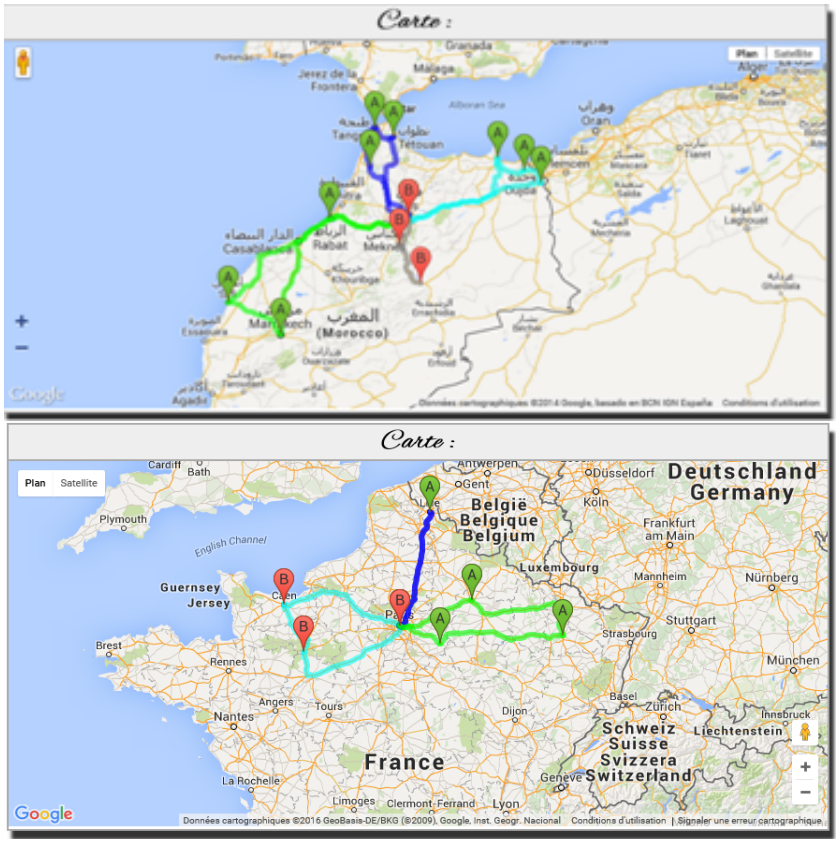



FIGURE 3.20 – Exemple d'affichage avec google map.

Cette implémentation exploite efficacement les performances des GPUs et construit des solutions pour le DVRP avec une haute efficacité de calcul et sans diminution de la qualité de la solution déjà planifiée. Le calcul des solutions se réalise presque en temps réel avec une ré-optimisation continue et afin de présenter les emplacements des véhicules, la technologie de google map a été utilisée.

## 3.6 Problèmes de tournées de véhicules dynamique de grandes tailles

### 3.6.1 Description de l'approche

Les VPRs sont dynamiques dans le sens où les nouvelles demandes des clients arrivent continuellement dans le temps, après que plusieurs véhicules aient déjà commencé leurs tournées. Ces nouvelles demandes doivent être intégrées aux tournées déjà planifiées. Deux approches principales sont utilisées pour résoudre le DVRP, l'approche de ré-optimisation périodique et la ré-optimisation continue.

Les VRPs dynamiques nécessitent des décisions aussi rapidement que possible. Cela nécessite des méthodes de résolution avec une grande efficacité de calcul, en particulier pour les problèmes avec un grand nombre de clients.
L'objectif de cette section est d'atteindre cet objectif. Pour cela nous avons conçu un algorithme génétique (GA) pour la résolution de grands DVRP avec une approche de ré-optimisation périodique. L'idée de base est de subdiviser le jour ouvrable en plusieurs périodes et de résoudre le DVRP en une série de VRP statiques, un VRP statique pour chaque période. À chaque période, de nouvelles demandes (clients) arrivent et doivent être traitées. Ces nouveaux clients et les clients qui ne sont pas déjà visités forment un VRP statique que nous résolvons en insérant les nouvelles demandes dans la solution du VRP statique au cours de la période précédente et en améliorant la solution obtenue par le GA.
Nous avons évalué l'efficacité de notre approche en la comparant à cinq travaux récents sur le DVRP : l'algorithme IVNS proposé par Xu et al. [149], l'algorithme génétique (GA-DVRP) proposé par Hanshar et Ombuki-Berman. [78], l'algorithme de colonies de fourmis (DVRP-ACS) proposé par Montemanni et al. [103], l'algorithme MBO proposé par Chen et al. [38], la recherche locale répétée (ILS-SP) proposée par Uchoa et al. [142]. Nous avons aussi proposé nos larges instances (jusqu'à 3000 clients).

Dans le reste de cette section, nous mettons le point sur un ensemble d'éléments qui se rapporte respectivement à la définition du problème DVRP avec ré-optimisation périodique, à la conception d'une solution initiale, à la réalisation de notre algorithme génétique, à l'implémentation de l'approche proposée sur GPU et finalement à la présentation des résultats de nos expériences [17].

### 3.6.2 Présentation du VRP dynamique avec ré-optimisation périodique

Nous avons utilisé la stratégie de ré-optimisation périodique pour résoudre le DVRP. L'idée de base est de diviser un DVRP en une série de VRPs statiques pour chaque période. Les états initiaux des VRPs statiques dans ce cas sont différents des VRPs statiques standards, car dans ce cas les véhicules commencent leurs tournées à partir de leur position actuelle et non pas à partir du dépôt. À chaque période, de nouveaux clients arrivent (après la mise en service de plusieurs véhicules) et doivent être traités.

Ces nouveaux clients et les clients qui ne sont pas déjà visités forment un VRP statique avec les caractéristiques suivantes : (1) de nouveaux clients doivent être intégrés aux tournées existantes en re-planifiant une ou plusieurs tournées et / ou en créant des nouvelles, (2) chaque véhicule est positionné au dernier client visité par ce véhicule, (3) les capacités des véhicules diminuent après chaque période (la capacité restante après avoir servi tous les clients précédemment visités). À la première période, la capacité de chaque véhicule est $C$.
Cette stratégie a été utilisée dans de nombreux travaux DVRP. Xu [149] a proposé un algorithme de recherche de voisinage variable amélioré (IVNS) pour le VRP statique à chaque période. Montemanni et al. [103] ont mis au point un système de colonies de fourmis (ACS) pour ces VRPs statiques. Hanshar et al. [78] a proposé un algorithme génétique qui résout la séquence des VRPs (à chaque période). Le problème est d'incorporer dans un délai minimal les nouveaux clients aux tournées déjà planifiées sans diminuer la qualité de la solution (en supposant que les tournées déjà planifiées forment une "bonne" solution).

Soient $R_1, ..., R_m$ la solution (tournées) du VRP statique (avant le début de la journée de travail). Rappelez vous, $m$ le nombre de tournées, $n$ le nombre de clients $1, ..., n$ avec le dépôt 0 et $nr$ est le nombre de nouveaux clients. Soit $R_i = (0, u_1, ..., u_k, 0)$, $demande(R_i) = d_{u_1} + ... + d_{u_k}$ et coût$(R_i) = c_{0u_1} + c_{u_1u_2} + ... + c_{u_k0}$. Le coût de la solution est $\sum_{i=1}^{m}$ coût$(R_i)$.

La figure 3.21 donne un exemple de DVRP avec $n = 7$ clients à visiter pendant la période 0 et $nr = 3$ nouveaux clients (demandes) à visiter pendant la période 1. La capacité des véhicules est $C = 40$. La période 1 commence avec le véhicule 1 positionné au nœud 1, le véhicule 2 au nœud 3 et le véhicule 3 au nœud 6.

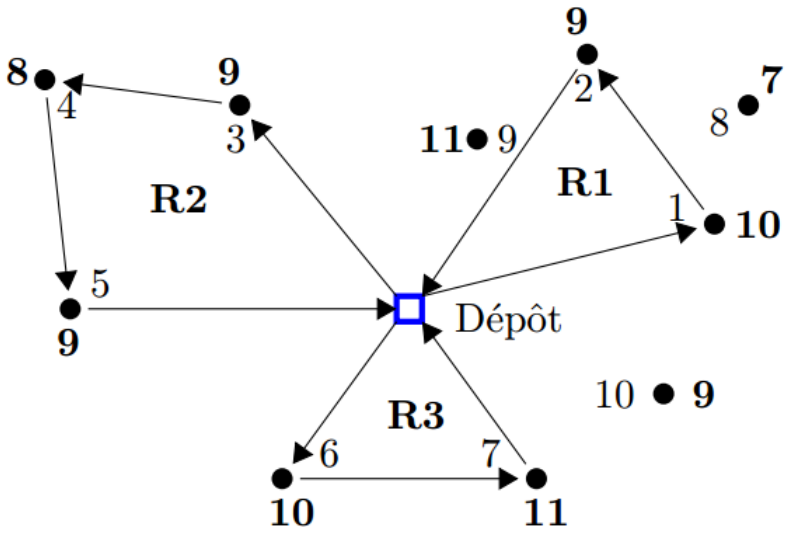



FIGURE 3.21 – Exemple de DVRP.

Nous proposons de résoudre le VRP statique à chaque période en deux étapes :

- *Étape 1 :* Insérer les nouveaux clients dans les tournées déjà planifiées à la période précédente, créer éventuellement de nouvelles tournées. La solution obtenue est utilisée pour produire une population initiale pour GA.
- *Étape 2 :* Améliorer la solution initiale en utilisant un GA avec une population initiale générée à partir de la solution initiale.

Dans la suite, nous détaillons ces deux étapes.

### 3.6.3 Solution initiale

Trier la liste $L$ des nouveaux clients par ordre croissant en fonction de leurs demandes. Autrement dit, $L_1$ devient le client avec la plus petite demande et $L_{nr}$ le client avec la plus grande demande. Ces nouveaux clients sont traités dans l'ordre $L_1$ à $L_{nr}$. L'insertion de petites requêtes permettrait d'insérer un maximum de tels clients dans les tournées déjà planifiées sans créer de nouvelles tournées (cela limite le nombre de véhicules).

Chaque client $L_i$ sera inséré dans sa tournée la plus proche $R_j$ si la capacité du véhicule le permet, c-à-d si la $demande(R_j) + d_{L_i} \leq C$, avec $C$ la capacité du véhicule desservant $R_j$. Sinon, il sera inséré dans la deuxième tournée la plus proche si cela ne viole pas la contrainte de capacité, etc. D'autre part, $L_i$ est inséré entre deux clients consécutifs $u^*$ et $v^*$ qui minimisent $c_{uL_i} + c_{L_iv} - c_{uv}$ pour tout $u$, $v$ deux

clients consécutifs de la tournée. Si $L_i$ ne peut pas être inséré sur aucune tournée, une nouvelle tournée sera créée pour lui (0, $L_i$, 0).

La figure 3.22 montre la solution initiale $\{R_1, R_2, R_3\}$ à la période 1 du problème représenté sur la figure 3.21. Nous obtenons cette solution en insérant les 3 nouveaux clients 8,9,10 dans la solution $\{R_1, R_2, R_3\}$ comme suit. On a la liste triée $L = \{8, 10, 9\}$. Donc on commence d'abord par insérer le client 8 dans $R_1$ car $R_1$ est la tournée la plus proche au client 8 et car la capacité du véhicule desservant $R_1$ permet cette insertion. Ensuite, insérer le client 10 dans $R_3$ parce que $R_3$ est la tournée la plus proche au client 10 et la capacité du véhicule desservant $R_3$ permet cette insertion. Enfin, insérer le client 9 dans $R_1$ car $R_1$ est la tournée la plus proche au client 9 et la capacité du véhicule desservant $R_1$ permet cette insertion. Notons que la période 1 commence avec le véhicule 1 positionné au nœud 1, le véhicule 2 positionné au nœud 3 et le véhicule 3 positionné au nœud 6. Cette solution initiale sera améliorée par le GA.

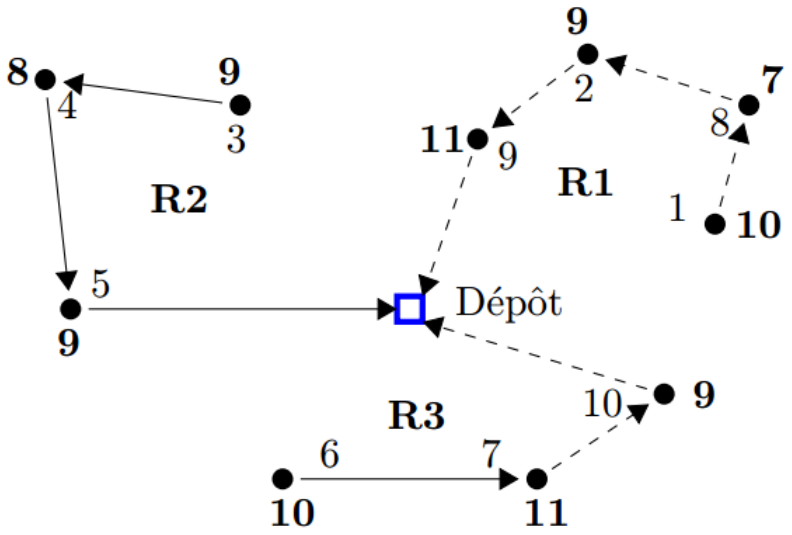



FIGURE 3.22 – Solution initiale du DVRP de la figure 3.21.

### 3.6.4 Algorithme génétique (GA)

L'algorithme génétique est une sorte de méthode évolutive qui simule un processus naturel d'évolution biologique. Dans un GA, une nouvelle population est produite à partir de la population précédente en utilisant des opérateurs génétiques de manière itérative (généralement avec un arrêt après un nombre donné d'itérations). Un

aspect important de GA est le choix de la représentation de la solution ou le codage chromosomique.

### Représentation de la solution

La plupart des GAs proposés dans la littérature pour résoudre le VRP, utilisent un tableau d'entiers comme codage, chaque géne représente un client afin de spécifier l'ordre de passage des clients. Ici, nous utilisons la représentation "Set-Based-Representation" : une représentation basée sur des clusters introduite par Naeem et Ombuki-Berman [104] (utiliser pour le problème de localisation des hubs) afin d'encoder une solution du DVRP comme suit : (1) Chaque tournée $R_i$ de la solution est représentée par un ensemble ordonné des clients (cluster) $C_i$ qui constituent la tournée. Nous disons le coût($C_i$) au lieu du coût($R_i$). (2) la solution est codée par les clusters $C_1, ..., C_m$. Le premier numéro de chaque cluster est utilisé pour identifier la position du véhicule. Ce codage donne explicitement les tournées de la solution. Donc, il ne nécessite pas d'étape de décodage.

Par exemple, le codage de la solution de la figure 3.22 est :
$C_1$ = (1,8,2,9,0), $C_2$ = (3,4,5,0), $C_3$ = (6,7,10,0) ce qui signifie que les positions initiales des véhicules sont 1,3,6 et que les clients 1,3,6 sont déjà visités.

### Les opérateurs génétiques

Nous définissons les opérateurs génétiques simples qui sont appliqués de manière itérative au codage de la solution initiale afin de l'améliorer.

1. *Population initiale* : La population initiale du GA est générée à partir de la solution initiale $p$ comme suit :
   - Sélectionner au hasard deux clusters $C$ et $C'$ de $p$ et permuter un client de $C$ avec un autre client de $C'$ (respectant la contrainte de capacité), le résultat est une autre solution réalisable (individu).
   - Itérer $2R$ fois ce processus pour produire une population initiale de $2R$ individus.

2. *Croisement* : Le Croisement est l'opérateur principal d'une GA et le plus compliqué à définir. Les quatre étapes suivantes définissent le croisement de deux solutions (individus) $p$ et $p'$.
   Étape 1. Sélectionner au hasard un cluster $C$ à partir de $p$ et $C'$ à partir de $p'$.
   Étape 2. Faire une découpe aléatoire en deux points aux clusters sélectionnés $C$ et $C'$. Soit $X$ (resp. $X'$) l'ensemble des clients entre les deux points de coupure de $C$ (resp. de $C'$). Fondamentalement, $X$ (resp. $X'$) est la coupe en $C$ (resp.

en $C'$). La longueur des coupes (nombre de clients) $X$ et $X'$ doit être la même. Le premier client de $C$ et le premier client de $C'$ ne doivent pas faire partie des coupes. Pour des raisons de simplicité, nous limitons la longueur de la coupe à 1 ou 2 ou 3. Cela permettra de permuter soit un seul client, soit un arc de 2 clients, soit un chemin de 3 clients entre $C$ et $C'$.

Étape 3. Remplacer $X$ par $X'$ dans $C$ et $X'$ par $X$ dans $C'$, si cela ne viole pas la contrainte de capacité en produisant deux solutions (enfants) $ch$ et $ch'$. Sinon, refaire l'étape 1.

Étape 4. Les clients dans $Y = X \backslash X \cap X'$ apparaissent deux fois dans $ch'$ et les clients dans $Y' = X' \backslash X \cap X'$ apparaissent deux fois dans $ch$. Cela doit être corrigé comme suit. Pour chaque $i \in Y$, trouver $j \in Y'$ et remplacer $i$ par $j$ dans $ch'$ si cela ne viole pas la constance de la capacité et mettre $Y' = Y' \backslash \{j\}$. Remplacer les doublons dans $ch$ de la même manière. Si ces remplacements s'avèrent impossibles à cause de la contrainte de capacité, refaire l'étape 1.

Un exemple du croisement proposé est illustré à la figure 3.23. $C_2$ est sélectionné à partir de $p$ et $C'_1$ à partir de $p'$. La longueur de la coupe est égale à 2. La coupe en $C_2$ est $X = \{4, 5\}$ et la coupe en $C'_1$ est $X' = \{1, 4\}$. Donc, $Y = \{5\}$ et $Y' = \{1\}$ signifie que le client 5 (resp. client 1) apparaît deux fois dans $p'$ (resp. dans $p$) et doit être remplacé par 1 (resp. par 5).

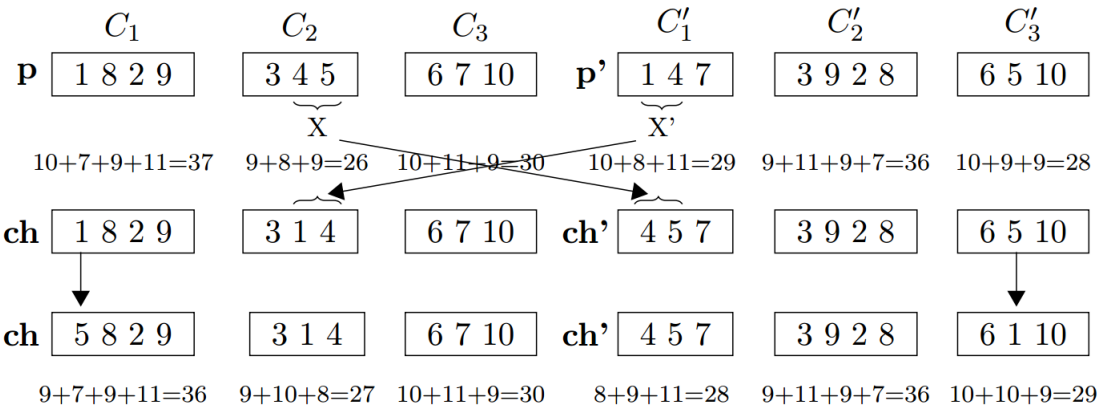



FIGURE 3.23 – Un exemple de croisement.

3. *Mutation* : L'opérateur de mutation contribue à maintenir un haut niveau de diversité parmi les individus pour éviter une convergence prématurée de l'algorithme vers un extremum local (minimum local). La mutation, que nous appliquons à 2% de la population actuelle, consiste à déplacer un client d'un cluster vers un autre cluster de la même solution. Une solution non réalisable

est rejetée et deux autres clusters / clients sont choisis de manière aléatoire. Les deux clusters sont sélectionnés aléatoirement parmi les clusters qui ont un nombre minimal de clients (le client est également sélectionné de manière aléatoire), afin de fusionner les tournées qui contiennent moins de clients. Les clusters modifiés doivent être corrigés comme suit.

4. *Correction* : Les opérateurs de croisement et de mutation déplacent certains clients vers d'autres clusters (tournées) ; L'opérateur de correction a pour but de réinsérer ces clients dans les meilleures positions de ces nouvelles tournées. Soit $k$ un client déplacé vers un cluster $C$. Trouvons alors $u^*$ et $v^*$ dans $C$ qui minimisent $c_{uk} + c_{kv} - c_{uv}$ pour tout $u$ et $v$ dans $C$ et insèrent $k$ entre $u^*$ et $v^*$. Faites ceci pour tous les clusters où certains clients sont déplacés. Si plusieurs clients ont été déplacés vers le même cluster, les meilleures positions pour insérer ces clients sont déterminées séquentiellement.

   Le GA est exécuté à chaque période de la journée de travail. Les clients visités à la période précédente sont supprimés de la représentation chromosomique et les nouveaux clients de la période en cours sont ajoutés à la représentation chromosomique. Par conséquent, la taille des chromosomes reste généralement la même au cours des périodes.

5. *Évaluation d'un individu (Fitness)* : A chaque itération du GA, les deux enfants produits par chaque paire de parents sont obtenus en ne modifiant que quelques clusters des deux parents. Ainsi, dans la même itération, les deux enfants ne changent pas beaucoup par rapport à leurs deux parents. Par conséquent, nous stockons avec les chromosomes $(C_1, ..., C_m)$ des parents les coûts (coût$(C_1)$, ..., coût$(C_m)$) et nous ne recalculons que les coûts des clusters modifiés durant cette itération.

### 3.6.5 Implémentation sur GPU du GA pour le VRP dynamique

Étant donné une bonne solution $s = C_1, ..., C_m$ du VRP statique (avant l'arrivée des $nr$ nouveaux clients) avec $m$ tournées et $n$ clients. Copier la matrice de distance $(n + nr) \times (n + nr)$ sur la mémoire globale du GPU, ni la mémoire partagée ni la mémoire constante ne peuvent stocker cette matrice pour une grande instance du VRP. Définir un bloc de threads et copier les chromosomes $(C_1, ..., C_m)$, les $n$ clients statiques et les $nr$ clients dynamiques sur la mémoire partagée de ce bloc.

- D'abord, générer une nouvelle solution réalisable en insérant les nouveaux clients dans $s$, en commençant par les clients avec les petites demandes. Cette étape est mise en œuvre sur le bloc précédemment défini comme suit.
  Thread $i$ calcule les clusters les plus proches au $i^{\text{ème}}$ nouveau client et coopère avec les autres threads pour insérer ce $i^{\text{ème}}$ nouveau client dans la meilleure position de son cluster le plus proche (tournée). Si un nouveau client $k$ ne peut pas être inséré dans aucun cluster (en raison de la contrainte de capacité), la création d'un nouveau cluster (0, $k$, 0) est effectué. Ceci constitue la solution initiale $p$ du DVRP. Le résultat est la solution initiale p composée d'au plus $m + nr$ clusters.
- La solution $p$ est améliorée par le GA comme suit :
  Définir les $R$ blocs $bloc_0, ..., bloc_{R-1}$ chacun avec $M$ threads où $M$ sera défini plus tard.
- Générer une population initiale à partir de $p$, en sélectionnant aléatoirement deux clusters $C$ et $C'$ à partir de $p$ et en échangeant un client de $C$ avec un autre de $C'$, pour aboutir à une autre solution. Une solution irréalisable est rejetée et deux autres clusters/clients sont choisis au hasard. Itérer $2R$ fois ce processus pour produire une population initiale de $2R$ individus notée $p_0, p'_0, ..., p_{R-1}, p'_{R-1}$. Le $bloc_i$ génère $p_i$, $p'_i$, cela se fait en parallèle par deux threads de chaque bloc.
- Chaque paire de parents $p_i$, $p'_i$ est croisée sur le $bloc_i$ pour produire deux enfants $ch_i$, $ch'_i$ (notons qu'il n'y a pas de sélection des parents). Ensuite, $ch_i$, $ch'_i$ sont corrigés sur le $bloc_i$. Chaque bloc sélectionne les deux individus notés par $q_i$, $q'_i$ qui ont le meilleur coût parmi $p_i$, $p'_i$, $ch_i$, $ch'_i$. L'opérateur de mutation est appliqué ensuite à 2% de la population $q_0$, $q'_0$, ..., $q_{R-1}$, $q'_{R-1}$. Le $bloc_i$ effectue la mutation (le cas échéant) à $q_i$ et/ou à $q'_i$ et identifie les meilleurs des deux solutions obtenues produisant ainsi la prochaine génération d'individus ($2R$ individus). Notons que tous les couples de parents ($p_i$, $p'_i$) dans la population sont traités, donc pas d'opérateur de sélection.

Chaque couple de parents ($p_i$, $p'_i$) est généré par un bloc de $m_i + m'_i$ de threads où $m_i$ (resp. $m'_i$) est le nombre de clusters de $p_i$ (resp. en $p'_i$), y compris les nouveaux

clients. Ainsi, tous les parents (la population entière) sont générés en parallèle sur les $R$ blocs, chacun de $M$ threads où $M = max_i(m_i + m'_i)$ car, dans cuda les blocs doivent être uniformes (de même taille).

Pour calculer rapidement la fitness, le chromosome $(C_1, ..., C_m)$ du parent et les coûts (coût($C_1$), ..., coût($C_m$)) sont stockés dans la mémoire partagée du bloc. Le bloc recalcule uniquement le coût des clusters modifiés durant cette itération.

#### Paramètres de l'implémentation GPU

Pour profiter de la façon dont fonctionne le GPU, nous définissons les paramètres suivants pour notre implémentation GPU. Soit $P$ le nombre de SM du GPU ($P = 2$ pour notre GPU modeste). Nous prenons $R$ multiple de $P$. Ainsi, chaque SM exécute séquentiellement $R/P$ blocs, ce qui implique que le temps d'exécution sera proportionnel à $R/P$. D'autre part, la taille de la population du GA est $2R$ et devrait augmenter avec la taille des instances et le nombre de blocs alloués à chaque SM ($R/P$) doivent être petits. Nous prenons donc $R = 96$ pour les petites instances (48 blocs par SM), $R = 192$ pour les instances moyennes (96 blocs par SM) et $R = 288$ pour les grandes et les très grandes instances (144 blocs par SM). Cela permet de traiter une population de 576 individus en parallèle.

En outre, la représentation chromosomique de quatre individus et le coût de chaque clusters doivent être stockés dans la mémoire partage (48ko). Rappelons que $n$ est la taille de l'instance et $M$ le nombre maximum de tournées par solution ($M$ doit être de préférence multiple de 32). Alors nous avons la relation :

$$4(n \times sizeof(int) + M \times sizeof(float)) \leq 49152 \; octets$$

Avec $sizeof(int) = 2$ octets et $sizeof(float) = 4$ octets, nous obtenons la relation $n + M \times 2 \leq 6144$. Nous prenons $M = 256$ pour les grandes instances que nous avons adaptées au DVRP, donc $n$ doit être inférieure à 5632 (la plus grande taille d'instance que notre matériel peut la résoudre). Le nombre d'itérations (critère d'arrêt de GA) varie en fonction de la taille des instances et est limité à 200 pour les grandes instances.

### 3.6.6 Résultats expérimentaux

Afin d'évaluer les performances de notre implémentation GPU, quatre problèmes de test relatifs à différentes tailles (petites, moyennes, grandes et très grandes) ont été résolus. Nous avons utilisé (1) des instances de petites tailles de Xu et al [149] (2) les instances de tailles moyennes proposées par Montemanni et al. [103] et ceux

compilés par Solomon [126] avec des tailles variant entre 50 à 150 (3) les grandes instances générées par Uchoa et al. [142] avec des tailles allant de 100 à 1000 (4) nos très grandes instances avec des tailles allant de 1000 à 3000 clients. Nous comparons nos résultats à ceux de cinq méthodes : l'algorithme IVNS [149], l'algorithme génétique (GA-DVRP) [78], l'algorithme de colonies de fourmis (ACS) [103], l'algorithme MBO [38] et l'algorithme de recherche locale itérative (ILS-SP) [142].

Les résultats reportés sont pris sur 10 exécutions de l'implémentation GPU. Pour tous les tests, les positions du véhicule au début de la première période sont supposées être au dépôt.

### Les instances de petites tailles

Xu [149] a proposé un exemple où l'aire de distribution est un carré de $50km \times 50km$ ; 30 clients à la demande statique et 10 nouvelles demandes générées aléatoirement. Chaque demande des clients est générée aléatoirement à partir de l'intervalle [0, 2], la capacité des véhicules est égale à $8t$, l'emplacement du centre de distribution (dépôt) est $(25km, 25km)$. Les coordonnées et les demandes des clients statiques et des nouveaux clients sont présentées dans les tableaux 3.2 et 3.3.

Tableau 3.2 – La position et la demande des clients statiques.

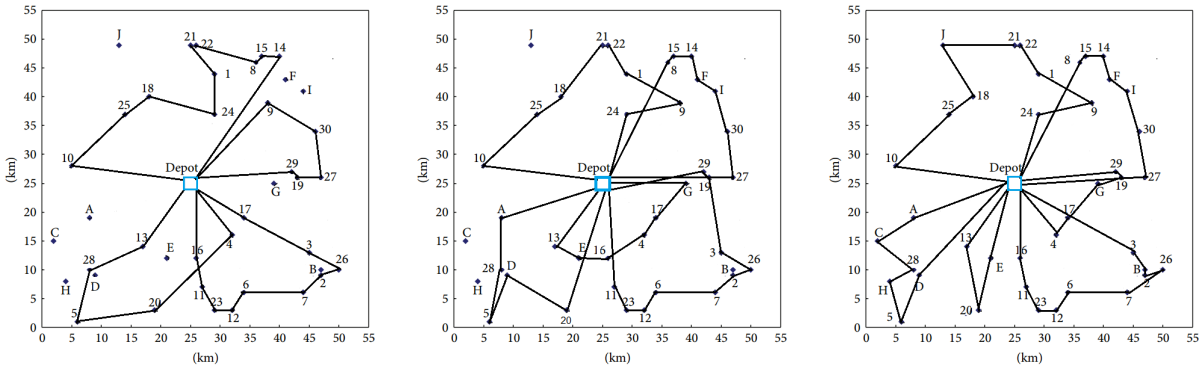
| No. | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| X | 29 | 47 | 45 | 32 | 6 | 34 | 44 | 36 | 38 | 5 | 27 | 32 | 17 | 40 | 37 |
| Y | 44 | 9 | 13 | 16 | 1 | 6 | 6 | 46 | 39 | 28 | 7 | 3 | 14 | 47 | 47 |
| D | 0.5 | 0.2 | 0.4 | 2 | 1.7 | 0.3 | 1.7 | 0.7 | 0.2 | 1.2 | 1.7 | 0.5 | 0.3 | 1.5 | 0.4 |
| No. | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 |
| X | 26 | 34 | 18 | 43 | 19 | 25 | 26 | 29 | 29 | 14 | 50 | 47 | 8 | 42 | 46 |
| Y | 12 | 19 | 40 | 26 | 3 | 49 | 49 | 3 | 37 | 37 | 10 | 26 | 10 | 27 | 34 |
| D | 0.8 | 1.4 | 1.4 | 1 | 1.7 | 0.2 | 0.9 | 0.3 | 0.2 | 0.5 | 0.3 | 0.9 | 0.5 | 1.4 | 2 |

Tableau 3.3 – La position et la demande des nouveaux clients.

| No. | A | B | C | D | E | F | G | H | I | J |
|---|---|---|---|---|---|---|---|---|---|---|
| X | 8 | 47 | 2 | 9 | 21 | 41 | 39 | 4 | 44 | 13 |
| Y | 19 | 10 | 15 | 9 | 12 | 43 | 25 | 8 | 41 | 49 |
| Period | 1 | 2 | 2 | 1 | 1 | 1 | 1 | 2 | 1 | 2 |
| D | 1 | 1.5 | 1.6 | 3 | 1.4 | 0.3 | 0.2 | 2.2 | 2.8 | 1 |

Au cours de la période 1, six nouveaux clients (A, D, E, F, I, G) arrivent et seront incorporés dans la solution obtenue avant le début de la journée de travail.

A la période 2, les quatre nouveaux clients (B, C, H, J) seront incorporés dans la solution de la période 1. Les figures suivantes montrent la position spécifique des clients et la planification des tournées pour les trois étapes : la distribution initiale, la solution de la première période et la solution de la deuxième période obtenue par notre implémentation GPU. Notons que dans les solutions de Xu, la tournée numéro 4 dans la solution à la période 1 et les tournées 1 et 4 à la période 2 dépassent la capacité du véhicule fixée à 8t [149].




(a) Solution initiale (b) Solution pour la période 1 (c) Solution pour la période 2

### Les instances de tailles moyennes

Ces instances de tailles moyennes ont été compilées par Salomon [126]. Les instances sont divisées en six catégories : R1, R2, C1, C2, RC1 et RC2, chacune contenant 100 clients. Nous reportons dans le tableau. 3.4 les résultats pour le DVRP sans fenêtre temps pour les cinq instances de données avec un degré dynamique de 90%, 70%, 50%, 30% et 10%. Le degré dynamique est défini comme suit :

$$Dod = \frac{nr}{n + nr} \times 100\%$$

Chaque instance est résolue en deux périodes, en fonction du Dod les premiers clients dans l'instance sont considérés comme des clients statiques et ils sont traités à la période 0. Concernant le reste des clients, ils sont considérés comme des clients dynamiques et sont traités à la période 1. Les véhicules à la période 0 sont positionnés au dépôt. Parmi les 30 instances, nous obtenons 20 nouvelles meilleures solutions (mentionné en gras dans le tableau. 3.4). Pour les 10 autres instances, nous obtenons des coûts entre ceux de Xu et de Lackner [149]. Notre implémentation a permis une amélioration de 73,3% par rapport aux résultats de Xu et de 86,6% par rapport aux résultats de Lackner reportés dans [149].

TABLEAU 3.4 – Comparaison entre les résultats du GPU, du IVNS et du Lackner [149].

| Instance | Dod (%) | IVNS [149] | Lackner [149] | GPU Sol |
|---|---|---|---|---|
| R1 | 90 | 1343.82 | 1278.33 | 1332.39 |
| | 70 | 1331.35 | 1336.10 | **1259.96** |
| | 50 | 1293.81 | 1329.98 | **1200.92** |
| | 30 | 1286.63 | 1337.86 | **1006.53** |
| | 10 | 1259.18 | 1278.06 | **968.001** |
| C1 | 90 | 1235.47 | 1479.60 | 1346.15 |
| | 70 | 1031.78 | 1261.30 | 1180.76 |
| | 50 | 1072.32 | 1236.06 | **1072.19** |
| | 30 | 970.80 | 1066.89 | **947.27** |
| | 10 | 895.68 | 996.35 | **830.15** |
| RC1 | 90 | 1506.43 | 1475.21 | 1506.29 |
| | 70 | 1513.23 | 1488.44 | **1352.95** |
| | 50 | 1519.12 | 1448.07 | **1293.70** |
| | 30 | 1489.96 | 1439.71 | **1253.40** |
| | 10 | 1431.79 | 1426.89 | **1123.75** |
| R2 | 90 | 1045.89 | 1193.33 | 1129.94 |
| | 70 | 1034.22 | 1116.93 | 1112.84 |
| | 50 | 1012.11 | 1138.78 | **957.15** |
| | 30 | 987.42 | 1085.42 | **943.96** |
| | 10 | 951.00 | 1052.85 | **803.48** |
| C2 | 90 | 695.50 | 792.46 | **693.50** |
| | 70 | 671.86 | 743.78 | 675.42 |
| | 50 | 610.98 | 689.25 | 661.19 |
| | 30 | 655.23 | 632.33 | 649.12 |
| | 10 | 582.32 | 629.08 | 643.52 |
| RC2 | 90 | 1351.21 | 1476.76 | **1312.85** |
| | 70 | 1256.32 | 1346.76 | **1190.74** |
| | 50 | 1189.13 | 1269.29 | **1151.82** |
| | 30 | 1151.35 | 1244.85 | **1072.65** |
| | 10 | 1183.11 | 1220.90 | **711.54** |
| | Total | 33559 | 35511 | **31384** |

Dans le tableau. 3.5 nous comparons notre implémentation sur GPU avec l'algorithme de colonies de fourmis (DVRP-ACS) proposé par Montemanni et al. [103] et avec l'algorithme génétique (DVRP-GA) proposé par Hanshar et al. [78].
Trois sources du VRP distinctes, à savoir tail* (pour Taillard et al. [131]), c* (pour Christophides et Beasley [40]) et f* (pour Fisher et al. [61]) sont considérées (le * est remplacé par la taille de l'instance).

Les résultats montrent que l'implémentation sur GPU a permis de trouver 18 nouvelles meilleures solutions (sur 21) par rapport aux DVRP-ACS et DVRP-GA et d'améliorer tous les résultats proposés par Montemanni et al. [103].

Tableau 3.5 – Comparaison entre les résultats du GPU, du DVRP-ACS [103] et du DVRP-GA [78].

| Instance | DVRP-ACS | DVRP-GA | GPU Sol |
|---|---|---|---|
| c100 | 973.26 | 961.10 | **918.81** |
| c100b | 944.23 | 881.92 | **830.15** |
| c120 | 1416.45 | 1303.59 | **1068.14** |
| c150 | 1345.73 | 1348.88 | **1209.20** |
| c199 | 1771.04 | 1654.51 | **1456.67** |
| c50 | 631.30 | 570.89 | 580.67 |
| c75 | 1009.36 | 981.57 | **900.26** |
| f134 | 15135.51 | 15528.81 | **12220.69** |
| f71 | 311.18 | 301.79 | **256.11** |
| tai100a | 2375.92 | 2232.71 | **2184.83** |
| tai100b | 2283.97 | 2147.70 | **2033.33** |
| tai100c | 1562.30 | 1541.28 | **1394.32** |
| tai100d | 2008.13 | 1834.60 | **1707.11** |
| tai150a | 3644.78 | 3328.85 | 3519.65 |
| tai150b | 3166.88 | 2933.40 | **2888.06** |
| tai150c | 2811.48 | 2612.68 | **2457.22** |
| tai150d | 3058.87 | 2950.61 | 2960.36 |
| tai75a | 1843.08 | 1782.91 | **1645.49** |
| tai75b | 1535.43 | 1464.56 | **1356.63** |
| tai75c | 1574.98 | 1440.54 | **1313.54** |
| tai75b | 1472.35 | 1399.83 | **1356.63** |
| Total | 50876.23 | 49202.82 | **44257.87** |

Plus récemment, Chen et al. [37] ont proposé une nouvelle approche pour le DVRP appelée Monarch Butterfly Optimization (MBO). Les auteurs ont testé cette approche sur les instances précédentes, mais ils ne spécifient pas les demandes statiques/dynamiques. Nous avons considéré le cas extrême où toutes les demandes sont dynamiques, nous avons amélioré certains de leurs résultats. A titre d'exemple, nous donnons dans le tableau. 3.6 la solution de l'instance f71 obtenue par le GPU et par l'approche MBO.

TABLEAU 3.6 – Comparaison entre la solution du MBO [38] et la solution du GPU de l'instance f71.

| MBO solution [38] | GPU solution |
|---|---|
| $R_1$=0-14-11-18-35-0 | $R_1$=0-18-11-35-0 |
| $R_2$=0-1-15-19-2-13-17-16-12-71-6-10-5-3-8-4-7-9-0 | $R_2$=0-9-7-4-8-3-5-10-6-27-44-42-43-46-53-45-52-48 -47-50-70-51-49-25-24-26-23-0 |
| $R_3$=0-60-61-58-59-63-62-64-65-66-67-69-37-38-40-68-39-57-56-34-0 | $R_3$=0-32-34-60-61-58-59-62.53-63-62-64-65-66-67-69-37-38-40-68-39-57-56-41-55-54-0 |
| $R_4$=0-49-51-70-50-47-48-52-45-53-46-43-44-42-27-28-22-21-30-0 | $R_4$=0-20-29-30-21-22-28-27-44-42-43-46-53-45-52-48-47-50-70-51-49-25-24-26-23-0 |
| $R_5$=0-20-29-23-26-24-25-41-55-54-32-31-33-36-0 | $R_5$=0-33-36-0 |
| **Coût 271.43** | **Coût 256.11** |

### Les instances de grandes tailles

Récemment, Uchoa et al. [142] ont proposé un ensemble de 100 instances plus réalistes avec des tailles de 100 à 1000 clients. Ces instances varient en fonction de la taille et de quatre paramètres qui affectent indépendamment la position du dépôt, les positions des clients, les demandes et la longueur moyenne des tournées. Comme indiqué dans [142], ces instances permettent de mieux évaluer les performances des approches par rapport aux différentes entités, permettant de mieux distinguer les algorithmes concurrents.

Les distances sont euclidiennes bidimensionnelles. Le dépôt et les clients ont des coordonnées entières générés dans l'intervalle $[0, 1000] \times [0, 1000]$. Chaque instance est caractérisée par les attributs suivants : nombre de clients, positionnement du dépôt, positionnement des clients, distribution des demandes et la taille moyenne de

TABLEAU 3.7 – Résultats du GPU pour les grandes instances de Uchoa

| # | Instance | n | Dép | Client | Dem | Coût de la solution | Temps GPU(s) |
|---|---|---|---|---|---|---|---|
| 11 | X-n148-k46 | 147 | R | RC(7) | 1-10 | **32946.36** | 0.64 |
| 20 | X-n190-k8 | 189 | E | C(3) | 1-10 | **17136.42** | 1.19 |
| 26 | X-n219-k73 | 218 | E | R | U | **100191.75** | 1.51 |
| 32 | X-n247-k50 | 246 | C | C(4) | SL | **32202.90** | 1.79 |
| 36 | X-n266-k58 | 265 | R | RC(6) | 5-10 | **73998.98** | 2.01 |
| 47 | X-n317-k53 | 316 | E | C(4) | U | **76568.83** | 2.83 |
| 69 | X-n502-k39 | 501 | E | C(6) | U | **66636.25** | 9.57 |
| 75 | X-n573-k30 | 572 | E | C(3) | SL | **49818.60** | 16.98 |
| 90 | X-n801-k40 | 800 | E | R | U | 81123.57 | 73.81 |
| 91 | X-n819-k171 | 818 | C | C(6) | 50-100 | 172215.51 | 77.10 |
| 92 | X-n837-k142 | 836 | R | RC(7) | 5-10 | 201273.73 | 84.77 |
| 93 | X-n856-k95 | 855 | C | RC(3) | U | 102658.58 | 90.23 |
| 94 | X-n876-k59 | 875 | E | C(5) | 1-100 | 101081.64 | 96.67 |
| 95 | X-n895-k37 | 894 | R | R | 50-100 | 70399.38 | 104.78 |
| 96 | X-n916-k207 | 915 | E | RC(6) | 5-10 | 338097.96 | 116.23 |
| 97 | X-n936-k151 | 935 | C | R | SL | 155051.89 | 127.55 |
| 98 | X-n957-k87 | 956 | R | RC(4) | U | 101118.41 | 140.36 |
| 99 | X-n979-k58 | 978 | E | C(6) | Q | 123092.28 | 159.78 |
| 100 | X-n1001-k43 | 1000 | R | R | 1-10 | 82142.17 | 176.42 |

la tournées (voir [142] pour plus de détails sur ces instances).

Les auteurs dans [142] reportent les résultats de l'algorithme ILS-SP de Subramanian et al. [129] et de la recherche génétique hybride unifiée (UHGS) de Vidal et al. [146] sur ces instances (pour le VRP statique). Dans le tableau. 3.7, nous reportons les résultats de notre implémentation, pour les instances entre 800 et 1000 clients, où nous supposons que, pour chaque instance les premiers 50% des clients sont statiques (période 0) et les derniers clients sont dynamiques (période 1). Les résultats des autres instances de notre implémentation sont disponibles dans [17]). Les résultats de [142] concernent le VRP statique et devraient être meilleurs que les nôtres, mais nous en avons amélioré 8 instances (les résultats en gras apparaissent dans tableau. 3.7).

### Les instances de très grandes tailles

Nous avons généré nos propres instances. Ces nouvelles instances complexes sont constituées de très grands réseaux avec 1000 à 3000 clients générés comme suit :

- Les coordonnées des clients sont générés aléatoirement dans l'intervalle $[0, 1000] \times [0, 1000]$,
- Pour chaque instance, 50% des demandes sont statiques et 50% sont dynamiques générées aléatoirement,
- Chaque demande est un nombre aléatoire entre [1, 200],
- L'emplacement du dépôt de distribution est positionné à (500, 500).

Tous les nouveaux clients, d'un problème d'instance, sont traités dans la période 1. Nous présentons dans le tableau. 3.8 les coûts des solutions obtenu sur GPU et les temps d'exécution correspondants qui augmentent de manière quasi-linéaire en fonction de $n$.

Tableau 3.8 – Résultats du GPU pour les instances de très grandes tailles.

| n | GPU Sol | Temps GPU(s) | n | GPU Sol | Temps GPU(s) |
|---|---|---|---|---|---|
| 1000 | 70626.21 | 187.28 | 1100 | 68488.65 | 108.10 |
| 1200 | 79750.60 | 174.45 | 1300 | 81232.98 | 242.53 |
| 1400 | 83799.35 | 343.84 | 1500 | 88531.78 | 448.99 |
| 1600 | 88123.15 | 501.11 | 1700 | 95355.32 | 609.23 |
| 1800 | 107430.60 | 747.98 | 1900 | 111741.55 | 976.61 |
| 2000 | 119407.52 | 1181.13 | 2100 | 129343.18 | 1249.37 |
| 2200 | 141307.29 | 1270.61 | 2300 | 137464.59 | 1356.31 |
| 2400 | 144077.32 | 1493.69 | 2500 | 157625.26 | 1568.66 |
| 2600 | 147283.43 | 1674.90 | 2700 | 147292.35 | 1757.28 |
| 2800 | 170980.03 | 1885.54 | 2900 | 161612,31 | 2028.76 |
| 3000 | 189496.37 | 2132.92 | | | |

## 3.7 Conclusion

Dans ce chapitre nous avons proposé quatre implémentations sur GPU pour résoudre quatre variantes du VRP sur les GPUs.

La première implémentation est une heuristique basée sur l'algorithme CW pour résoudre le VRP unique et multi dépôts. Dans les tests de cette implémentation, on s'est focalisé sur le facteur d'accélération et les performances de calcul sur le GPU.

La deuxième implémentation est consacrée à la variante multi capacités VRP. Cette implémentation a permis d'améliorer les solutions trouvées dans la littérature et de résoudre des instances non résolues jusqu'à l'heure.

Enfin, pour la variante dynamique VRP ; nous avons présenté un algorithme et son implémentation sur GPU. L'approche consiste à insérer rapidement et efficacement, les requêtes dynamiques dans les routes déjà planifiés avec une ré-optimisation continue. Aussi, nous avons conçu un algorithme génétique basé sur le GPU pour la résolution de grandes instances DVRP (jusqu'à 3000 nœuds) avec ré-optimisation périodique. Les implémentations proposées exploitent efficacement le parallélisme et la puissance de calcul du GPU et ont permis de résoudre des instances non résolues dans la littérature en temps raisonnable et d'améliorer les solutions d'autre instances. Les implémentations proposées sur le GPU ont donné lieu à cinq publications.

Chapitre 4

# Algorithmes génétiques parallèles pour la résolution des problèmes de localisation des hubs sur GPU

## 4.1 Introduction

Les problèmes de localisation des hubs (HLPs) sont des problèmes d'optimisation combinatoire connus dans plusieurs domaines comme le transport et les télécommunications et ont une large gamme d'applications comme les systèmes de distribution postale, les réseaux de transport aérien ou terrestre. Parmi les applications réelles les plus connues, il y a les applications de livraison postale qui peuvent livrer des millions de colis par jour. L'utilisation des simples CPU, avec des méthodes séquentielles ne peut pas répondre à ce besoin surtout pour les problèmes de grande taille. D'autre part la performance des GPUs peut atteindre cet objectif !

Les algorithmes génétiques sont des approches de recherche bien connues qui sont appliquées dans le domaine de l'optimisation et produisent des solutions quasi optimales dans les grands espaces de recherche. L'implémentation de tel algorithme dans des environnements parallèles augmente énormément leurs performances et cela est déjà prouvé par Van Luong et al. [144] qui ont décrit l'augmentation de la performance des GAs sur la plate-forme CUDA et l'environnement multicœurs.

Dans ce chapitre nous allons proposer une implémentation sur GPU basée sur un algorithme génétique parallèle (GA) pour résoudre différentes variantes du problème de localisation des hubs. Le GA proposé adapte son codage, sa solution initiale, son critère de localisation des hubs susceptibles d'être des hubs dans la solution opti-

male, ses opérateurs génétiques et son implémentation à chaque variante traitée afin d'exploiter les performances du GPU. Les résultats expérimentaux obtenus comparés aux meilleures solutions connues sur tous les benchmarks, sur des instances jusqu'à 1000 nœuds et sur des instances plus importantes jusqu'à 6000 nœuds générées par nous-même. Les implémentations proposées surpassent les heuristiques les plus connues en termes de qualité de la solution et en temps d'exécution. Aussi, elles ont permis de résoudre des instances qui n'étaient pas résolus jusqu'à présent.

Dans la prochaine section, on va faire une revue de la littérature dans laquelle on va présenter certains travaux qui ont utilisé les ressources du calcul haute performance pour résoudre des variantes du HLP.

Dans la section 4.3, le GA proposé et son implémentation sur GPU pour l'USApHMP et l'USAHLP seront présentés ; la section 4.4 présente tous les améliorations effectuées sur GA et sur son implémentation pour s'adapter au CSApHMP et CSAHLP. La section 4.5 est réservée aux adaptations effectuées pour résoudre multi affectations (UMApHMP). Enfin, la section 4.6 vise à conclure tous ces travaux.

## 4.2 Revue de la littérature

Parmi les recherches effectuées en parallèle pour résoudre les variantes du HLP on cite : Crainic et al. [47] qui ont proposé une méthode coopérative de recherche VNS basée sur le mécanisme de mémoire centrale pour résoudre le p-median HLP. Garcia et al. [65] ont développé plusieurs stratégies pour la parallélisation de la métaheuristique appelée recherche par dispersion. Trois types de parallélisation ont été proposés pour le problème p-médian hub.

À cause de la nouveauté de l'utilisation de processeurs graphiques (GPUs), seules quelques implémentations sur GPU qui résolvent le HLP sont disponibles dans la littérature. Parmi ces travaux on peut citer : Lim et Ma [96] qui ont introduit un algorithme de substitution de vertex parallèle basé sur GPU pour le problème p-médian hub. Santos et al. [120] ont proposé une implémentation GPU basée sur la métaheuristique GRASP appliquée au problème p-médian hub.

Plus récemment, AlBdaiwi [6] a présenté un nouvel algorithme génétique basé sur une formulation pseudo-booléenne du problème p-médian qui est implémenté sur GPU.

## 4.3 Résolution des problèmes USApHMP et USAHLP

Un algorithme génétique parallèle (GA) implémenté sur GPU, est proposé dans cette section pour résoudre deux variantes du problème de localisation des hubs, l'USApHMP et l'USAHLP. Le GA utilise un codage binaire et entier avec des opérateurs génétiques adaptés au problème USApHMP. Il est amélioré en localisant initialement les $p$ hubs aux nœuds intermédiaires au lieu d'une solution aléatoire. Dans notre implémentation, nous générons plusieurs sous-populations que nous traitons en parallèle dans l'objectif de trouver une solution optimale de l'USApHMP. La solutions de l'USAHLP consiste à résoudre l'USApHMP pour $p = 1, ..., n$ en parallèle avec $n$ est le nombre de nœuds. Les résultats expérimentaux obtenus sont comparés aux meilleures solutions pour les benchmarks bien connues pouvant atteindre 6 000 nœuds. Ils montrent que notre approche surpasse les heuristiques les plus connues en termes de qualité de la solution et en temps d'exécution. Aussi, nous avons résolu des instances non résolues avant cette implémentation.

Dans la suite nous présentons un simple exemple de codage d'une solution de l'USApHMP. Nous discutons la création d'une solution initiale appropriée calculée plutôt qu'une solution aléatoire. Nous présentons la description du GA et son implémentation sur GPU. Ensuite nous discutons les résultats expérimentaux de l'USApHMP [22]. Enfin, nous présentons la résolution de l'USAHLP [21].

### 4.3.1 Algorithme génétique pour l'USApHMP

Les algorithmes génétiques produisent des solutions quasi optimales dans de grands espaces de recherche et doivent être exécutés dans des environnements parallèles. Van Luong et al. [144] ont décrit l'augmentation de la performance avec la mise en œuvre parallèle de GA sur la plateforme CUDA et l'environnement multicœurs.

#### Codage et solution initiale

La plupart des GAs proposés dans la littérature pour la résolution du HLPs, utilisent des tableaux binaires/entiers comme codage. Dans Kratica et al. [87, 88], une solution de l'UMAHLP et du CSAHLP sont représentées par un tableau binaire de longueur $n$ (le gène $i$ est mis à 1 si le nœud $i$ est localisé comme hub, sinon à 0).

Dans Stanimirović Z. [128], le code génétique est constitué de $n$ gènes de la forme bit | entier, chacun faisant référence à un nœud. Le premier bit de chaque gène indique si le nœud actuel est localisé comme hub ou non. L'entier dans le gène fait référence au hub affecté au nœud actuel.

Dans Erken M. [55], chaque chromosome comporte deux parties. La première partie définit l'emplacement des hubs et la deuxième partie concerne l'affectation des nœuds aux hubs. Certains travaux séparent le chromosome en deux tableaux, une méthode qui reste en fait, identique à la représentation à un tableau unique.

Les GAs proposés dans Naeem [104] utilisent deux schémas de représentation, la représentation classique des tableaux entiers et un codage basé sur des clusters, pour encoder la solution du CSAHLP.
Dans la représentation basée sur les clusters, un chromosome est une collection d'ensembles de nombres dans lesquels chaque nombre représente un nœud. Le premier nombre de l'ensemble représente un hub et les autres numéros représentent les nœuds affectés à ce hub. Nous avons déjà adapté ce codage au DVRP dans le chapitre précédent et on va l'adapter aussi pour coder une solution du problème de localisation des hubs avec multiple affectations (voir la cinquième section de ce chapitre).

Une solution réalisable de l'USApHMP se base sur deux critères : (1) sélectionner $p$ nœuds parmi les $n$ nœuds comme des hubs ; (2) chaque non-hubs nœuds doit être affecté à un seul hub. Dans notre implémentation nous utilisons des tableaux binaires ($H$) / entiers ($S$) de taille $n$ pour encoder chaque solution (individu) du problème, ce codage a été proposé par Topcuoglu et al [138], avec :

- $H$ représente les emplacements des hubs, c-à-d $H[i] = 1$ si le nœud $i$ est sélectionné comme hub, sinon $H[i] = 0$.
- $S$ représente l'affectation des nœuds aux hubs, c-à-d que $S[i] = k$ avec $k$ est le hub auquel le nœud $i$ est affecté. De plus, chaque hub est affecté à lui-même.

Dans [138], la solution initiale est générée aléatoirement. Ici, nous procédons différemment afin d'atteindre rapidement les meilleurs emplacements des hubs. Pour construire une solution initiale avec $p$ hubs, nous calculons d'abord les $p$ nœuds centraliser, c-à-d les $p$ hubs ($i$) avec la plus petite distance $c_i$ par rapport aux autres nœuds, avec $c_i = \sum_j d_{ij}$. Ainsi, les $p$ hubs seront initialement situés au milieu du réseau où se trouve la densité de trafic. Ensuite, chaque nœud est affecté à son hub le plus proche dans cette solution initiale.

**Exemple numérique :**
La figure 4.1 montre un exemple de l'USApHMP avec $n = 7$ nœuds, $p = 2$ hubs. La solution proposée sélectionne les nœuds 2 et 5 comme des hubs, car ces deux nœuds sont initialement situés au milieu du réseau où se trouve la densité de trafic. Les nœuds 1, 2 et 6 sont affectés au hub 2, tandis que les nœuds 3, 4, 5, 7 sont

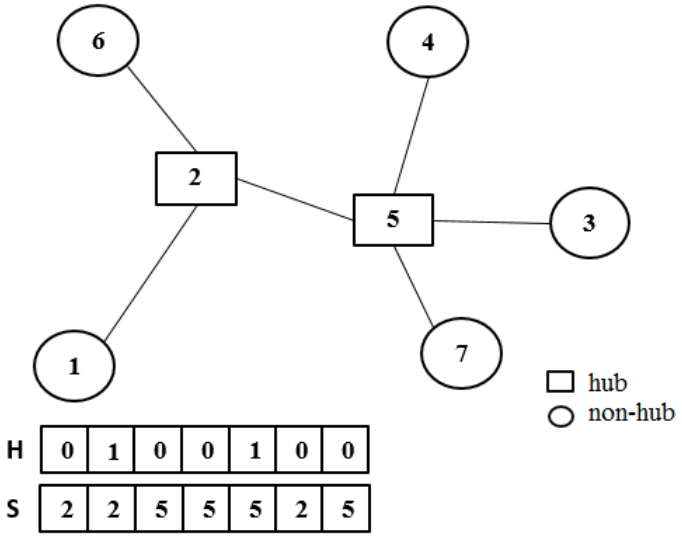



FIGURE 4.1 – Exemple de l'USApHMP avec $n = 7$, $p = 2$.

affectés au hub 5. Le codage de cette solution est donné dans figure 4.1 par les deux tableaux $H$ et $S$.

### Les opérateurs génétiques

Nous définissons les opérateurs génétiques qui sont appliqués pour résoudre l'USApHMP.

- *Population initiale* : La population initiale du GA est générée à partir de la solution initiale. On sélectionne aléatoirement un hub et le permuter avec un non-hub nœud, ensuite chaque nœud est affecté à son hub le plus proche. Le résultat est une autre solution réalisable (individu), cette opération est définie dans une fonction nommée par 1-$exchange()$. Itérer $t$ fois 1-$exchange()$ pour produire une population initiale de $t$ individus.
- *Croisement* : Un croisement en un seul point aléatoire est utilisé pour échanger les deux parties des parents $p_1$ et $p_2$ afin de générer deux descendants (enfants) $ch_1$ et $ch_2$. Si $ch_1$ (resp. $ch_2$) n'est pas une solution réalisable, alors elle sera corrigée, afin d'obtenir deux individus réalisables $c_1$ (resp. $c_2$).
- *Mutation* : L'objectif est d'agrandir l'espace de recherche et d'éviter les optimaux locaux. Nous utilisons un opérateur de mutation qui transforme 10% de l'affectation des nœuds entre hubs pour 2% des individus.

- *Correction des individus* : La contrainte sur le nombre de hubs peut être violée par l'opérateur de croisement dans le GA. Ainsi, une solution est corrigée si elle contient un nombre de hubs diffèrent de $p$. Dans notre approche, la correction est réalisée comme suit : Si une solution contient un nombre de hubs supérieur à $p$, nous ne conservons que les $p$ hubs qui se trouvent au centre. Si le nombre de hubs est inférieur à $p$, les hubs manquants sont choisis au hasard parmi les $n$ nœuds autres que les nœuds hubs.
- *Évaluation de la solution (Fitness)* : La fonction d'évaluation est utilisée pour évaluer une solution. Les définitions des fonctions d'évaluation utilisées pour différents types de benchmarks sont disponible dans [22].

Ces opérateurs sont notés respectivement par *croisement*, *mutation*, *correction* et $fitness()$.

### 4.3.2 Implémentation du GA pour l'USApHMP

Les paramètres suivants sont utilisés : $n$ le nombre de nœuds, $t$ la taille de population, $R$ le nombre de générations, $N1$ le nombre d'itérations dans la boucle interne, $N2$ le nombre d'itérations dans la boucle externe.

Le GA crée $R$ sous-populations, chacune avec $t$ individus, pour cela nous partitionnons le GPU à $R$ blocs, chacun de $t$ threads. Le thread maître de chaque bloc est le thread 0 et le thread maître global est le thread 0 du bloc 0.
Le bloc $i$, $0 \leq i < R$ stocke dans sa mémoire partagée les données nécessaires pour exécuter GA, à partir d'un individu ancêtre $P_0$ (solution initiale) générée comme indiqué précédemment par le CPU. Au début, $P_0$ est dupliqué dans tous les blocs $i$, ce qui donne $P_i = P_0$ et est mis à jour après chaque itération de la boucle interne.

Notons $T_0^i, ..., T_{t-1}^i$ les threads du bloc $i$. À partir de $P_i$, chaque thread $T_j^i$ génère une nouvelle solution (individu) $p_j^i$ en appliquant une permutation aléatoire à $P_i$ ($p_j^i$ = 1-$exchange(P_i)$). Alors $p_0^i, ..., p_{t-1}^i$ est la population initiale du GA exécutée par le bloc $i$. Notons que, la taille de la population $t$ est la même pour tous les blocs.

Le bloc $i$ exécute GA sur cette sous-population. Chaque thread $T_{2j}^i$ génère deux enfants, $ch_1$ et $ch_2$ en croisant les parents $p_{2j}^i$, $p_{2j+1}^i$, puis $T_{2j}^i$ corrige $ch_1$ et $T_{2j+1}^i$ corrige $ch_2$ produisant des individus réalisables $c_{2j}^i$, $c_{2j+1}^i$. Ensuite, la mutation est appliquée à 2% de la nouvelle sous-population composée du $c_j^i$, $0 \leq j < t$ et chaque individuel $c_j^i$ est évalué par le thread $T_j^i$, pour obtenir $f_j^i = fitness(c_j^i)$.

Tous les $c_j^i$ et $f_j^i$ sont stockés dans la mémoire partagée du bloc $i$. Par conséquent, le thread maitre du bloc $i$ met à jour l'ancêtre, pour la prochaine itération, avec l'in-

dividu $c^i_{j*}$ avec $min(min_i\{f^i_j\}, f_i)$, où $f_i$ est le coût de $P_i$ (la solution initiale) et met à jour l'ancêtre $P_i$ en tant que $c^i_{j*}$ pour la prochaine itération. La boucle interne de GA se termine après $N1$ itérations (la même pour tous les blocs).

Les $c^i_{j*}$, $0 \leq i < R$, sont copiés dans la mémoire globale et l'individu $c^{i*}_{j*}$ avec le $min_i\{f^i_{j*}\}$ est sélectionné comme solution finale ou comme nouvelle solution initiale pour la prochaine itération de la boucle externe. Le processus est répété $N2$ fois, ce qui donne $N1 * N2$ générations.

La figure 4.2 donne le schéma de l'implémentation du GA parallèle sur GPU.

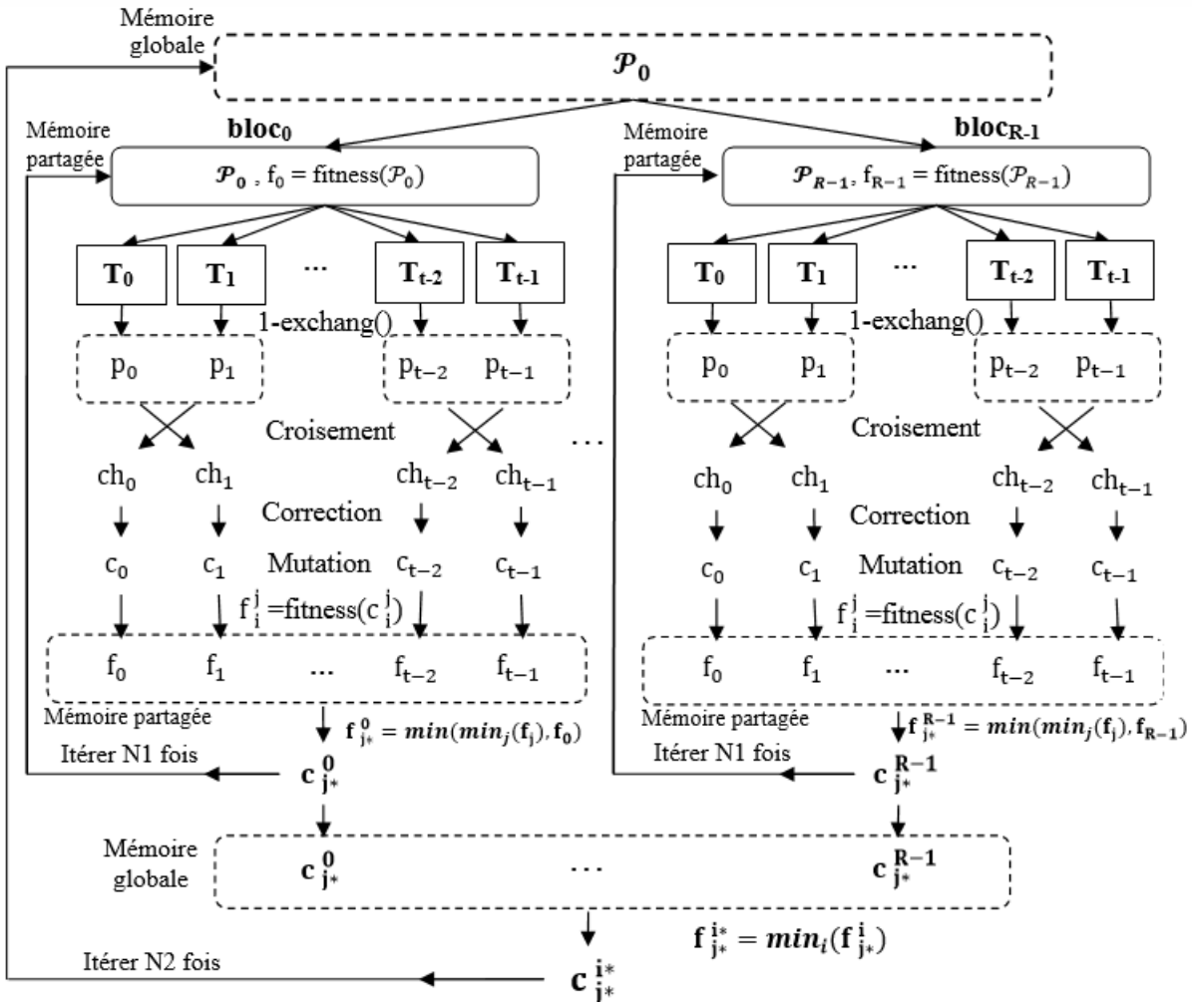



FIGURE 4.2 – GA parallèle pour l'USApHMP.

Le pseudo code CUDA exécuté par le CPU est le suivant :

1. Copier la matrice de distance $(d_{ij})$ et la matrice de flux $(w_{ij})$ sur la mémoire globale du GPU,
2. Générer $(P_0)$ // Générer la solution initiale $P_0$,
3. Copier $P_0$ sur la mémoire globale du GPU,
4. Définir les blocs et la grille comme suit :
   $dim3\ dimBlock(t, 1)$ ; $dim3\ dimGrid(R, 1)$ ;
5. Lancer le kernel GA $(P_0)$ :
   $GA <<< dimGrid, dimBlock >>> (P_0)$ ;
6. Lire la solution à partir de la mémoire globale.

### 4.3.3 Résultats expérimentaux

Nous avons utilisé une carte Nvidia Quadro avec 2 Go et 384 cœurs pour tester l'implémentation proposée sur cinq types d'instances : CAB, AP, PlanetLab et Urand et sur les larges instances générées par nous-même :

- L'ensemble d'instances CAB est un ensemble d'instances introduites par O'Kelly M. E. [109] et basées sur le flux de passagers des compagnies aériennes entre 25 villes américaines. Il contient des distances (qui satisfont l'inégalité triangulaire) et une matrice de flux symétrique entre les villes. Les instances sont de 10, 15, 20 et 25 nœuds. Les facteurs de distribution et de collecte $\delta$ et $\chi$ sont égaux à 1, tandis que le facteur de transfert (facteur d'économie) $\alpha$ prend les valeurs 0,2, 0,4, 0,6, 0,8 ou 1 et le coût fixe pour la création des hubs égale à 10, 150, 200 ou 250.
- L'ensemble d'instances AP (Australian Post) est un ensemble d'instances réel représentant les flux des courriers en Australie. Les facteurs de distribution et de collecte $\delta$ et $\chi$ sont respectivement égaux à 3 et 2, tandis que le facteur de transfert $\alpha$ est égal à 0,75 pour toutes les instances. Les flux des courriers ne sont pas symétriques et il existe des flux possibles entre chaque nœud et lui-même (c-à-d les flux de district intra-code possibles dans les flux de courrier). Le coût fixe pour la mise en place d'un hub est différent pour chaque hub et pour chaque problème, il existe deux types des coûts fixes, les coûts serrés (Tight) et les coûts larges (loose). Les instances à coûts serrés sont plus difficiles à résoudre, car les nœuds avec des flux totaux plus importants sont définis avec des coûts fixes plus élevés.
- L'ensemble d'instances Urand est constitué d'instances aléatoires jusqu'à 400 nœuds générés par Meyer et al. [99], tandis que l'instance à 1000 nœuds a été générée par Ilic et al. [81]. Dans ces tests, les coordonnées des nœuds ont été

générées de manière aléatoire de 0 à 100 000 et la matrice de flux a été générée aussi de manière aléatoire.

- L'ensemble d'instances PlanetLab sont des instances de retard entre nœuds (délai nœud à nœud) pour effectuer des mesures sur Internet (Ilic et al. [81]). Dans ces réseaux, $\chi = \alpha = \delta = 1$ et la matrice de distance ne respecte pas l'inégalité triangulaire ; le flux entre les nœuds est égal à 0 si $i = j$ et 1 sinon.
- L'ensemble d'instances larges sont des instances plus importantes que nous avons généré en utilisant la même procédure d'instanciation utilisée pour les instances Urand (Meyer et al [99]). Ces nouvelles instances complexes comprennent jusqu'à 6000 nœuds.

L'implémentation GPU proposée utilise les mémoires partagées (plutôt que la mémoire globale) pour stocker les populations et cela pour réduire le temps d'accès aux mémoires. Cependant, le transfert de temps entre le CPU et le GPU varie en fonction du nombre de nœuds. Dans le reste, les temps donnés incluent (calcul de la solution initiale, transferts de données entre le CPU et le GPU et l'exécution du GA sur GPU).

Les notations suivantes sont utilisées dans les tableaux dans le reste du manuscrit :

- n : le nombre de nœuds.
- p : le nombre de hubs.
- Sol CPU : la meilleure solution si elle est connue ; sinon "-" est écrite.
- Sol GPU : la solution obtenue par l'implémentation GPU.
- Temps CPU : le meilleur temps (en seconde) dans la littérature.
- Temps GPU : le temps (en seconde) de l'implémentation GPU.

Notre implémentation atteint rapidement les solutions optimales ou les meilleures solutions pour tous les tests.

Pour les instances CAB, nous avons obtenu les solutions optimales dans toutes les instances dans un temps $< 1s$. Comme toutes les instances sont résolues à l'optimalité dans les travaux précédents (Silva et Cunha [124]) et (Abyazi et Ghanbari [5]). Nous ne reporterons pas les résultats du CAB pour l'USApHMP (ce n'est pas un chalenge pour nous), nous ne signalerons que les temps de calcul du GPU.

Les tableaux. 4.1 - 4.4 comparent nos résultats aux meilleurs résultats obtenus dans la littérature sur tous les benchmarks connus pour résoudre l'USApHMP. Comme le montre le tableau. 4.1, pour les instances AP, nous avons obtenu des

solutions optimales pour 100 et 200 nœuds en très peu de temps (pour les nœuds jusqu'à 50, nous avons obtenu les solutions optimales dans un temps $< 0,3s$). A notre connaissance, les résultats des instances AP pour l'USApHMP de 300 et 400 nœuds n'ont pas encore été résolu dans la littérature. Trouver des solutions exactes à des instances volumineuses à l'aide du solveur standard (CPLEX) est un véritable défi et nécessite beaucoup de temps.

TABLEAU 4.1 – Résultats des instances AP.

| n | p | Sol CPU | Sol GPU | Temps GPU | n | p | Sol CPU | Sol GPU | Temps GPU |
|---|---|---|---|---|---|---|---|---|---|
| 100 | 5 | 136929.44 | 136929.44 | 1.31 | 300 | 5 | - | **174914.73** | 5.63 |
| | 10 | 106469.56 | 106469.56 | 1.31 | | 10 | - | **134773.55** | 5.71 |
| | 15 | 90533.52 | 90533.52 | 1.49 | | 15 | - | **114969.85** | 5.89 |
| | 20 | 80270.96 | 80270.96 | 1.63 | | 20 | - | **103746.44** | 5.87 |
| 200 | 5 | 140062.64 | 140062.64 | 3.60 | 400 | 5 | - | **176357.92** | 6.74 |
| | 10 | 110147.65 | 110147.65 | 3.72 | | 10 | - | **136378.19** | 6.84 |
| | 15 | 94459.20 | 94459.20 | 3.78 | | 15 | - | **117347.10** | 7.10 |
| | 20 | 84955.32 | 84955.32 | 3.84 | | 20 | - | **104668.27** | 7.42 |

Nous reportons nos résultats pour les instances PlanetLab dans le tableau. 4.2. Nous pouvons constater que notre approche surpasse les autres résultats (par exemple, Ilic et al. [81]), à la fois en termes de coût et de temps de calcul. Ilic et al. [81] reportent des résultats pour $p \approx \sqrt{n}$. Nous présentons aussi les résultats pour d'autres valeurs de $p$ (voir benaini et al. [22]).

TABLEAU 4.2 – Résultats des instances PlanetLab.

| Instance | n | p | Sol CPU | Sol GPU | Temps CPU | Temps GPU |
|---|---|---|---|---|---|---|
| 01-2005 | 127 | 12 | 2927946 | **2904434** | 148.9 | **0.5** |
| 02-2005 | 321 | 19 | 18579238 | **18329984** | 462.7 | **6.9** |
| 03-2005 | 324 | 18 | 20569390 | **20284132** | 543.8 | **7.5** |
| 04-2005 | 70 | 9 | 739954 | **730810** | 0.6 | **0.3** |
| 05-2005 | 374 | 20 | 25696352 | **25583240** | 622.6 | **8.3** |
| 06-2005 | 365 | 20 | 22214156 | **22191592** | 581.7 | **7.9** |
| 07-2005 | 380 | 20 | 30984986 | **30782956** | 546.6 | **8.5** |
| 08-2005 | 402 | 21 | 30878576 | **30636170** | 637.6 | **8.7** |
| 09-2005 | 419 | 21 | 32959078 | **32649752** | 684.9 | **9.3** |
| 10-2005 | 414 | 21 | 32836162 | **32687796** | 731.9 | **9.1** |
| 11-2005 | 407 | 21 | 27787880 | **27644374** | 588.3 | **9.2** |
| 12-2005 | 414 | 21 | 28462348 | **28213748** | 680.3 | **9.1** |

L'implémentation GPU offre les meilleures solutions pour les instances Urand comportant jusqu'à 400 nœuds et surpasse celles de Ilic et al. [81] pour les instances de 1000 nœuds. Les temps d'exécution de notre approche est très rapide et est $\leq 18s$ pour toutes les instances, alors que le meilleur temps connu est de plus de 7 minutes. De plus, il augmente rapidement avec $p$ dans Ilic et al. [81] alors qu'il ne change pas beaucoup pour l'implémentation sur GPU. Clairement, nos solutions sont bien meilleures en terme de coût que celles de Ilic et al. [81], comme on peut le voir dans le tableau. 4.3 $Temps\ CPU \approx 20 * Temps\ GPU$.

Tableau 4.3 – Résultats des instances Urand larges.

| n | p | Sol CPU | Sol GPU | Temps CPU | Temps GPU |
|---|---|---|---|---|---|
| 1000 | 2 | 198071412.53 | **8184986.50** | 1.7245 | **9.321** |
| | 3 | 169450816.35 | **7024184.00** | 8.1550 | **9.785** |
| | 4 | 150733606.87 | **6184749.01** | 2.2240 | **10.431** |
| | 5 | 142450250.26 | **5860994.06** | 58.6070 | **10.89** |
| | 10 | 114220373.07 | **4752317.00** | 187.8385 | **13.7** |
| | 15 | - | **4228256.88** | - | **15.23** |
| | 20 | 198071412.53 | **3928617.48** | 403.4280 | **17.923** |

Nous reportons dans le tableau. 4.4 les résultats des larges instances que nous avons générées jusqu'à 6000 nœuds.

Tableau 4.4 – Résultats des instances Urand très larges.

| n | p | Sol GPU | Temps GPU | n | p | Sol GPU | Temps GPU |
|---|---|---|---|---|---|---|---|
| **1500** | 20 | 454787506 | 196 | **4000** | 20 | 3234999192 | 3076 |
| | 30 | 407155164 | 286 | | 30 | 2983891783 | 3276 |
| | 40 | 380114045 | 423 | | 40 | 2769550514 | 3365 |
| | 50 | 363586538 | 574 | | 50 | 2644606684 | 3648 |
| **2000** | 20 | 805749722 | 477 | **5000** | 20 | 5085803132 | 4662 |
| | 30 | 733375448 | 580 | | 30 | 4656787498 | 4720 |
| | 40 | 686515363 | 714 | | 40 | 4353561395 | 4996 |
| | 50 | 655938000 | 965 | | 50 | 4143849388 | 5112 |
| **3000** | 20 | 1804950952 | 1157 | **6000** | 20 | 7398401957 | 5614 |
| | 30 | 1642145354 | 1544 | | 30 | 6675723961 | 5748 |
| | 40 | 1538548764 | 1869 | | 40 | 6293053841 | 5964 |
| | 50 | 1468780124 | 2086 | | 50 | 5999780197 | 6212 |

### 4.3.4 Résolution de l'USAHLP

Pour résoudre l'USAHLP, on résout l'USApHMP pour $p = 1, ..., n$ en parallèle.

#### Solutions initiales

Nous générons en parallèle $n$ solutions initiales, chacune localisant $p$ hubs (où $p$ varie de 1 à $n$). Le pseudo code CUDA exécuté par le CPU pour générer ces $n$ solutions initiales est le suivant :

1. Copier la matrice de distance et la matrice de flux sur la mémoire globale du GPU.
2. Définir les blocs et la grille comme suit :
   $dim3\ dimBlock(1, 1)$ ; $dim3\ dimGrid(n, 1)$ ;
3. Lancer le kernel :
   $GenerateInitialSolutions <<< dimGrid, dimBlock >>> ()$ ;

Dans notre implémentation, $GenerateInitialSolutions()$ est exécuté sur $n$ blocs ; le bloc $p$ génère la solution initiale avec $p$ hubs. On obtient donc parallèlement $n$ solutions initiales $P_i$ avec $i$ hubs $1 \leq i \leq n$.

#### Implémentation du GA pour l'USAHLP

Comme expliqué précédemment, les $R$ blocs créent $R$ sous-populations comportant chacune $t$ individus pour trouver la solution optimale ou la meilleure solution pour l'USApHMP. Par conséquent, $R * n$ blocs sont nécessaires pour calculer les $n$ solutions ($p$ variant de 1 à $n$) pour l'USAHLP. Par conséquent, le pseudo-code CUDA pour l'USAHLP est le suivant :

1. Générer en parallèle $n$ solutions initiales, $P_1, ..., P_n$.
2. Définir les blocs et la grille comme suit :
   $dim3\ dimBlock(t, 1)$ ; $dim3\ dimGrid(R * n, 1)$ ;
3. Lancer en parallèle le kernel GA avec les solutions initiales $P_1, ..., P_n$, on résout l'USApHMP pour $p = 1, ..., n$.
   $GA <<< dimGrid, dimBlock >>> (P_1, ..., P_n)$ ;
4. La solution de l'USAHLP est la solution avec le coût minimal, parmi les $n$ solutions finales ;

#### Résultats expérimentaux

Pour les 80 instances CAB, nous avons obtenu les solutions optimales en temps de calcul $< 0.8s$. En particulier, nous confirmons la valeur de $1081,05$ trouvée par Abyazi et Ghanbari [5] pour $n = 10$, $\alpha = 1$, $F_K = 150$, dont la valeur est incorrecte

dans certains travaux de la littérature, comme dans Silva et Cunha [124].

Les résultats pour les instances AP sont reportés dans le tableau. 4.5. Nous avons obtenu des solutions optimales jusqu'à 200 nœuds, pour les deux types de coûts fixes d'installation des hubs (Tight et Loose). L'implémentation proposée dépasse largement les solutions les plus connues de Silva et Cunha [124] pour les instances de 200, 300 et 400 nœuds et fournit des solutions pour des instances non résolues.
Ces résultats montrent que, même si nous localisons les mêmes hubs, mais avec des affectations différentes, on obtient des solutions avec des coûts différents (c'est le cas des instances 300T et 400T).

Tableau 4.5 – Résultats des instances AP pour l'USAHLP.

| n | Type FK | Sol CPU | Hubs de la Sol CPU | Sol GPU | Hubs de la Sol GPU | Temps GPU |
|---|---|---|---|---|---|---|
| 10 | Loose | 224250.05 | 3, 4, 7 | 224250.05 | 3, 4, 7 | 0.024 |
| 20 | Loose | 234690.95 | 7, 14 | 234690.95 | 7, 14 | 0.061 |
| 25 | Loose | 236650.62 | 8, 18 | 236650.62 | 8, 18 | 0.069 |
| 40 | Loose | 240986.23 | 14, 28 | 240986.23 | 14, 28 | 0.243 |
| 50 | Loose | 237421.98 | 15, 36 | 237421.98 | 15, 36 | 0.490 |
| 100 | Loose | 238016.28 | 29, 73 | 238016.28 | 29, 73 | 1.624 |
| 200 | Loose | 233803.02 | - | **233,801.35** | **43, 148** | 3.316 |
| 300 | Loose | 263913.15 | 26, 79, 170, 251, 287 | **258823.42** | **79, 126, 170, 192, 287** | 8.183 |
| 400 | Loose | 267873.65 | 99, 180, 303, 336 | **259416.31** | **146, 303, 336** | 13.32 |
| 10 | Tight | 263399.94 | 4, 5, 10 | 263399.94 | 4, 5, 10 | 0.026 |
| 20 | Tight | 271128.18 | 7, 19 | 271128.18 | 7, 19 | 0.072 |
| 25 | Tight | 295667.84 | 13 | 295667.84 | 13 | 0.085 |
| 40 | Tight | 293164.83 | 19 | 293164.83 | 19 | 0.297 |
| 50 | Tight | 300420.98 | 24 | 300420.96 | 24 | 0.548 |
| 100 | Tight | 305097.96 | 52 | 305097.93 | 52 | 1.864 |
| 200 | Tight | 272237.78 | - | **272,188.10** | **54, 122** | 4.961 |
| 300 | Tight | 276023.35 | 30, 154, 190 | **266030.76** | **30, 154, 190** | 9.742 |
| 400 | Tight | 284037.25 | 101, 179, 372 | **275769.09** | **101, 179, 372** | 15.98 |

Le tableau. 4.6 présente les résultats des instances Urand non résolues, générées par Ilic et al. [81], qui ont résolu ces instances que pour l'USApHMP où le nombre de hubs à localiser est donné. Ici, nous donnons les premiers résultats pour l'USAHLP.

TABLEAU 4.6 – Résultats des instances Urand pour l'USAHLP.

| n | Sol GPU | Nombre de hubs | Temps GPU |
|---|---|---|---|
| 100 | 43072.82 | 17 | 5.78 |
| 200 | 160799.86 | 27 | 9.45 |
| 300 | 343803.97 | 29 | 15.78 |
| 400 | 598628.03 | 34 | 21.65 |

Dans le tableau. 4.7, nous comparons les temps d'exécution de notre implémentation parallèle avec le temps d'exécution de Silva et Cunha [124] pour les instances AP de 100 à 400 nœuds et pour les deux types de coûts fixes d'installation des hubs (Tight et Loose). Nos résultats sont meilleurs en qualités de solutions et en temps d'exécutions.

TABLEAU 4.7 – Comparaison entre le temps d'exécutions du Silva [124] et du GPU.

| n | Type Fk | Temps Silva | Temps GPU |
|---|---|---|---|
| 100 | Loose | 11.99 | 1.62 |
| 200 | Loose | 65.19 | 3.31 |
| 300 | Loose | 41.15 | 8.18 |
| 400 | Loose | 106.33 | 13.4 |
| 100 | Tight | 19.21 | 1.86 |
| 200 | Tight | 79.51 | 4.96 |
| 300 | Tight | 48.65 | 9.74 |
| 400 | Tight | 115.86 | 15.98 |

## 4.4 Résolution des problèmes CSApHMP et CSAHLP

Notre objectif dans cette section est d'adapter le présent GA au HLP où les hubs ont une capacité limitée. Le GA part de différentes populations initiales et les améliore via des opérateurs génétiques classiques adaptés à ce problème. Une population initiale est générée à l'aide d'un nouveau critère qui identifie les nœuds susceptibles d'être des hubs dans une solution optimale. L'implémentation sur GPU développé surpasse, à la fois en termes de temps et en qualité de solution, les approches classiques proposées dans la littérature pour ces problèmes. Nous l'avons comparé aux meilleures solutions récentes sur des benchmarks jusqu'à 400 nœuds et nous avons résolu de larges instances jusqu'à 1500 nœuds. À notre connaissance, c'est la première implémentation sur GPU pour ces problèmes.

Dans ce qui suit nous présentons un exemple simple du CSApHMP. Nous discutons un nouveau critère qui identifie les nœuds susceptibles d'être des hubs dans une solution optimale. Nous présentons la description du GA et son implémentation sur GPU. Nous présentons les résultats expérimentaux pour le CSApHMP. Enfin, nous présentons l'approche de résolution du CSAHLP.

### 4.4.1 Exemple Simple de CSApHMP

Le CSApHMP peut être modélisé exactement comme l'USApHMP qui est déjà présenté dans le premier chapitre sections 1.5.1, à cela en ajoute la contrainte qui permet de limiter la capacité du flux entre les nœuds (Ernst et Krishnamoorthy [58]).

Le CSAHLP est modélisé exactement comme l'USAHLP, où le nombre de hubs et leur localisation sont des décisions à prendre. Ceci est formulé en ajoutant un terme supplémentaire à la fonction objectif, visant à minimiser le coût de transport dans le réseau et aussi à minimiser le coût fixe de l'installation des hubs. Pour CSAHLP par rapport à l'USAHLP en ajoutant la contrainte qui permet de limiter la capacité du flux entre les nœuds (voir Stanimirovic Z. [128]).

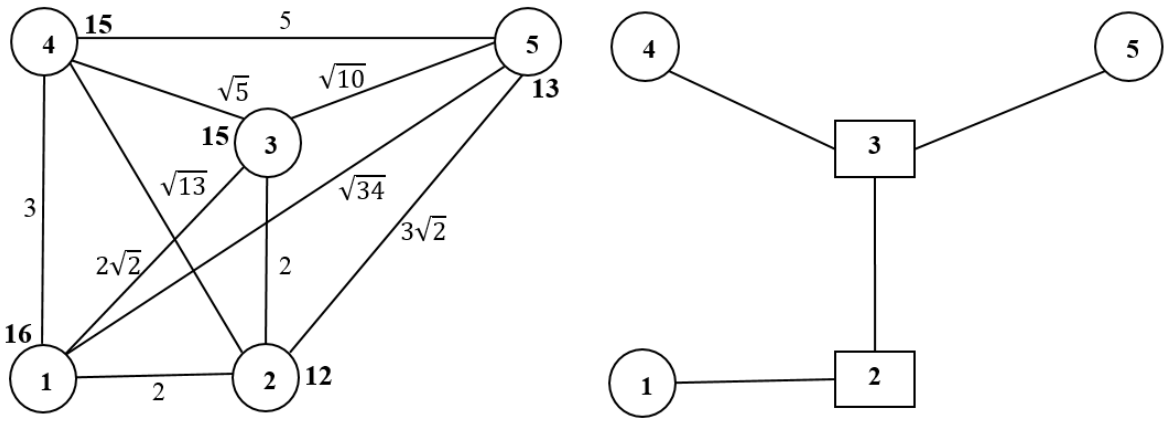
La contrainte de capacité pour limiter les flux entre les nœuds est :

$$\sum_{i \in N} O_i Z_{ik} \leq G_k Z_{kk} \qquad i, k \in N$$

avec $G_k$ est la capacité du hub k.

**L'exemple de CSApHMP :** La figure 4.3 (a) montre un exemple de CSApHMP présenté dans Stanimirović Z. [128] avec $n = 5$ nœuds présentés par leurs distances (nombres sur les arcs) et leurs capacité (en gras).

Le nombre de hubs à localiser est $p = 2$ et les paramètres $\chi = \delta = 1$ et $\alpha = 0,25$. La quantité de flux $w_{ij}$ d'un nœud $i$ à un nœud $j$ est égale à 1, même pour $i = j$. Donc, $O_i = D_i = 5$ pour tout $i$. La figure 4.3 (b) présente la solution optimale pour cet exemple. Les hubs sont les nœuds 2 et 3. Les nœuds 1 et 2 sont affectés au hub 2, tandis que les nœuds 3, 4 et 5 sont affectés au hub 3 (chaque hub est affecté à lui-même). Ceci implique que $z_{22} = z_{33} = z_{12} = z_{43} = z_{53} = 1$ et que $y^i_{kl} * d_{kl}$ est non nul pour $k, l = 2, 3$ et $k, l = 3, 2$. D'où le coût de transport global de cette solution est $d_{12}(\chi O_1 + \delta D_1) + d_{43}(\chi O_4 + \delta D_4) + d_{53}(\chi O_5 + \delta D_5) + \alpha d_{23}(w_{13} + w_{14} + w_{15} + w_{23} + w_{24} + w_{25} + w_{32} + w_{31} + w_{42} + w_{41} + w_{52} + w_{51}) = 79,983$. Les contraintes de capacité sont satisfaites puisque $O_1 + O_2 \leq G_2$ et $O_3 + O_4 + O_5 \leq G_3$.




(a) Problème de réseau des hubs (b) La solution optimale

FIGURE 4.3 – Un exemple du CSApHMP avec $n = 5$ et $p = 2$.

### 4.4.2 Choix des hubs initiaux

Quels critères peuvent être utilisés pour identifier les nœuds susceptibles d'être des hubs dans une solution optimale ?
Peker et al. [115] ont réalisé une étude très intéressante sur ce sujet, pour l'USAHMP qui nous a fortement inspiré dans cette étude. Ils ont observé à partir des solutions optimales sur des benchmarks bien connus, que les emplacements optimaux des hubs sont fortement influencés à la fois par l'ampleur des demandes $(O_i + D_i)$ aux nœuds et la distribution spatiale des nœuds (Centralité intermédiaire (betweenness)). La centralité du nœud $i$ est mesurée par $c_i = \sum_j d_{ij}$ donc les nœuds centraux ont les valeurs les plus petites de $c_i$. La centralité intermédiaire pour le nœud $k$ est la demande totale qui utiliserait $k$ comme hub intermédiaire sur le chemin $i - k - j$ et est mesuré par $B_k = \sum_{i,j:k=argmin\ (d_{im}+d_{mj})} w_{ij}$. Leur algorithme CBS partitionne les nœuds en clusters, chacun est centré sur son nœud important. Ces nœuds centraux sont susceptibles d'être les hubs de la solution optimale. Conformément à Peker et al. [115], les mesures d'importance des nœuds qui trouvent l'optimal dans la plupart des instances de CAB et AP sont : $max_i(O_i + D_i)c_i$, $max_i \sum_j d_{ij}(w_{ij} + w_{ji})$ et $max_i\ a * \dfrac{(O_i + D_i)}{\sum_j (O_j + D_j)} + b * \dfrac{c_i}{\sum_j c_j}$, avec $a = b = 0,5$ ou $a = 0,75$ et $b = 0,25$. Les mesures avec la centralité intermédiaire fournissent des performances médiocres et l'importance des nœuds $max_i(O_i + D_i)c_i$ atteint rapidement l'optimal.

Pour le CSAHLP, Sasaki et Fukushima [122] ont utilisé les valeurs $h(k) = a * \dfrac{F_k}{G_k} + \sum_{i,j}(d_{ik}w_{ik} + d_{kj}w_{kj})$, où $a$ est un paramètre $< 1$, $F_k/G_k$ représente le

coût fixe par unité de capacité et $\sum_j (d_{ik}w_{ik} + d_{kj}w_{kj})$ représente le coût total traverser via le hub $k$. Un candidat hub avec petit $h(k)$ est susceptible d'être choisi comme hub dans une solution optimale.

Dans cette étude, nous introduisons des critères similaires prenants en compte les capacités, l'ampleur des demandes et les centralités. Donc, nous utilisons les valeurs :

$$h(k) = \frac{1}{G_k} \sum_j (d_{kj}w_{kj}).$$

Comme critère de partitionnement de l'ensemble des nœuds en $p$ groupes. Chaque groupe pouvant contenir un hub de la solution optimale. Un nœud $k$ de grande capacité qui minimise le coût total sortant via $k$ est susceptible d'être un hub de la solution optimale. Ainsi, le nœud $k$ avec le plus petit $h(k)$ est le plus susceptible de vérifier ce critère.

Nous devons choisir $p$ hubs initiaux pour le CSApHMP. Pour ce faire, nous trions les $n$ nœuds en ordre croissant selon leur $h(k)$. Les nœuds ainsi triés sont partitionnés en $p$ groupes chacun de $r = \left\lceil \frac{n}{p} \right\rceil$ nœuds. Le $groupe_1$ composé des $r$ premiers nœuds, le $groupe_2$ des $r$ nœuds suivants, etc. Par exemple, la liste $h()$ de l'exemple de la figure 4.3 (a) est (0.85, 0.98, 0.68, 0.92, 1.40). Ainsi, la liste ordonnée des nœuds selon $h()$ est (3, 1, 4, 2, 5) qui est partitionnée en $groupe_1$ = (3, 1, 4) et $groupe_2$ = (2, 5). Donc, si la solution initiale localise les nœuds avec le plus petit $h(k)$ dans leurs groupes, alors ces hubs seront les nœuds 2 et 3 qui sont effectivement les hubs de la solution optimale obtenue par Stanimirović Z. [128].

Plus généralement, ayant les $p$ groupes, nous avons adopté la méthodologie de Peker et al. [115] pour sélectionner un seul hub dans chaque groupe. Nous avons testé, sur les instances AP et Urand, différents critères de sélections parmi lesquels : le nœud $k$ avec le plus petit $h(k)$ dans son groupe, le nœud $k$ avec le plus grand $(O_k + D_k)$ dans son groupe et autres mesures d'importance des nœuds présentés dans Peker et al. [115]. Ces tests montrent que le choix du nœud avec le plus grand $(O_k + D_k)$ dans chaque groupe (le nœud ayant des positions spatiales où le trafic est dense) est le meilleur critère pour ces tests. Près de la moitié des hubs initiaux sélectionnés restent dans la solution optimale ou la meilleure solution et les hubs de la solution optimale / meilleure sont distribués dans environ 75% dans les groupes produits par $h()$. Cela signifie que les solutions initiales qui localisent les hubs ayant les plus grands $(O_k + D_k)$ dans leurs groupes sont assez proches des meilleures / optimaux hubs.

Nous illustrons ce propos sur l'instance AP 25.4l à 25 nœuds et $p = 4$ représenté sur la figure 4.4 . Les 4 groupes obtenus par le critère $h()$ sont représentés par des étoiles pour les nœuds du $groupe_1$, des carrés pour les nœuds du $groupe_2$, des cercles pour les nœuds du $groupe_3$ et par des triangles pour les nœuds du $groupe_4$. Les nœuds avec $max_k(O_k + D_k)$ dans leurs groupes sont les quatre nœuds mentionnés dans cette figure et sont sélectionnés comme hubs initiaux. Une solution optimale pour cette instance localise les hubs représentés par des objets remplis dans la figure 4.4 et affecte chaque non-hub à son hub le plus proche. Par conséquent, trois hubs initiaux restent des hubs de la solution optimale et chacun des quatre groupes contient un hub dans cette solution optimale, ce qui signifie que le découpage en $p$ groupes donne exactement la distribution des hubs optimaux / meilleurs.

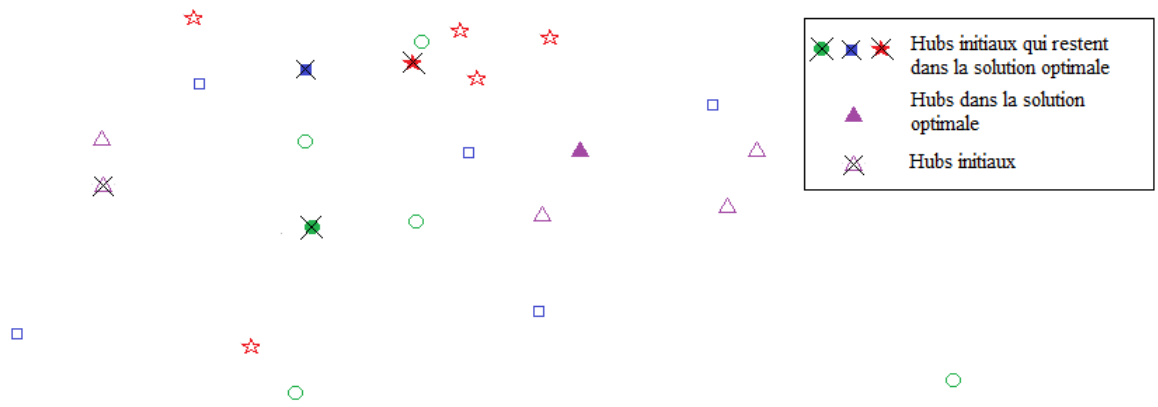



FIGURE 4.4 – Comparaison entre les emplacements des hubs initiaux et les optimaux pour l'instance AP 25.4L.

De plus, notre critère peut être naturellement étendu au cas de CSAHLP avec un coût fixe en prenant :

$$h(k) = \frac{F_k}{G_k} \sum_j (d_{kj} w_{kj})$$

Nous n'utilisons pas ce critère pour résoudre le CSAHLP, puisque la solution du CSAHLP dans notre implémentation est celle avec un coût minimum parmi les solutions du CSApHMP, pour $p = 1, ..., n$ (on profite de la puissance de calcul du GPU !). Ainsi, nous résolvons en parallèle les CSApHMPs pour tous les $p$ pour lesquels le problème a une solution.

### 4.4.3 Algorithme génétique pour CSApHMP

Nous avons gardé le même principe, notre GA part de différentes populations initiales pour résoudre le CSApHMP sur GPU. Dans ce qui suit, nous allons expliquer l'adaptation effectuée au codage des chromosomes, à la solution initiale et aux opérateurs génétiques.

#### Prétraitement

L'approche nécessite des calculs préliminaires qui sont effectués une fois au début du GA et qui seront utilisés tout au long de l'implémentation.

- La liste ordonnée $h()$ est calculée une fois au début du GA et triée par ordre croissant. Elle sera utilisée tout au long du GA pour localiser les hubs initiaux en tant que hubs dans une solution optimale.
- Pour chaque nœud $i$, la liste ordonnée $L_i$ qui contient les nœuds triés selon leurs distances à $i$ c-à-d $L_i(k) = k^{\text{ième}}$ nœud le plus proche à $i$, $1 \leq k \leq n$. La liste $L_i$ est utilisée pour trouver plus rapidement le hub le plus proche du nœud $i$, et éventuellement pour affecter $i$ à son hub. Ainsi, $L_i$, $1 \leq i \leq n$, sera calculé une seule fois au début du GA et utilisé tout au long de l'implémentation pour affecter des non-hubs à leurs hubs les plus proches qui satisfont la contrainte de capacité.

#### Codage

Nous adoptons un codage simple par un tableau d'entier $s$ qui représente l'affectation de nœuds aux hubs, c-à-d que $s(i)$ est le hub auquel le nœud $i$ est affecté. De plus, chaque hub $k$ est affecté à lui-même $s(k) = k$.

#### Solution initiale

Une solution initiale appropriée sera calculée (plutôt qu'une solution aléatoire) pour atteindre rapidement une solution quasi optimale. Cette solution initiale localise les $p$ hubs initiaux en utilisant le critère $h()$ et affecter chaque non-hub $i$ à son hub le plus proche $k$ qui satisfait la contrainte de capacité, à savoir $\sum_{r=1}^{t} O_{i_r} \leq G_k$ où $i_1, ..., i_t$ sont les nœuds affectés au hub $k$. Sinon, le nœud $i$ est affecté au premier hub suivant dans $L_i$ qui satisfait la contrainte de capacité.
Notons que, si la somme des flux entrants pour tous les nœuds est supérieure à la somme des $p$ plus grandes capacités, le problème n'a pas de solution.

#### Les opérateurs génétiques

Les opérateurs suivants sont utilisés dans notre parallèle GA :

- *Générer un nouvel individu à partir d'un autre individu par 1-exchange()* : 1-*exchange*($s$) génère un nouvel individu à partir de l'individu $s$ en échangeant aléatoirement un hub $k$ avec un nœud non-hub $l$ qui vérifie $O_l \leq G_l$ et affecter (à l'aide de $L_i$) chaque nœud $i$ non-hub à son hub le plus proche qui satisfait la contrainte de capacité. Notons que, si $O_l > G_l$, le nœud $l$ ne peut pas être hub.

- *Croisement* : Nous utilisons un croisement en un seul point aléatoire pour échanger les deux parties des parents $p_1$ et $p_2$, afin de générer deux descendants $ch_1$ et $ch_2$. Si $ch_1$ (resp. $ch_2$) n'est pas une solution réalisable, alors elle sera corrigée, ce qui donne deux individus réalisables $c_1$ (resp. $c_2$).

- *Mutation* : La mutation prend normalement la forme d'une modification mineure aléatoire du codage de la solution. Nous utilisons un opérateur de mutation qui transforme 10% de l'affectation des nœuds entre hubs pour 2% d'individus.

- *Correction des individus* : Les contraintes sur le nombre $p$ de hubs à localiser et la capacité des hubs peuvent être violées par les opérateurs de croisement et de mutation dans le GA (1-*exchange*() ne viole pas ces contraintes). Ainsi, une solution est corrigée si elle contient un nombre de hubs différent de $p$ ou si un débordement de hub se produit en raison de l'affectation de nœuds aux hubs. Dans notre approche, la violation de capacité est traitée comme suit.

  - Soit un hub $k$ et $i_1, i_2, ..., i_t$ les non hubs qui lui sont affectés ; suppose que $k$ ne satisfait pas à la contrainte de capacité, c-à-d $\sum_{r=1}^{t} O_{i_r} > G_k$. Dans ce cas, nous parcourons la liste $L_k$ et nous supprimons les derniers nœuds les plus éloignés $i_{t1}, ..., i_t$ de sorte que $\sum_{r=1}^{t1-1} O_{i_r} \leq G_k$ et affecter chaque nœud $i_{t1}, ..., i_t$ au hubs suivant dans $L_i$ qui satisfait la contrainte de capacité. Il est clair que le résultat dépend de l'ordre dans lequel les nœuds sont traités, mais les nœuds peuvent être ré-affectés en parallèle. Notons que, comme dans Naeem [104] et Kratica et al. [87], il est possible d'éviter de violer la capacité lors de l'étape d'affectation des nœuds aux hubs en affectant des nœuds aux hubs ayant une capacité suffisante. Dans ce cas, les nœuds doivent être affectés aux hubs de manière séquentielle, ce qui ne convient pas au calcul parallèle. Dans ce cas, le résultat dépend de l'ordre dans lequel les nœuds sont traités.

  - Si une solution a plus de $p$ hubs, alors on conserve les $p$ hubs avec les plus petits $h()$. Si le nombre de hubs est inférieur à $p$, les hubs manquants

sont choisis au hasard parmi les nœuds autres que les nœuds hub avec le plus petit $h()$. Enfin, les nœuds non hub sont affectés aux hubs les plus proches qui satisfont la contrainte de capacité.

### 4.4.4 Implémentation du GA pour CSApHMP

Nous décrivons l'implémentation GPU du GA et nous montrons son efficacité de notre implémentation sur plusieurs instances du CSApHMP et du CSAHLP de la littérature.

#### Stockage de données, prétraitement et solution initiale

Copier la matrice de distance $(d_{ij})$ et la matrice de flux $(w_{ij})$ dans la mémoire globale du GPU. Définir un bloc où ses threads calculent en parallèle les listes $L_i$. Le thread maître calcule la liste ordonnée $h()$ et les groupes $groupe_1, ..., group_p$ et sélectionne un hub de chaque groupe selon le critère $(O_k + D_k)$. Une fois les $p$ hubs connus par tous les threads, chaque thread $i$ alloue le nœud $i$ à son hub le plus proche en utilisant $L_i$. Cela constitue la solution initiale $P_0$ à partir de laquelle sont générées toutes les sous-populations initiales, traitées par GA. Une sous-population de $t$ individus est générée à partir de $P_0$ en appliquant $t$ fois 1-$exchange()$ à $P_0$.

#### Implémentation sur GPU du GA pour CSApHMP

Le GA génère $R$ sous-populations de taille $t$. Le nombre d'itérations dans la boucle interne (resp. boucle externe) est $N1$ (resp. $N2$). Donc, $N1 * N2$ est le nombre de générations et $R * t * N1 * N2$ est le nombre total d'individus évalués.

Par conséquent, on utilise une grille de $R$ blocs chacun de $t$ threads. La GA part de la solution initiale $P_0$ et la duplique $R$ fois dans les mémoires partagées des $R$ blocs. Chaque bloc $i$, $0 \leq i < R$, génère une sous-population de $t$ individus en appliquant $t$ fois 1-$exchange()$ à $P_0$ et stocke dans sa mémoire partagée les données nécessaires à l'exécution du GA avec $P_i$ comme solution initiale (initialement $P_i = P_0$).

Les threads $T_0^i, ..., T_{t-1}^i$ du bloc $i$ génèrent individuellement la sous-population initiale $p_0^i, ..., p_{t-1}^i$ en appliquant 1-$exchange()$ à $P_i$. Notons que la taille de la sous-population $t$ est la même pour tous les blocs et que $P_i$ est mis à jour après chaque itération.

Le bloc $i$ exécute GA sur cette sous-population. Chaque thread $T_{2j}^i$ génère deux enfants, $ch_1$ et $ch_2$ en croisant les parents $p_{2j}^i$, $p_{2j+1}^i$ puis $T_{2j}^i$ corrige $ch_1$ et $T_{2j+1}^i$

corrige $ch_2$ produisant des individus réalisables $c^i_{2j}$, $c^i_{2j+1}$. Ensuite, la mutation est appliquée à 2% de la nouvelle sous-population composée du $c^i_j$, $0 \leq j < t$ et chaque individuel $c^i_j$ est évalué par le thread $T^i_j$, pour obtenir $f^i_j = fitness(c^i_j)$.

Les $c^i_j$ et $f^i_j$ sont stockés dans la mémoire partagée du bloc $i$. Par conséquent, le thread maitre du bloc $i$ met à jour l'ancêtre, pour la prochaine itération, avec l'individu avec $min_i\{f^i_j\}$. La boucle interne de GA se termine après $N1$ itérations (la même pour tous les blocs).

Les $c^i_{j*}$, $0 \leq i < R$, sont copiés dans la mémoire globale et l'individu $c^{i*}_{j*}$ avec le $min_i\{f^i_{j*}\}$ est sélectionné comme solution finale ou comme nouvelle solution initiale pour la prochaine itération de la boucle externe. Le processus est répété $N2$ fois. Le pseudo code CUDA exécuté par le CPU est le suivant :

1. Copier la matrice de distance $(d_{ij})$ et la matrice de flux $(w_{ij})$ dans la mémoire globale du GPU.
2. Générer la solution initiale.
3. Définir les blocs et la grille comme suit :
   $dim3\ dimBlock(t, 1)$ ; $dim3\ dimGrid(R, 1)$ ;
4. Exécuter en parallèle le kernel GA :
   $GA <<< dimGrid, dimBlock >>> ()$ ;

### 4.4.5 Résultats expérimentaux

Deux types d'instances (les seuls pour ce problème à notre connaissance) et de larges instances générées par nous ont été utilisés pour évaluer cet implémentation.

- Les instances AP avec capacité, Ernst et al. [58]. Il existe deux types de capacités pour les hubs, capacité serrée (Tight) et capacité large (Loose).
- Les instances Urand est constitué aléatoires jusqu'à 400 nœuds générés par (Meyer et al., [99]).
- Les instances large entre 1000 et 1500 nœuds générés par nous-même. Dans ces instances, les coordonnées des nœuds, les capacités et la matrice de flux ont été générées de manière aléatoire.

Les tableaux. 4.8 - 4.11 donnent la taille de l'instance ($n$), le nombre de hubs ($p$), le coût de la meilleure solution connue dans la littérature (Sol CPU), le coût obtenu par notre implémentation GPU (Sol GPU), le meilleur temps de calcul dans la littérature (Temps CPU) en secondes et le temps d'exécution sur GPU (Temps GPU). La colonne C1 indique le nombre de hubs initial restant dans la solution optimale/meilleure et la colonne C2 indique le nombre de groupes contenant au moins un hub de la solution optimale/meilleure.

Nous n'avons pas raporté les résultats pour les instances AP jusqu'à 50 nœuds, nous n'avons pas les reportés ici, mais sont disponible dans [19]. Globalement, les temps exécutions pour ces instances vérifient $Temps\ GPU \approx \frac{Temps\ CPU}{10}$.

Le tableau. 4.8 montre les solutions optimales où les meilleures solutions obtenues pour les instances AP jusqu'à 200 nœuds dans un temps $< 0.7s$. Il reporte également les solutions pour les instances AP avec 100 et 200 nœuds dont ou connaissait pas les solutions.

Tableau 4.8 – Résultats des instances AP.

| n | p | Type Capacité | Sol CPU | Sol GPU | Temps CPU | Temps GPU | C1 | C2 |
|---|---|---|---|---|---|---|---|---|
| **100** | 10 | Loose | 107207.72 | 106469.57 | 29.53 | 0.307 | 5 | 8 |
| | 15 | Loose | 91285.26 | 90605.10 | 36.09 | 0.315 | 7 | 8 |
| | 20 | Loose | 81034.59 | 80682.71 | 43.29 | 0.338 | 9 | 11 |
| | 25 | Loose | - | **74311.28** | - | 0.357 | 11 | 15 |
| | 30 | Loose | - | **69124.76** | - | 0.367 | 14 | 17 |
| **100** | 10 | Tight | 109633.40 | 111088.33 | 32.11 | 0.306 | 4 | 7 |
| | 15 | Tight | 92635.94 | 92635.94 | 34.64 | 0.312 | 8 | 11 |
| | 20 | Tight | 82425.42 | 82425.42 | 42.90 | 0.347 | 9 | 13 |
| | 25 | Tight | - | **74784.58** | - | 0.361 | 14 | 17 |
| | 30 | Tight | - | **69031.34** | - | 0.365 | 17 | 19 |
| **200** | 10 | Loose | 110400.53 | 110400.53 | 182.99 | 0.661 | 6 | 7 |
| | 15 | Loose | 94525.39 | 94525.39 | 223.72 | 0.670 | 7 | 7 |
| | 20 | Loose | 85290.57 | 85290.57 | 261.82 | 0.676 | 9 | 10 |
| | 25 | Loose | - | **78347.00** | - | 0.690 | 9 | 9 |
| | 30 | Loose | - | **73380.68** | - | 0.705 | 12 | 17 |
| **200** | 10 | Tight | 112056.60 | 112056.60 | 204.91 | 0.663 | 4 | 6 |
| | 15 | Tight | 96448.95 | 96448.95 | 223.06 | 0.672 | 6 | 9 |
| | 20 | Tight | 86496.09 | 86496.09 | 260.02 | 0.680 | 8 | 11 |
| | 25 | Tight | - | **79353.27** | - | 0.693 | 9 | 13 |
| | 30 | Tight | - | **74023.13** | - | 0.703 | 11 | 16 |

À notre connaissance, les instances AP avec 300 et 400 nœuds n'ont pas encore été résolues. Donc, nous donnons des solutions pour ces instances dans le tableau. 4.9. Les résultats des instances Urand avec 1000 et 1500 nœuds sont donnés dans le tableau. 4.10). Nous avons généré au hasard la capacité des nœuds entre [1000, 5000] pour les capacités larges et entre [500, 3000] pour les faibles capacités. Les temps d'exécution sont raisonnables et vérifient Temps GPU $< 85s$ pour $n = 1000$ et Temps GPU $< 170s$ pour $n = 1500$.

Pour montrer la pertinence et vérifient le choix des hubs initiaux, nous avons comparé les hubs des solutions initiales avec ceux de la solution optimale/meilleure solution obtenue dans les colonnes C1 et C2 (dans chaque tableau). La colonne C1 présente le nombre de hubs de la solution initiale qui restent hubs dans la solution optimale/meilleure solution (environ 50%). La colonne C2 indique le nombre de groupes produits par $h()$ contenant au moins un hub dans la solution optimale (environ 75%). Ces résultats montrent que les critères de partitionnement $h()$ et le choix des hubs initiaux sont efficaces au moins pour les instances AP et Urand.

Tableau 4.9 – Résultats des instances AP.

| **n** | **p** | **Type Capacité** | **Sol GPU** | **Temps GPU** | **C1** | **C2** |
|---|---|---|---|---|---|---|
| **300** | 15 | Loose | 115381.92 | 0.84 | 7 | 9 |
| | | Tight | 115483.60 | 0.84 | 6 | 10 |
| | 20 | Loose | 104187.94 | 0.85 | 11 | 12 |
| | | Tight | 104561.84 | 0.85 | 9 | 14 |
| | 25 | Loose | 96390.77 | 0.86 | 13 | 17 |
| | | Tight | 95611.29 | 0.87 | 11 | 14 |
| | 30 | Loose | 90369.17 | 0.86 | 13 | 18 |
| | | Tight | 90541.98 | 0.90 | 12 | 18 |
| **400** | 15 | Loose | 116921.63 | 1.27 | 7 | 11 |
| | | Tight | 117288.68 | 1.31 | 8 | 11 |
| | 20 | Loose | 105175.25 | 1.53 | 13 | 13 |
| | | Tight | 105433.46 | 1.61 | 8 | 12 |
| | 25 | Loose | 97500.29 | 1.59 | 12 | 16 |
| | | Tight | 97697.74 | 1.67 | 14 | 15 |
| | 30 | Loose | 91837.62 | 1.91 | 16 | 17 |
| | | Tight | 92800.39 | 1.89 | 16 | 17 |

Tableau 4.10 – Résultats des instances Urand.

| n | p | Sol GPU | Temps GPU | C1 | C2 | n | p | Sol GPU | Temps GPU | C1 | C2 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| **100** | 2 | 36930.30 | 0.267 | 0 | 2 | **200** | 2 | 148235.45 | 0.607 | 0 | 1 |
| | 3 | 34763.02 | 0.281 | 1 | 3 | | 3 | 141622.84 | 0.640 | 1 | 1 |
| | 4 | 32608.27 | 0.346 | 1 | 3 | | 4 | 133722.45 | 0.656 | 2 | 3 |
| | 5 | 31107.69 | 0.357 | 2 | 4 | | 5 | 127220.02 | 0.672 | 2 | 3 |
| | 10 | 27156.85 | 0.385 | 6 | 7 | | 10 | 113512.16 | 0.717 | 5 | 7 |
| | 15 | 25413.15 | 0.392 | 6 | 10 | | 15 | 106530.60 | 0.710 | 5 | 10 |
| | 20 | 24558.68 | 0.460 | 9 | 15 | | 20 | 102712.04 | 0.732 | 8 | 14 |
| | 25 | 23771.12 | 0.467 | 12 | 19 | | 25 | 99872.58 | 0.738 | 11 | 18 |
| | 30 | 23236.81 | 0.483 | 13 | 22 | | 30 | 97901.93 | 0.765 | 14 | 24 |
| **300** | 2 | 328811.16 | 0.681 | 0 | 1 | **400** | 2 | 579982.34 | 1.041 | 0 | 1 |
| | 3 | 309116.03 | 0.717 | 1 | 2 | | 3 | 543927.51 | 1.092 | 1 | 2 |
| | 4 | 293794.08 | 0.738 | 1 | 3 | | 4 | 520216.07 | 1.145 | 0 | 3 |
| | 5 | 282551.88 | 0.763 | 2 | 5 | | 5 | 504730.28 | 1.117 | 2 | 4 |
| | 10 | 252271.94 | 0.778 | 6 | 8 | | 10 | 448461.25 | 1.217 | 4 | 6 |
| | 15 | 238210.44 | 0.806 | 6 | 9 | | 15 | 425089.56 | 1.302 | 7 | 9 |
| | 20 | 229130.64 | 0.836 | 9 | 11 | | 20 | 409044.39 | 1.482 | 9 | 13 |
| | 25 | 222517.79 | 0.852 | 11 | 16 | | 25 | 397467.64 | 1.560 | 8 | 16 |
| | 30 | 218054.27 | 0.880 | 14 | 22 | | 30 | 389792.28 | 1.842 | 11 | 20 |
| **1000** | 15 | 2726907.15 | 5.170 | 13 | 13 | **1500** | 20 | 2984509.88 | 11.901 | 7 | 11 |
| | 20 | 2628599.20 | 6.558 | 8 | 12 | | 30 | 2838577.10 | 14.280 | 13 | 19 |
| | 25 | 2555139.49 | 7.390 | 13 | 16 | | 40 | 2769084.04 | 17.591 | 18 | 27 |
| | 30 | 2518130.57 | 7.967 | 14 | 18 | | 50 | 2700460.70 | 20.795 | 22 | 39 |
| | 40 | 2449660.17 | 9.401 | 19 | 25 | | | | | | |
| | 50 | 2394607.99 | 10.428 | 24 | 37 | | | | | | |

En résumé, l'implémentation sur GPU a permis d'obtenir les solutions optimales ( si elles sont connues) et de résoudre des instances larges plus importantes pouvant atteindre 1500 nœuds en temps d'exécution raisonnable. Basé sur le résultat, nous pensons que notre approche a le potentiel de résoudre des instances encore plus larges (limitées par la capacité de mémoire du GPU, mais pas par le temps d'exécution).

### 4.4.6 Résolution de CSAHLP

La borne inférieure du nombre de hubs est $q = \left\lceil \frac{\sum_k O_k}{\sum_k G_k} \right\rceil$, c-a-d il n'y a pas de solution pour le CSAHLP avec un nombre de hubs $p < q$. À notre connaissance, il n'existe aucune méthode pour calculer le nombre minimum de hubs pour le CSAHLP. Le seul travail sur ce sujet est effectué par Chen J. F. [36] qui a donné une procé-

dure pour calculer le nombre minimum de hubs pour le cas de multiple affectations. Puisque nous ne connaissons pas de limite inférieure sur le nombre minimal de hub plus fin que $n/2$, nous résolvons le CSAHLP, en résolvant en parallèle le CSApHMP pour $p = q, ..., n/2$. La solution du CSAHLP est celle avec le coût minimal parmi les solutions obtenues. Pour cela, nous générons $M = \frac{n}{2} - q + 1$ solutions initiales $P_i$ chacune localise $i$ hubs, $q \leq i \leq n/2$. Nous exécutons en parallèle les $M$ kernels GA avec $P_i$, $q \leq i \leq n/2$ comme solution initiale qui localise $i$ hubs.

Chaque GA utilise $R$ blocs qui créent $R$ sous-populations (chacune avec $t$ individus) ; pour trouver une solution pour le CSApHMP. Par conséquent, les blocs $R * M$ sont nécessaires pour calculer les $M$ solutions pour le CSAHLP. Le pseudo code Cuda pour le CSAHLP est le suivant :

1. Générer en parallèle les solutions initiales $P_q, ..., P_{n/2}$.
2. Définir les blocs et la grille comme suit :
   $dim3\ dimBlock(t, 1)$ ; $dim3\ dimGrid(R * M, 1)$ ;
3. Exécuter en parallèle le kernel GA avec les solutions initiales $P_q, ..., P_{n/2}$.
   $GA <<< dimGrid, dimBlock >>> (P_q, ..., P_{n/2})$ ;
4. Sélectionner la solution avec le coût minimal parmi les M solutions ;

### Résultats expérimentaux

Le tableau. 4.11 compare nos résultats par rapport à ceux obtenus par Ernst et Krishnamoorthy [58] pour le CSAHLP et donne les hubs de la solution optimale / meilleure, pour les instances AP jusqu'à 200 nœuds (les grandes instances publiées pour ce problème à notre connaissance). Récemment Corberan et al. [44] ont généré des instances jusqu'à 250 nœuds, mais pour le CSAHLP avec des capacités identiques pour le nœud et des capacités de liaison pour les arcs. Le temps d'exécution sur GPU requis (non reportés dans le tableau. 4.11) sont compris entre 0,03 et 19(s).

Tableau 4.11 – Résultats des instances AP pour CSAHLP.

| Instance | Sol CPU | Sol GPU | Hubs de la Sol GPU |
|---|---|---|---|
| 10LL | 224250.05 | 224250.05 | 3, 4, 7 |
| 10LT | 250992.26 | 250992.26 | 1, 4, 5, 10 |
| 10TL | 263399.94 | 263399.94 | 4, 5, 10 |
| 10TT | 263399.94 | 263399.94 | 4, 5, 10 |
| 20LL | 234690.96 | 234690.96 | 7, 14 |
| 20LT | 253517.40 | 253517.40 | 10, 14 |
| 20TL | 271128.18 | 271128.18 | 7, 19 |
| 20TT | 296035.40 | 296035.40 | 1, 10, 19 |
| 25LL | 238977.95 | 238977.95 | 8, 18 |
| 25LT | 276372.49 | 276372.49 | 9, 16, 25 |
| 25TL | 310317.64 | 310317.64 | 9, 23 |
| 25TT | 348369.14 | 348369.14 | 9, 16, 25 |
| 40LL | 241955.70 | 241955.70 | 11, 29 |
| 40LT | 272218.32 | 272218.32 | 14, 26, 30 |
| 40TL | 298919.01 | 298919.01 | 14, 19 |
| 40TT | 354874.10 | 354874.10 | 14, 19, 40 |
| 50LL | 238520.58 | 238520.58 | 15, 35 |
| 50LT | 272897.48 | 272897.48 | 6, 26, 32, 46 |
| 50TL | 319015.77 | 319015.77 | 3, 24 |
| 50TT | 417440.99 | 422430.31 | 6, 12, 26, 48 |
| 100LL | 246713.96 | 246713.96 | 29, 64, 73 |
| 100LT | 256638.38 | **256390.05** | 29, 68, 76 |
| 100TL | 362950.09 | 362950.09 | 44, 52 |
| 100TT | 474680.32 | 474680.32 | 5, 34, 52, 86, 95 |
| 200LL | 241992.97 | 241992.97 | 43, 159 |
| 200LT | 268894.41 | 269273.40 | 41, 124, 168, 171 |
| 200TL | 273443.81 | 273443.81 | 54, 95, 186 |
| 200TT | 292754.91 | **292142.16** | 54, 113, 168, 186 |

## 4.5 Résolution du problème UMApHMP

Nous présentons dans cette section un GA pour résoudre le problème p-médian hub avec multiple allocations sans capacité (UMApHMP). Dans [18] nous avons proposé un simple GA et son implémentation GPU pour résoudre ce problème. Le GA utilise le Set-Based-Representation [104] comme codage. La solution initiale est obtenue à partir d'un algorithme qui permet d'identifier les nœuds susceptibles d'être des hubs optimaux. Dans cette section nous apportons des modifications à cette approche au niveau de : l'allocation des nœuds aux hubs (un nœud est affecté au hub le plus proche ou aléatoirement), le croisement (échange de plusieurs clusters au lieu de deux) et au niveau de l'implémentation GPU (les matrices des distances et des flux sont découpées par blocs et stockées dans les mémoires partagées au lieu de la mémoire globale).

### 4.5.1 Présentation de l'UMApHMP

L'UMApHMP peut être indiqué de manière informelle comme suit. Étant donné un ensemble $N$ de $n$ nœuds, trouvez un sous-ensemble de $p$ hubs $H \subset N$ qui minimisent :

$$\sum_{i,j \in N} min_{k,l \in H} C_{ijkl} x_{ijkl}$$

où la variable de décision $x_{ijkl} = 1$, si le flux $w_{ij}$ de $i$ à $j$ passe par les hubs $k$ et $l$ ($k$ peut être égal à l) et 0 sinon. Étant donné qu'il n'y a pas de contraintes de capacité sur les hubs ou les liens du réseau, il existe toujours une solution réalisable dans laquelle chaque flux est entièrement routé sur un seul chemin [57], ce qui justifie le fait que les $x_{ijkl}$ sont des variables binaires. Les contraintes générales imposées par l'UMApHMP sont les suivantes. Le nombre de hubs est $p$. Les flux et les coûts de transport sont supposés être connus et déterministes. Le trafic entre deux nœuds $i$ et $j$ doit être acheminé via un ou deux hubs établis $k$ et $l$. Tous les hubs sont interconnectés et deux non-hubs ne peuvent être directement connectés. Chaque non-hub peut-être alloué à plusieurs hubs. Une formulation plus précise de l'UMApHMP peut être trouvée dans Boland et al. [27] et Kratica et al. [88].

La figure 4.5 (a) montre un exemple d'un réseau des hubs avec $n = 5$ nœuds. Le nombre de hubs à localiser est $p = 2$. La figure 4.5 (b) présente une solution pour le cas d'affectation unique avec les hubs à localiser sont situés aux nœuds 2 et 4. Les non-hubs 1 et 5 sont affectés au hubs 4 et le non-hubs 3 au hub 2. Par exemple, le flux de 1 à 3 prend le chemin 1 - 4 - 2 - 3. Une solution d'affectation multiple est présentée dans la figure 4.5 (c) avec les hubs à localiser sont situés aux

nœuds 2 et 4. Le non-hub 3 est affecté aux hubs 2 et 4, le non-hub 1 est affecté aux hubs 2 et 4 et le non-hubs 5 est affecté au hub 4. Le flux de 1 à 3 prend le chemin ayant le coût minimal parmi les trois chemins 1 - 4 - 2 - 3 ou 1 - 2 - 3 ou 1 - 4 - 3.

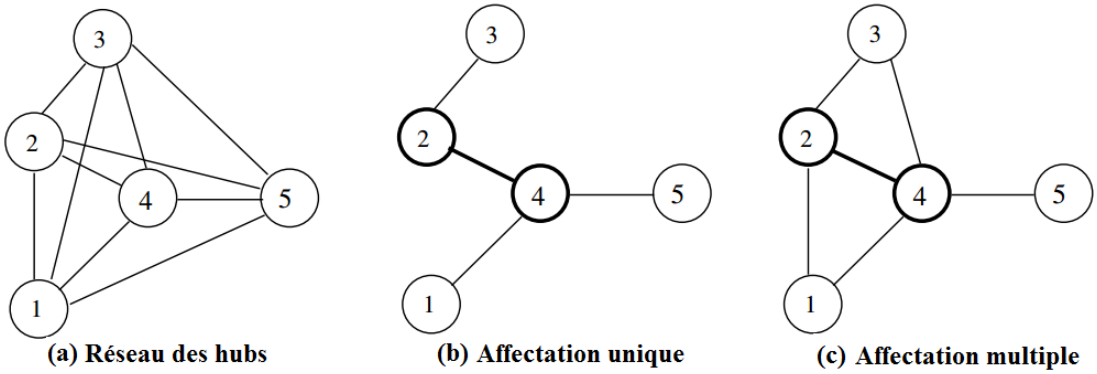



FIGURE 4.5 – Un exemple de p-médian hub avec des solutions unique et multiple affectations.

### 4.5.2 Choix des hubs initiaux

Comme pour le CSApHMP, nous avons réalisé des tests sur différentes instances pour concevoir un algorithme simple et efficace pour partitionner l'ensemble des nœuds en $p$ groupes disjoints, chacun d'entre eux pouvant contenir un hub optimal. Cette méthodologie a déjà donné de bons résultats pour la variante CSApHMP (section 4.4.2). Dans cette section nous allons présenter l'adaptation de cette méthodologie pour la variante UMApHMP. Nous avons effectué plusieurs tests sur plusieurs benchmarks connus (y compris les larges instances aléatoires). Le pourcentage du nombre de groupes contenant chacune au moins un hub optimal est d'environ 82% (voir les résultats expérimentaux pour plus de détails).

Notre algorithme est fondé sur les deux observations suivantes :
i) Dans une solution optimale de l'UMApHMP, un non-hub est nécessairement affecté au hub le plus proche de lui. ii) Un nœud $k$ avec le plus grand index $h(k) = O_k + D_k$ est un hub optimal potentiel et les nœuds les plus proches sont susceptibles d'être aussi des hubs optimaux, ce sont plutôt les nœuds affectés au hub $k$.
L'algorithme suivant partitionne l'ensemble des nœuds en $p$ groupes disjointes $G_i$ ; chacun pouvant contenir un hub de la solution optimale ou la meilleure solution.

**Algorithme 5** : Algorithme de partitionnement [18].

$L$ = la liste des nœuds triés par ordre décroissant en fonction du critère $h()$ ;
$r = \left\lceil \frac{n}{p} \right\rceil$ ; $i = 1$ ;
**tant que** $(i < p)$ **faire**
  $h_i$ = est le premier nœud de $L$ non affecté à $G_1 \cup ... \cup G_{i-1}$ ;
  **si** $(i < p)$ **alors**
    $G_i$ = l'ensemble de $r$ nœuds les plus proches à $h_i$ (comprenant $h_i$)
  **sinon**
    $G_i$ = les nœuds restants dans $L$ ;
  **fin si**
  Retirer $G_i$ de $L$ $(L = L \setminus G_i)$ ;
  $i = i + 1$ ;
**fin tant que**

Cet algorithme étant dédié à l'UMApHMP, les $G_i$ peuvent être non disjoints et par conséquent peuvent contenir différents nombres de nœuds au lieu de $r$ nœuds. Identifier le hub optimal potentiel dans chaque $G_i$ est une tâche plus délicate. En effet, le hub potentiel en $G_i$ n'est pas nécessairement $h_i$, mais peut être un nœud proche de $h_i$ (selon certains critères). Nous avons testé différents critères pour sélectionner le hub optimal dans chaque groupe. Nous avons observé que le nœud $k \in G_i$ avec petit $\sum_{o,d \in C_i} (\chi d_{ok} + \delta d_{kd}) w_{od}$ est le meilleur hub candidat dans $G_i$ pour plusieurs benchmarks mais pas pour tous.

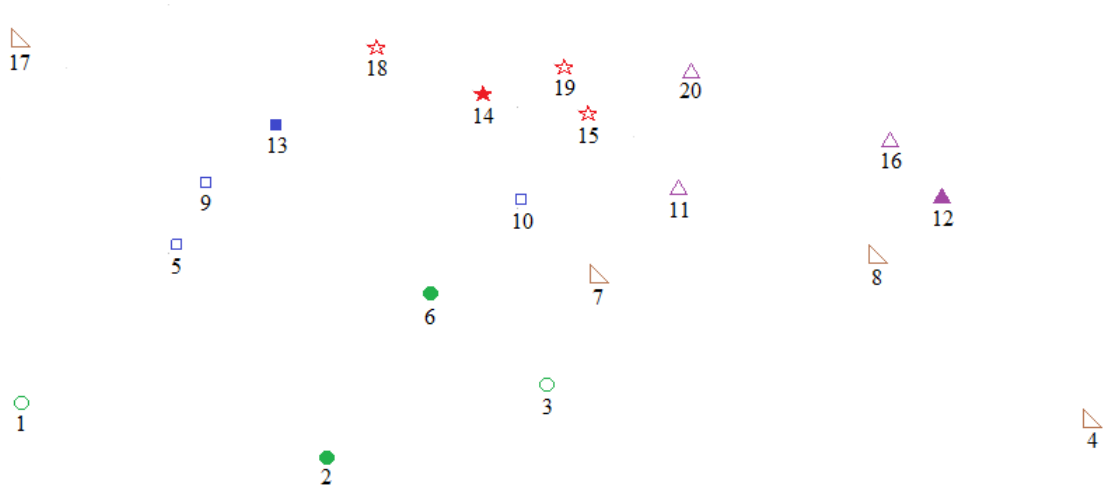



FIGURE 4.6 – Partitionnement des hubs optimaux pour l'instance AP 20.5.

La figure 4.6 illustre ce propos sur l'instance AP 20.5 avec $n = 20$ et $p = 5$. La liste ordonnée est $L$ = (14, 15, 2, 19, 6, 13, 20, 16, 3, 5, 9, 4, 12, 18, 10, 7, 8, 17, 1, 11). Les cinq groupes sont $G_1$ = (14, 19, 15, 18), $G_2$ = (2, 3, 1, 6), $G_3$ = (13, 9, 10, 5), $G_4$ = (20, 16, 11, 12 ), $G_5$ = (4, 7, 8, 17) et sont reproduits sur la figure 4.6 par différents symboles ($G_1$ avec étoiles, $G_2$ avec cercles, etc.). Les hubs optimaux sont $\{14, 2, 13, 12, 6\}$ et sont représentés par des objets remplis sur cette figure. Comme nous l'avons vu, seul le $G_5$ ne contient pas de hub optimal. Si nous sélectionnons $h_i$ comme hub dans $G_i$, nous obtenons l'ensemble des hubs initiaux $\{14, 2, 13, 20, 4\}$. Trois d'entre eux sont en fait des hubs optimaux, mais pas les deux autres.

### 4.5.3 Algorithme génétique

#### Codage et solution initiale

Dans la littérature la plupart des GAs proposés pour résoudre l'UMApHMP proposent soit un codage binaire ou entier pour représenter une solution. Ces deux types de codage sont un peu couteux dans l'évaluation de la solution dans le cas de l'UMApHMP.

Soit $H$ l'ensemble des hubs. Dans le codage binaire, il n'y a qu'un seul tableau qui contient des $\{0$ *ou* $1\}$ pour spécifier les nœuds qui sont choisis comme des hubs. Par conséquent, pour calculer le coût de transport d'un nœud $i$ vers un nœud $j$, il faut calculer tous les chemins $i-k-l-j$ et $i-k-j$ $\forall\ k, l \in H$ et sélectionner le minimum entre eux. De l'autre côté, dans le codage avec des entiers, il n'y a de même qu'un seul tableau, mais cette fois-ci, il contient les affectations des nœuds aux hubs. Ainsi, pour calculer le coût de transport d'un nœud $i$ affecté au hub $k$ vers un nœud $j$ alloué au hub $l$, il faut comparer les coûts des trois chemins $i-k-l-j$, $i-k-j$ et $i-l-j$ et sélectionner le minimum. Même si le nœud $i$ n'est pas affecté au hub $l$, mais il se peut que Coût($i$-$l$-$j$)<Coût($i$-$k$-$j$) ou Coût($i$-$l$-$j$)<Coût($i$-$k$-$l$-$j$) (multiple allocations). Il est clair que le codage avec des entiers fait moins de calcul dans l'évaluation de la solution par rapport au codage binaire. La bonne représentation de la solution est un critère important pour la performance de l'algorithme génétique.

Naeem et Ombuki-Berman [104] ont proposé un codage basé sur des clusters, qui est utilisé pour représenter une solution du problème CSAHLP. Nous avons adapté ce codage pour l'UMApHMP. Dans la représentation Set-based Representation, chaque cluster est un ensemble d'entier chacun représente un nœud. Le premier nombre dans chaque cluster représente le hub et les restes des nombres représentent les non-hubs affectés à cet hub. Cette représentation a été proposé pour résoudre le CSAHLP. L'ensemble $N$ des nœuds est partitionné en clusters $C_i$, tels que $\forall i, j\ C_i \cap C_j = \emptyset$, c-à-d que chaque nœud est alloué à un seul hub. Par contre, dans l'UMApHMP

un nœud peut être alloué à plusieurs hubs, ce qui fait qu'il peut apparaître dans plusieurs clusters différents.

La solution initiale du GA localise les $p$ hubs, à l'aide de Algorithme 5 qui partitionne l'ensemble des nœuds en $p$ clusters et sélectionne un hub dans chaque cluster en se basant sur le critère $h()$. Ensuite pour chaque non-hubs nœud est alloué au hub le plus proche (en terme de distance). Ce qui fait que notre solution initiale est une solution initiale du problème de l'USApHMP. Au fur et à mesure, les opérateurs génétiques vont changer cette solution pour en une solution de l'UMApHMP. L'utilisation des clusters comme codage pour représenter une solution de l'UMApHMP présente deux avantages : (i) la correction des individus est plus facile et (ii) la fonction d'évaluation calcule plus rapidement le coût d'une solution (elle calcule exactement les chemins qui sont représentés dans le codage de la solution et pas plus).

Le codage avec des clusters qui représente la solution multi allocations du problème de la figure 4.5 est : $C_1 = \{\mathbf{2}, 3, 1\}$ , $C_2 = \{\mathbf{4}, 1, 3, 5\}$.

### Les Opérateurs génétiques

- *Génération de la population et sélection* : Comme dans les autres GA déjà proposé, 1-*exchange*() est utilisé pour générer un nouvel individu (on change un hub avec un non-hub dans le même groupe). Dans notre GA, on génère un ensemble de populations qui sont traitées en parallèle. L'opérateur de sélection ne favorise pas un individu à un autre. La sélection se fait d'une manière aléatoire et chaque individu est choisi une seule fois dans chaque population.

- *Croisement* par échange de multi-cluster (MCEC) [104] est utilisé comme opérateur de croisement dans ce GA. Il consiste à échanger plusieurs clusters entre deux parents, afin de produire deux descendants. Plus précisément, Soient $p1 = \{C_1, C_2, ..., C_p\}$ et $p2 = \left\{C'_1, C'_2, ..., C'_p\right\}$ deux parents sélectionnés dans MCEC, les solutions enfants $ch1$ et $ch2$ sont produites en échangeant un ou plusieurs clusters (au maximum 4 clusters dans notre implémentation) entre $p1$ et $p2$. Le processus d'échange est suivi par un processus de correction dans lequel les solutions irréalisables sont corrigées.

  Le processus de correction est basé sur deux étapes de correction : la première étape est la vérification du nombre de hubs et la deuxième est la vérification des affectations (affecter les nœuds non affectés à aucun hub).

  - Puisque nous échangeons le même nombre de clusters entre parents, le nombre de clusters ne dépasse jamais $p$. Par contre, il peut diminuer. Cela se produit lorsque le même individu a deux clusters avec le même hub.

Dans ce cas le cluster avec moins d'affectations aura changé son hub par un non-hub dans le même cluster.

- Un hub ne doit pas apparaître dans un autre cluster comme un simple non-hub ; tous les nœuds non affecté seront affectés au hub le plus proche.

Par conséquent, après ce processus, nous obtenons deux solutions réalisables notées par $c1$ et $c2$.

- *Mutation* : Les opérations utilisés dans ce GA sur 20% de la population sont :
  - Le déplacement d'un nœud (utiliser à 10% de la population) : un nœud $i$ est détaché aléatoirement d'un cluster $C_j$ et inséré dans un autre cluster $C_k$ de la même solution. L'opération de mutation par déplacement ne peut être effectuée que pour les clusters ayant au moins deux nœuds.
  - La permutation des nœuds (utiliser à 10% de la population) : permettre de permuter un nœud $i \in C_{i'}$ avec un autre nœud $j \in C_{j'}$ de la même solution.

### 4.5.4 Résultats expérimentaux

Étant donné que nous utilisons le même codage que celui du GA pour le VRP dynamique, l'implémentation GPU de cet approche est similaire à celle présentée dans la section 3.6.5.
Nous reportons nos résultats pour les cas d'affectation unique et multiple pour chaque instance testée. Ainsi, les coûts des solutions peuvent être comparés dans les deux cas. Nous comparons nos résultats de l'affectation unique à ceux d'Ilic et al. [81] et les résultats de l'affectation multiple à ceux de Kratica et al. [88].

On garde les notations déjà utilisées dans les tableaux précédents avec *Clust opt* est le nombre de clusters contenant chacun au moins un hub de la solution optimale ou la meilleure pour l'UMApHMP. La moyenne de ces nombres est indiquée en gras à la fin de cette colonne. Certains résultats pour le cas d'affectation unique (USApHMP) présentés ici sont présentés dans la section 4.3.3. Pour tous les benchmarks considérés, notre implémentation atteint les optimaux ou les meilleures solutions en temps d'exécution $\leq 7s$ pour $n \leq 419$ et entre $10s$ et $15s$ pour $n = 1000$ et entre $25s$ et $600s$ pour $n \in [1500, 6000]$.

Le tableau. 4.12 fournit les résultats sur les instances AP avec 300 et 400 nœuds. Nous ne connaissons pas d'autres solutions de l'UMApHMP pour ces instances afin de les comparer avec nos résultats. La moyenne de la colonne *Clust opt* est de 87%. Cela signifie qu'en moyenne, le pourcentage de groupes contenant chacune au

moins un hub optimal est de 87% (ou que les hubs optimaux de chaque problème sont répartis sur 87% des groupes). Cela montre l'efficacité de notre algorithme de partitionnement.

TABLEAU 4.12 – Résultats des larges instances AP.

| n | p | Type affect | Sol CPU | Sol GPU | Clust opt |
|---|---|---|---|---|---|
| 300 | 5 | M | - | 170679.01 | 5 |
| | 10 | M | - | 131788.68 | 9 |
| | 15 | M | - | 112806.91 | 13 |
| | 20 | M | - | 101884.17 | 17 |
| 400 | 5 | M | - | 172818.43 | 5 |
| | 10 | M | - | 133487.54 | 8 |
| | 15 | M | - | 114314.49 | 13 |
| | 20 | M | - | 103404.57 | 17 |
| | | | | | **87%** |

Dans le tableau 4.13, nous avons signalé des nouveaux meilleurs coûts pour les instances PlanetLab pour les cas d'affectation unique et multiple. Les anciennes meilleures solutions pour l'USApHMP reportées dans ce tableau sont dues à Ilic et al. [81]. Étant donné qu'il n'y a pas d'autres résultats de l'UMApHMP connus pour ces instances, nos résultats deviennent désormais les seules dans la littérature. La moyenne de la colonne *Clust opt* est de 67,3%. Ce pourcentage est assez faible par rapport aux autres benchmarks. Nous pensons que cela est dû au flux qui est égale à 1 dans les instances PlanetLab.

Tableau 4.13 – Résultats des instances PlanetLab * -2005.

| Inst. | n | p | Type affect | Sol CPU | Sol GPU | Clust opt |
|---|---|---|---|---|---|---|
| 1 | 127 | 12 | M | - | 2566990 | 7 |
| | | | S | 2927946 | 2904434 | |
| 2 | 321 | 19 | M | - | 16518458 | 10 |
| | | | S | 18579238 | 18329984 | |
| 3 | 324 | 18 | M | - | 18238014 | 10 |
| | | | S | 20569390 | 20284132 | |
| 4 | 70 | 9 | M | - | 682596 | 6 |
| | | | S | 739954 | 730810 | |
| 5 | 374 | 20 | M | - | 20653586 | 15 |
| | | | S | 25696352 | 25583240 | |
| 6 | 365 | 20 | M | - | 19365696 | 15 |
| | | | S | 22214156 | 22151862 | |
| 7 | 380 | 20 | M | - | 27417830 | 14 |
| | | | S | 30984986 | 30782956 | |
| 8 | 402 | 21 | M | - | 28540846 | 15 |
| | | | S | 30878576 | 30636170 | |
| 9 | 419 | 21 | M | - | 26593496 | 16 |
| | | | S | 32959078 | 32649752 | |
| 10 | 414 | 21 | M | - | 26355946 | 13 |
| | | | S | 32836162 | 28211380 | |
| 11 | 407 | 21 | M | - | 24664598 | 14 |
| | | | S | 27787880 | 27644374 | |
| 12 | 414 | 21 | M | - | 22317134 | 15 |
| | | | S | 28462348 | 28213748 | |
| | | | | | | **67.3%** |

Ilic et al. [81] et Kratica et al. [88] ont reporté les meilleurs résultats connus pour les instances de Urand pour $n = 100, 200, 300, 400$ où ($\chi = 3$ et 1, $\alpha = 0{,}75$, $\delta = 2$ et 1). Notre implémentation GPU obtient les mêmes résultats. Ainsi, dans le tableau. 4.14, nous reportons uniquement la colonne *Clust opt* pour ces instances avec ($\chi = 1$, $\alpha = 0{,}75$, $\delta = 1$). On observe que les nombres dans cette colonne ne sont pas très sensibles à la variation de $n$. Ici aussi, la moyenne de la colonne *Clust opt* est assez élevée.

TABLEAU 4.14 – Résultats des larges instances Urand ($\chi = 1$, $\alpha = 0{,}75$, $\delta = 1$).

| n | p | Clust opt | n | p | Clust opt | n | p | Clust opt | n | p | Clust opt |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 100 | 2 | 2 | 200 | 2 | 2 | 300 | 2 | 2 | 400 | 2 | 2 |
| | 3 | 3 | | 3 | 2 | | 3 | 3 | | 3 | 3 |
| | 4 | 4 | | 4 | 3 | | 4 | 4 | | 4 | 4 |
| | 5 | 5 | | 5 | 5 | | 5 | 4 | | 5 | 4 |
| | 10 | 9 | | 10 | 9 | | 10 | 9 | | 10 | 8 |
| | 15 | 13 | | 15 | 12 | | 15 | 14 | | 15 | 12 |
| | 20 | 17 | | 20 | 16 | | 20 | 17 | | 20 | 16 |
| | | | | | | | | | | | **85.6%** |

Dans le tableau. 4.15, nous reportons de nouveaux résultats (pour les cas d'affectation unique et multiple) sur les instances Urand à 1 000 nœuds avec ($\chi = 1$, $\alpha = 0{,}75$, $\delta = 1$), car Ilic et al. [81] et Kratica et al. [88] n'ont reporté que les résultats pour ($\chi = 3$, $\alpha = 0{,}75$, $\delta = 2$).

TABLEAU 4.15 – Résultats des larges instances Urand.

| n | p | Type affect | Sol GPU | Clust opt |
|---|---|---|---|---|
| 1000 | 2 | M | 3380404.48 | 2 |
| | | S | 3644705.83 | |
| | 3 | M | 3086100.81 | 3 |
| | | S | 3397386.83 | |
| | 4 | M | 2907515.00 | 4 |
| | | S | 3206255.10 | |
| | 5 | M | 2825585.52 | 4 |
| | | S | 3119060.41 | |
| | 10 | M | 2572595.51 | 9 |
| | | S | 2823592.72 | |
| | 15 | M | 2460725.82 | 14 |
| | | S | 2679974.67 | |
| | 20 | M | 2393602.47 | 16 |
| | | S | 2577234.14 | |
| | | | | **88.1%** |

Le tableau. 4.16 présente nos résultats avec les paramètres ($\chi = 3$, $\alpha = 0{,}75$, $\delta = 1$) pour les larges instances Urand jusqu'à 6000 nœuds. À notre connaissance, aucune solution des instances de taille supérieure à 1000 n'a été publié. L'efficacité de notre algorithme de partitionnement en groupe et de l'implémentation GPU est clairement établie dans la résolution de ces larges instances aléatoires. Par rapport à ce travail, aucun d'autre traitant des problèmes de localisation des hubs n'a présenté de résultats aussi importants.

Tableau 4.16 – Résultats des larges instances urand générées par nous-même.

| **n** | **p** | **Type affect** | **Sol GPU** | **Clust opt** | **n** | **p** | **Type affect** | **Sol GPU** | **Clust opt** |
|---|---|---|---|---|---|---|---|---|---|
| **1500** | 20 | M | 448739772 | 17 | **4000** | 20 | M | 3163446506 | 17 |
| | | S | 454787506 | | | | S | 3234999192 | |
| | 30 | M | 406017592 | 26 | | 30 | M | 2935022500 | 23 |
| | | S | 407155164 | | | | S | 2983891783 | |
| | 40 | M | 377761261 | 36 | | 40 | M | 2727794413 | 33 |
| | | S | 380114045 | | | | S | 2769550514 | |
| | 50 | M | 358245622 | 44 | | 50 | M | 2604040619 | 41 |
| | | S | 363586538 | | | | S | 2644606684 | |
| **2000** | 20 | M | 800331930 | 17 | **5000** | 20 | M | 5001798606 | 16 |
| | | S | 805749722 | | | | S | 5085803132 | |
| | 30 | M | 724383243 | 27 | | 30 | M | 4610179267 | 25 |
| | | S | 733375448 | | | | S | 4656787498 | |
| | 40 | M | 679709297 | 37 | | 40 | M | 4277839570 | 32 |
| | | S | 686515363 | | | | S | 4353561395 | |
| | 50 | M | 647033467 | 44 | | 50 | M | 4060895916 | 39 |
| | | S | 655938000 | | | | S | 4143849388 | |
| **3000** | 20 | M | 1777794117 | 17 | **6000** | 20 | M | 7217166936 | 15 |
| | | S | 1804950952 | | | | S | 7398401957 | |
| | 30 | M | 1616819166 | 24 | | 30 | M | 6586643628 | 24 |
| | | S | 1642145354 | | | | S | 6675723961 | |
| | 40 | M | 1522572071 | 33 | | 40 | M | 6197499299 | 34 |
| | | S | 1538548764 | | | | S | 6293053841 | |
| | 50 | M | 1445702469 | 42 | | 50 | M | 5902887457 | 43 |
| | | S | 1468780124 | | | | S | 5999780197 | |
| | | | | | | | | | **84%** |

## 4.6 Conclusion

Dans ce chapitre nous avons proposé un algorithme génétique parallèle et son implémentation sur GPU pour résoudre différentes variantes du problème de localisation des hubs. Le GA proposé adapte son codage, sa solution initiale, son critère de localisation des hubs susceptibles d'être des hubs dans la solution optimale, ses opérateurs génétiques et son implémentation à chaque variante traitée afin d'exploiter les performances du GPU. Pour les cinq variantes du HLP traitées, les résultats expérimentaux obtenus sont comparés aux meilleures solutions connues sur des instances jusqu'à 1000 nœuds et sur des instances plus larges jusqu'à 6000 nœuds générées par nous-même. Ils montrent que nos implémentations surpassent les heuristiques les plus connues en termes de qualité de la solution et en temps d'exécution. Nos implémentations ont permis de résoudre des instances non résolues jusqu'à présent. Nous pensons que notre approche a le potentiel de résoudre des instances plus importantes (limitées par la capacité de la mémoire du GPU, mais pas par le temps d'exécution).

Chapitre 5

# Problème de localisation des hubs avec des données incertaines

## 5.1 Introduction

En pratique, les données d'un problème d'optimisation peuvent être incertaines, inexactes, bruitées ou susceptibles de changer à l'avenir. Une solution optimale calculée en utilisant les paramètres actuels peut être fortement affectée par des perturbations, devenant sous optimale ou même infaisable. Ce que la plupart des décideurs appellent une solution robuste est une solution qui résiste le plus possible à de telles perturbations.

Le choix d'un ensemble d'incertitudes et d'un critère de robustesse est essentiel, afin de résoudre un problème d'optimisation robuste. L'ensemble d'incertitudes peut être défini par un ensemble convexe, par ex. un polyèdre Ben-Tal et al. [15] et Bertsimas et al. [26], ou par une affectation de valeurs possibles à chaque paramètre du modèle, appelé scénario. Dans les scénarios d'intervalle, chaque paramètre peut prendre n'importe quelle valeur dans un intervalle donné. Tandis que dans les scénarios discrets considérés dans notre étude, ses valeurs possibles sont explicitement listées. En pratique, la plupart des entreprises enregistrent des archives ou des données historiques qui peuvent être exploitées pour fournir des scénarios réalistes.

Soit un problème de minimisation avec un ensemble réalisable $X$ et un ensemble $S$ de $|S|$ scénarios discrets. Soit $f(x, s)$ le coût de la solution $x$ dans le scénario $s$, $x_s^*$ est l'optimum pour le scénario $s$, et $f_s^*$ son coût. La version *min-max* [148] cherche une solution minimisant le pire coût sur tous les scénarios, c-a-d, $\min_{x \in X} \max_{s \in S} \{f(x, s)\}$. La version du modèle *min-max*, Ogryczak W. [108] utilise les autres scénarios, du

pire au meilleur, pour rompre les liens. La robustesse lexicographique, *ρ-robustness* Kalai et al. [83] est similaire, mais inclut un seuil de tolérance, pour éviter de discriminer entre des solutions ayant des valeurs similaires. Le *min–max regret* version $\min_{x \in X} \max_{s \in S} \{f(x,s) - f_s^*\}$, est traité par exemple dans Candia et al. [32].
Dans ce qui suit, nous passons en revue les travaux connexes à ce problème. Ensuite, nous présentons la formulation du problème et l'approche de résolution.

## 5.2 Revue de la littérature

Parmi les travaux traitant des problèmes de localisation des hubs avec des données incertaines citons :

Alumur et al. [8] ont proposé des modèles robustes, pour optimiser les problèmes des hubs avec affectation unique et multiple. Ils ont considéré l'incertitude de la demande et le coût d'installation fixe des hubs en utilisant une approche basée sur des scénarios et ont démontré la sous-optimalité de la solution déterministe en présence d'incertitude. Dans le même sens, Ghaffari-Nasab et al. [70] présentent des problèmes robustes de localisation des hubs avec capacité et des affectations multiples dans lesquelles l'incertitude de la demande est modélisée avec un intervalle d'incertitude. Les auteurs considèrent l'incertitude des contraintes de capacité et utilisent un budget d'incertitude pour chacun d'entre eux. Boukani et al. [28] étudient les mêmes problèmes de localisation des hubs avec capacité et des affectations multiples, mais les incertitudes se rapportent aux coûts et aux capacités des hubs. Les auteurs présentent des modèles de min-max regrets impliquant à cinq scénarios pour chaque paramètre incertain.

Makui et al. [97] ont développé un modèle d'optimisation robuste multi-objectif pour le problème de localisation p-hub avec capacité. D'autre part, Zetina et al. [150] ont présenté trois problèmes robustes différents de l'UHLP. Le premier est le problème de localisation des hubs sans capacité avec des demandes incertaines (UHLP-D), le second est le problème de localisation des hubs sans capacité avec des coûts de transport incertains (UHLP-TC) et le troisième est le problème de localisation des hubs sans capacité avec des demandes et des coûts de transport incertains (UHLP-DTC). Ce dernier problème considère que les incertitudes des deux classes sont indépendantes les unes des autres. Dans ces problèmes robustes de l'UHLP, l'objectif est de minimiser la somme des coûts de mise en place des hubs et des coûts de transport de la demande (flux), dans le pire des cas qui pourrait se produire pour les données incertaines. Des modèles robustes sont proposés, ainsi que des formulations mathématiques qui ne sont pas linéaires en raison de la nature min-max des fonctions objectifs.

Ghaderi et Rahmaniani [69], ont proposé une extension au problème p-médian hub sans capacité avec affectation unique avec incertitude où les demandes sont stochastiques. Le modèle développé minimise les coûts totaux de transport prévus, tout en limitant le regret relatif ($\rho$-*robust optimization*) de chaque scénario. Pour résoudre efficacement le modèle, une approche en deux étapes est étudiée pour concevoir deux heuristiques hybrides. Ils ont appliqué d'abord la recherche du voisinage variable (VNS) ou l'optimisation d'essaim de particules (PSO) pour trouver la meilleure combinaison pour l'emplacement des hubs. Ensuite, ils ont utilisé une recherche tabou (TS) pour l'allocation des non-hubs aux hubs. Les résultats obtenus par la métaheuristique sont comparés avec les résultats du CPLEX, mais dans ce travail ils n'ont pas pu résoudre des instances $> 50$ nœuds.

Rahmaniani et al. [116] ont étudié des problèmes de localisation des hubs multiples avec des flux et des distances incertains. Ils ont proposé une heuristique basée sur l'algorithme de recherche de voisinage variable (VNS) pour résoudre le problème et ont comparé leurs résultats avec le solveur CPLEX sur les instances extraites de l'ensemble d'instances CAB. De même dans ce chapitre, nous montrons comment adapter les GAs précédents pour résoudre le RUSApHMP avec des scénarios discrets.

## 5.3 Description du problème

Le RUSApHMP [1] avec des scénarios discrets, consiste à localiser les $p$ hubs et d'affecter chaque non-hubs à un seul hub de sorte que le coût de transport total soit minimal. En considérant l'incertitude sur les demandes, chaque scénario définit les demandes des clients dans une période précise. Le critère de robustesse utilisé est le modèle min-max, où on minimise le coût dans le pire des cas sur tous les scénarios.

Dans la littérature, la plupart des travaux sur les HLPs avec des données incertaines utilisent des modèles de min-max, qui sont directement résolus par des solveurs de MIP (CPLEX). Ces approches sont limités à la résolution d'instances de moins de 50 clients. Nous avons proposé une implémentation GPU basée sur GA pour résoudre RUSApHMP avec le critère des pires des cas [20]. Cette approche, nous a permis de résoudre des larges instances jusqu'à 4000 nœuds en temps raisonnables et avec de meilleures solutions robustes (vérifier avec CPLEX pour les instances $\leq 40$ nœuds). À notre connaissance, il s'agit de la première solution GPU du RUSApHMP avec des scénarios discrets.

1. Robust Uncapacitated Single Allocation p-hub Median Problem

## 5.4 Formulation mathématique

Le RUSApHMP avec des demandes incertaines et avec des scénarios discrets, consiste à choisir $p$ hubs parmi $n$ nœuds, de telle sorte que le coût total est le minimum des pires des cas sur tous les scénarios. Notons que :

- $N$ l'ensemble des nœuds ; $i, j, k, l \in \{1, 2, \ldots, n\}$ ;
- $S$ l'ensemble des scénarios ; $s \in \{1, 2, \ldots, |S|\}$ ;
- $p$ le nombre de hubs à localiser ;
- $q_s$ la probabilité du scénario s ;
- $w_{ij}^s$ la quantité de flux originaire du nœud $i$ à $j$, sous scénario $s$ ;
- $d_{ij}$ la distance entre le nœud $i$ et le nœud $j$ ;
- $O_i^s = \sum_j w_{ij}^s$ le flux sortant total dans le nœud $i$ sous scénario $s$ ;
- $D_i^s = \sum_j w_{ji}^s$ le flux entrant total dans le nœud $i$ sous scénario $s$ ;
- $C_{iklj}^s$ le coût de transport du nœud $i$ à $j$ sous scénario $s$ ;

Puisque le coût de transport est $C_{iklj}^s = w_{ij}^s(\chi.d_{ik} + \alpha.d_{kl} + \delta.d_{lj})$ ; Le coût total du réseau est la somme de tous les $C_{iklj}^s$.

Le RUSApHMP sous l'incertitude des demandes avec des scénarios discrets est formulé en minimisant le pire des cas sur tous les scénarios comme suit :

$$\min \max_{s \in S} \sum_{s \in S} q_s \left[ \sum_{i \in N} \sum_{k \in N} d_{ik} Z_{ik} (\chi O_i^s + \delta D_i^s) + \sum_{i \in N} \sum_{k \in N} \sum_{l \in N} \alpha d_{kl} Y_{kl}^{is} \right] \tag{5.1}$$

Sous les contraintes :

$$\sum_{k \in N} Z_{kk} = p \tag{5.2}$$

$$\sum_{k \in N} Z_{ik} = 1 \qquad i \in N \tag{5.3}$$

$$Z_{ik} \leq Z_{kk} \qquad i, k \in N \tag{5.4}$$

$$\sum_{l \in N} Y_{kl}^{is} - \sum_{l \in N} Y_{lk}^{is} = O_i^s Z_{ik} - \sum_{j \in N} w_{ij}^s Z_{ik} \qquad i, k \in N,\ s \in S \tag{5.5}$$

$$\sum_{l \in N, l \neq k} Y_{kl}^{is} \leq O_i^s Z_{ik} \quad i, k \in N,\ s \in S \tag{5.6}$$

$$Z_{ik} \in \{0, 1\} \qquad i, k \in N \tag{5.7}$$

$$Y_{kl}^{is} \geq 0 \qquad i, l, k \in N,\ s \in S \tag{5.8}$$

La fonction objectif (5.1) minimise le coût dans les pires des cas ; elle a donné des résultats mitigés de transport totaux prévus de la collecte, du transfert et de la

distribution sous tous les scénarios. La première partie de la fonction objectif à l'intérieur de la parenthèse sert à minimiser les coûts de transport entre les hubs et les non-hubs et la seconde minimise les coûts de transport entre hubs. Notons qu'il y a $|S|$ termes dans la fonction objectif dans laquelle chacun a le poids de $qs$ et minimise le coût de transport de ce scénario. La contrainte (5.2) garantit la localisation exacte de p hubs. La contrainte (5.3) impose que chaque non-hub soit affecté à un seul hub. La contrainte (5.4) garantit que les hubs sont établis pour chaque distribution / collecte, empêchant ainsi la transmission directe entre les nœuds non-hubs. La contrainte (5.5) est la contrainte de conservation du flux sous chaque scénario.

## 5.5 Génération de la solution initiale

L'ensemble des emplacements initiaux des hubs est très important pour la performance des algorithmes évolutifs et multi-start. Les solutions générées doivent posséder une qualité et diversité afin d'accélérer la convergence des algorithmes. Le GA génère une solution initiale appropriée à tous les scénarios et à partir de cette solution un ensemble de populations seront générées.

La création de la solution initiale consiste à comparer entre deux critères afin d'utiliser le meilleur entre eux. Le premier consiste à assigner une probabilité à chaque nœud, puis en fonction des poids, nous sélectionnons l'emplacement initial des hubs les plus fréquemment visités.

Ce poids noté $\varphi(k)$, indique la probabilité que le nœud $i$ devienne un hub. Cela dépend du flux total entrant et sortant au nœud $k$ et de sa localisation spatiale dans le réseau [69]. Nous avons calculé $\varphi(k)$ pour l'ensemble des scénarios afin de cumuler les poids qu'un nœud $k$ soit un hub, les $p$ nœuds premiers avec les grands poids dans la liste $\varphi(k)$ seront sélectionnés comme des hubs dans la solution initiale.

$$\varphi(k) = \frac{\sum_s (O_{ks} + D_{ks})}{\sum_s \sum_k (O_{ks} + D_{ks})} * \left(1 - \frac{\sum_s \sum_j (d^s_{kj} + d^s_{kj})}{\sum_s \sum_k \sum_j (d^s_{kj} + d^s_{kj})}\right).$$

Le deuxième critère, est similaire à celui utilisé pour le CSApHMP [19]. Nous avons observé que la solution de chaque scénario individuel peut fournir une approximation assez proche à la solution optimale du CSApHMP. Nous avons proposé $h(k) = \frac{1}{G_k} \sum_j d_{kj} W_{kj}$ (avec $G_k$ est la capacité du nœud k), comme critère pour partitionner l'ensemble des nœuds en $p$ groupes, où chaque groupe est susceptible de contenir un hub dans une solution optimale.

De même dans cette étude la liste $h(k)$ se base sur le critère défini comme suit :

$$h(k) = \sum_{j} d_{kj} W^{s}_{kj}$$

pour partitionner l'ensemble des nœuds en $p$ groupes, où chaque groupe est susceptible de contenir un hub dans une solution optimale dans les scénarios. Le nœud $k$ avec le plus grand $(O^s_k + D^s_k)$ pour $1 \leq s \leq |S|$ dans son groupe est choisi comme hub de la solutions initiale basée sur le critère $h(k)$.

Ces deux critères permettent de créer deux solutions initiales l'une à partir du critère $\varphi(k)$ et l'autre à partir du critère $h(k)$. Nous sélectionnons celle avec le coût minimal dans le pire des scénarios proposés. Des tests sur un ensemble de benchmarks de la littérature, ont montré que ces deux critères sont les meilleurs pour sélectionner les p hubs susceptibles d'être des hubs de la solution robuste.

## 5.6 Algorithme génétique

Ayant générer une solution initiale, chaque scénario sera résolu comme un USApHMP comme c'est déjà expliqué dans la section 4.3. La seule différence réside dans la définition de la fitness.

*Fonction d'évaluation (Fitness)* : Soit $X$ l'ensemble des individus dans notre implémentation ($|X| = R * t$), Chaque individu $x \in X$ sera évaluer pour chaque scénario $s \in S$ en utilisant la fonction objectif

$$fitness(x, s) \;=\; q_s \left[ \sum_{i \in N} \sum_{k \in N} d_{ik} Z_{ik} (\beta O^s_i + \chi D^s_i) + \sum_{i \in N} \sum_{k \in N} \sum_{l \in N} \alpha d_{kl} Y^{is}_{kl} \right]$$

sous les contraintes (5.2)-(5.8).

Soit $SolR$ est la solution robustesse au pire des cas :
$SolR \;=\; \min_{x \in X} \max_{s \in S} \{fitness(x, s)\}$.

## 5.7 Implémentation du GA pour le RUSApHMP

Le nombre de solutions à traiter et très important pour trouver la meilleure solution robuste pour cela notre GA se base sur le traitement d'un nombre important des individus en parallèle. Comme nous l'avons déjà expliqué précédemment notre GA génère $R$ sous-populations chacune de taille $t$. Le nombre d'itérations dans la boucle interne (resp la boucle externe) est $N1$ (resp. $N2$). Ainsi, $N1 * N2$ est le nombre de générations et $R * t * N1 * N2$ est le total des individus évalué.

Nous configurons le GPU comme une grille de $R$ blocs ($bloc_0, ..., bloc_{R-1}$) chacun de $t$ threads. Le GA part de la solution initiale $P_0$ et le duplique $R$ fois dans les mémoires partagées des $R$ blocs. Ensuite, chaque bloc $i$, $0 \leq i < R$, génère une sous-population avec $t$ individus en appliquant $t$ fois 1-$exchange(P_0)$ et stocke dans sa mémoire partagée les $t$ individus générés plus la liste $L$.

Les threads $T_0^i, ..., T_{t-1}^i$ du bloc $i$ génère individuellement la sous-population $p_0^i, ..., p_{t-1}^i$ en appliquant 1-$exchange(P_i)$ (initialement $P_i = P_0$). Ensuite chaque thread $T_{2j}^i$ fait le croisement des deux parents $p_{2j}^i$ et $p_{2j+1}^i$ pour générer deux enfants, $ch1$ et $ch2$ puis $T_{2j}^i$ corrige $ch1$ et $T_{2j+1}^i$ corrige $ch2$ qui permet de produire des individus réalisables $c_{2j}^i$ et $c_{2j+1}^i$.

Il ressort après le croisement de deux parents que dans chaque bloc $i$, $0 \leq i < R$ une population de $2t$ individus ($t$ parents et $t$ enfants) représenter par $\{p_j^i, c_j^i\}$, avec $0 \leq j < t$, chaque deux individus (parent et fils) sont évalués par le thread $T_j^i$, pour obtenir $f_{m,k}^i = fitness(x_j^i, k)$, avec $x_j^i \in \{p_j^i, c_j^i\}$,$0 \leq j < t$, $0 \leq m < 2t$ et $0 \leq k < |S|$.

Les coûts des solutions de la population du bloc $i$ sont représentés sous forme d'une matrice $f_{m,k}^i$ de $2t$ lignes pour indiquer les numéros des solutions et de $|S|$ colonnes pour indiquer les numéros des scénarios. Pour trouver la solution robuste de la population $i$ on a $SolR_i = \min_{0 \leq m < 2t} \max_{0 \leq k < |S|} \{f_{m,k}^i\}$.

Après avoir les $R$ solutions robustes locales des $R$ populations ($SolR_i, 0 \leq i < R$), on va les copier dans la mémoire globale pour que le thread maître ($thread_0$ du $bloc_0$) puisse sélectionner la solution robuste finale parmi les solutions robustes des populations proposées :

$$Sol_{Robuste} = \min_{0 \leq i < R} \max_{0 \leq k < |S|} \{fitness(SolR_i, k)\}$$

cette solution est considérée comme une solution robuste finale du problème ou comme nouvelle solution initiale pour l'itération suivante de la boucle externe pour

créer les nouvelles populations. Le processus est répété $N2$ fois, comme il peut s'arrêter s'il n'y a pas d'amélioration de la solution robuste finale pendant 20 itérations.

Le pseudo-code CUDA pour cette implémentation est le suivant :

1. Copier la matrice de distance $(d_{ij})$ et les $|S|$ matrices des flux $(w_{ij}^s)$ sur la mémoire globale du GPU.
2. Générer la solution initiale.
3. Définir les blocs et la grille comme suit :
   $dim3\ dimBlock(t, 1)$ ; $dim3\ dimGrid(R, 1)$ ;
4. Exécuter en parallèle le kernel GA :
   $GA <<< dimGrid, dimBlock >>> ()$ ;

## 5.8 Résultats expérimentaux

Toutes les expériences ont été réalisées sur la carte graphique Nvidia Quadro 2200 avec 2 Go avec 384 cœurs. L'implémentation du GA sur GPU a été codée avec CUDA C et CPLEX 12.6, et a été utilisée pour déterminer les bornes inférieures pour les instances du RUSApHMP considérées.

Trois types d'instances : AP, Urand et large Urand ont été utilisées.

Ces benchmarks incluent diverses instances de tests pour les problèmes déterministes (HLP). Ils englobent juste un scénario pour chaque paramètre (une seule matrice $w_{ij}$). Par conséquent, nous avons généré des scénarios supplémentaires en multipliant un nombre aléatoire tiré uniformément de [0.5, 1.5] par les flux des instances AP et Urand. En plus de cela, nous avons utilisé les instances que nous avons proposées dans la variante USApHMP jusqu'à l'instance de 4000 nœuds. Nous avons aussi adapté ces larges instances pour le problème robuste.

À notre connaissance, il s'agit de la première étude qui a mis en œuvre une implémentation parallèle sur GPU pour un problème de localisation des hubs avec des demandes incertaines. Par conséquent, aucune comparaison avec d'autres études de littérature ne nous a été possible. Cependant, les résultats obtenus sont comparés avec ceux du solveur CPLEX, (l'un des solveurs commerciaux les plus efficaces). Chaque instance de test est constituée de cinq scénarios $|S| = 5$, chacun avec une probabilité de 0.2.

Gap est l'erreur relative de la meilleure solution trouvée par l'implémentation à partir de la meilleure solution connue :

$$Gap\% \ = \ \frac{Sol_{GPU} - Sol_{LB}}{Sol_{GPU}}.$$

$Sol_{LB}$ désigne la meilleure solution possible pour le problème (la borne inférieure ou la solution optimale) et $Sol_{GPU}$ est le coût de la meilleure solution obtenue par l'implémentation sur GPU.

Chaque instance a été exécuté 10 fois indépendamment et les principaux résultats sont résumés dans les tableaux. 5.1-5.4. On garde les notations déjà utilisées dans les chapitres précédents ; $L.B$ la borne inférieure, $Gap\%$ l'erreur relative de la meilleure solution trouvée par CPLEX ou par l'implémentation du GPU par rapport à la borne inférieure, $IS\ Gap\%$ l'erreur relative de la solution initiale trouvée par rapport à la meilleure solution trouvée et $Opt.run$ le nombre de lancements du

programme dans lesquels la meilleure solution est trouvée.

Le tableau. 5.1 reporte les résultats des instances AP comparés à ceux obtenus par le CPLEX. Notre implémentation trouve la meilleure solution robuste avec un $Gap = 0\%$ pour toutes les instances ($\leq 40$) pour les 10 fois de lancement. Cela signifie que la borne inférieure obtenue par notre implémentation sur GPU est identique à la borne inférieure finale trouvée par le CPLEX. Le coût présenté dans ce tableau est le coût de la solution robuste pour chaque scénario. À noter que pour les instances ($> 40$ nœuds) on n'a pas pu avoir des solutions avec le solveur CPLEX.

Tableau 5.1 – Résultat du CPLEX et du GPU pour les instances AP.

| Inst. | n | p | CPLEX | | | GA sur GPU | | |
|---|---|---|---|---|---|---|---|---|
| | | | L.B | Temps CPU(s) | Gap% | Gap% | Temps GPU(s) | Opt.run |
| 1 | 10 | 2 | 172204.14 | 0.63 | 0.00 | 0.00 | 0.38 | 10 |
| 2 | | 3 | 141403.01 | 0.80 | 0.00 | 0.00 | 0.38 | 10 |
| 3 | | 4 | 116583.44 | 1.40 | 0.00 | 0.00 | 0.38 | 10 |
| 4 | | 5 | 91468.85 | 0.90 | 0.00 | 0.00 | 0.39 | 10 |
| 5 | 20 | 2 | 175185.19 | 5.18 | 0.00 | 0.00 | 0.41 | 10 |
| 6 | | 3 | 154606.01 | 15.55 | 0.00 | 0.00 | 0.41 | 10 |
| 7 | | 4 | 137954.39 | 19.82 | 0.00 | 0.00 | 0.42 | 10 |
| 8 | | 5 | 125190.99 | 23.50 | 0.00 | 0.00 | 0.42 | 10 |
| 9 | 25 | 2 | 176499.82 | 27.72 | 0.00 | 0.00 | 0.43 | 10 |
| 10 | | 3 | 156159.87 | 28.89 | 0.00 | 0.00 | 0.43 | 10 |
| 11 | | 4 | 139940.42 | 55.92 | 0.00 | 0.00 | 0.43 | 10 |
| 12 | | 5 | 124435.16 | 52.01 | 0.00 | 0.00 | 0.43 | 10 |
| 13 | 40 | 2 | 180697.70 | 402.40 | 0.00 | 0.00 | 0.51 | 10 |
| 14 | | 3 | 161737.66 | 1276.56 | 0.00 | 0.00 | 0.51 | 10 |
| 15 | | 4 | 147314.03 | 1090.30 | 0.00 | 0.00 | 0.52 | 10 |
| 16 | | 5 | 137697.46 | 1689.81 | 0.00 | 0.00 | 0.53 | 10 |

Le tableau. 5.2 présente les résultats obtenus par notre implémentation sur GPU pour les instances AP entre 50 et 400 nœuds. On y a reporte les coûts des solutions robustes pour chaque scénario et aussi le Gap entre la solution initiale et la solution robuste (Colonne IS Gap). Nous pouvons constater que notre implémentation peut résoudre des instances AP de 400 nœuds en temps $< 31s$.

Tableau 5.2 – Résultats des instances AP.

| n | p | Sol GPU des Scénarios | | | | | IS Gap% | Temps GPU(s) |
|---|---|---|---|---|---|---|---|---|
| | | $S_1$ | $S_2$ | $S_3$ | $S_4$ | $S_5$ | | |
| 50 | 2 | 175508.10 | 179184.37 | 178899.02 | 178488.74 | 178810.39 | 0.18 | 0.61 |
| | 3 | 155365.81 | 158681.28 | 159089.64 | 158897.54 | 158760.83 | 0.26 | 0.67 |
| | 4 | 140573.74 | 143564.87 | 143633.55 | 143301.25 | 143824.67 | 0.19 | 0.72 |
| | 5 | 130155.10 | 132693.03 | 133092.45 | 133526.46 | 133173.29 | 0.19 | 0.84 |
| 100 | 5 | 136858.96 | 136939.84 | 136147.00 | 137275.90 | 137301.63 | 0.08 | 4.18 |
| | 10 | 106594.49 | 106263.54 | 105881.01 | 106801.13 | 106830.33 | 0.15 | 4.25 |
| | 15 | 90239.81 | 90471.28 | 90029.46 | 90517.29 | 90720.88 | 0.13 | 5.32 |
| | 20 | 79996.55 | 79931.77 | 79793.12 | 80502.46 | 80585.30 | 0.16 | 5.94 |
| 200 | 5 | 139476.26 | 141362.21 | 139627.68 | 139234.33 | 139869.77 | 0.10 | 12.47 |
| | 10 | 109828.93 | 110965.30 | 110074.10 | 109621.02 | 110136.46 | 0.18 | 13.21 |
| | 15 | 94371.88 | 95415.13 | 94402.07 | 94096.02 | 94579.03 | 0.15 | 13.88 |
| | 20 | 85353.69 | 86033.97 | 85335.08 | 85151.08 | 85525.21 | 0.19 | 14.39 |
| 300 | 5 | 190646.45 | 186974.51 | 187035.30 | 187014.48 | 185503.38 | 0.08 | 19.71 |
| | 10 | 141590.88 | 139358.39 | 139141.51 | 138921.60 | 138217.60 | 0.18 | 21.85 |
| | 15 | 123346.96 | 117525.20 | 117607.28 | 117663.93 | 116719.79 | 0.10 | 24.16 |
| | 20 | 110803.94 | 104933.54 | 105023.65 | 105083.58 | 104326.87 | 0.16 | 25.74 |
| 400 | 5 | 179574.93 | 178160.17 | 177761.14 | 178192.89 | 177522.40 | 0.13 | 24.10 |
| | 10 | 139381.11 | 137479.75 | 137266.98 | 137683.47 | 137224.02 | 0.13 | 26.86 |
| | 15 | 120059.66 | 119770.72 | 119494.92 | 119956.90 | 119518.19 | 0.14 | 28.14 |
| | 20 | 108544.19 | 106167.78 | 105926.72 | 106391.67 | 105910.45 | 0.19 | 30.49 |

Le tableau. 5.3 reporte les résultats des instances Urand générées aléatoirement par Ilić et al. [81] jusqu'à 1000 nœuds. Le tableau. 5.4 présente les résultats des larges instances Urand proposées dans [22]. Ce sont des instances de 1500, 2000, 3000 et 4000 nœuds.

Tableau 5.3 – Résultats des instances Urand.

| n | p | Sol GPU des Scénarios | | | | | IS Gap% | Temps GPU(s) |
|---|---|---|---|---|---|---|---|---|
| | | $S_1$ | $S_2$ | $S_3$ | $S_4$ | $S_5$ | | |
| 100 | 10 | 26999.37 | 27117.19 | 27147.45 | 26961.75 | 27084.12 | 0.14 | 4.33 |
| | 15 | 25265.69 | 25398.33 | 25432.83 | 25243.19 | 25389.42 | 0.11 | 5.71 |
| | 20 | 24326.16 | 24468.39 | 24479.75 | 24327.40 | 24463.99 | 0.08 | 6.14 |
| 200 | 10 | 112911.01 | 112987.22 | 112714.50 | 113217.66 | 113128.35 | 0.16 | 13.71 |
| | 15 | 106047.61 | 106158.36 | 105887.71 | 106377.09 | 106271.80 | 0.17 | 14.10 |
| | 20 | 102278.26 | 102373.20 | 102110.79 | 102567.03 | 102499.67 | 0.14 | 14.86 |
| 300 | 10 | 253702.54 | 254418.52 | 254177.72 | 253965.56 | 254029.47 | 0.09 | 22.11 |
| | 15 | 238185.41 | 238941.44 | 238694.71 | 238503.02 | 238565.94 | 0.13 | 25.01 |
| | 20 | 229302.86 | 230017.54 | 229695.05 | 229594.06 | 229665.82 | 0.11 | 26.12 |
| 400 | 10 | 448570.44 | 449251.92 | 448477.05 | 448802.24 | 449195.06 | 0.18 | 32.19 |
| | 15 | 425492.09 | 426145.74 | 425442.34 | 425610.80 | 425982.76 | 0.16 | 34.62 |
| | 20 | 410343.00 | 411021.83 | 410309.08 | 410472.63 | 410877.02 | 0.13 | 37.55 |
| 1000 | 10 | 2823710.31 | 2824771.43 | 2824775.12 | 2826250.54 | 2823068.18 | 0.21 | 88.28 |
| | 15 | 2687765.37 | 2688960.29 | 2688646.11 | 2690321.05 | 2687047.57 | 0.16 | 87.65 |
| | 20 | 2589591.14 | 2590372.75 | 2590278.05 | 2591615.97 | 2588470.79 | 0.13 | 92.36 |

Tableau 5.4 – Résultats des instances Urand larges.

| n | p | Sol GPU des Scénarios | | | | | IS Gap% | Temps GPU(s) |
|---|---|---|---|---|---|---|---|---|
| | | $S_1$ | $S_2$ | $S_3$ | $S_4$ | $S_5$ | | |
| 1500 | 20 | 442881106 | 442658523 | 442781454 | 442793793 | 442838656 | 0.14 | 387.41 |
| | 30 | 385730802 | 385550054 | 385633132 | 385725502 | 385722688 | 0.19 | 391.54 |
| | 40 | 385382712 | 385183238 | 385286069 | 385369970 | 385354395 | 0.11 | 407.11 |
| | 50 | 369195715 | 369033140 | 369123800 | 369188889 | 369161402 | 0.10 | 424.67 |
| 2000 | 20 | 796719685 | 796876027 | 796886643 | 796918575 | 796681779 | 0.14 | 467.32 |
| | 30 | 735412205 | 735418411 | 735516681 | 735578988 | 735321691 | 0.13 | 471.58 |
| | 40 | 721628486 | 721708773 | 721766506 | 721852237 | 721676322 | 0.07 | 482.27 |
| | 50 | 681768360 | 681878014 | 681929079 | 681971889 | 681793901 | 0.07 | 495.49 |
| 3000 | 20 | 1783011657 | 1783159224 | 1783436424 | 1783396260 | 1783004035 | 0.16 | 551.10 |
| | 30 | 1641764989 | 1641842453 | 1642024797 | 1642125741 | 1641698657 | 0.11 | 564.77 |
| | 40 | 1557885484 | 1557963347 | 1558239571 | 1558182713 | 1557912578 | 0.08 | 571.97 |
| | 50 | 1477186653 | 1477256537 | 1477430297 | 1477542744 | 1477112236 | 0.10 | 593.20 |
| 4000 | 20 | 3204196655 | 3203761483 | 3203756544 | 3203792926 | 3203727777 | 0.15 | 631.17 |
| | 30 | 2945689727 | 2945370174 | 2945320893 | 2945349774 | 2945223575 | 0.16 | 649.78 |
| | 40 | 2840755728 | 2840400974 | 2840407531 | 2840335030 | 2840276418 | 0.10 | 667.93 |
| | 50 | 2663251465 | 2662922066 | 2663036819 | 2662989335 | 2662828529 | 0.10 | 675.47 |

## 5.9 Conclusion

Dans ce chapitre, nous avons adapté notre GA pour résoudre le problème de localisation de p-médian hub avec unique affectation sous l'incertitude des demandes avec des scénarios discrets. L'incertitude était associée aux demandes. La fonction objectif minimise les coûts totaux en se basant sur le modèle *min-max* dans chaque scénario (le pire des cas pour chaque scénario). Nous avons amélioré notre GA parallèle sur GPU pour qu'il s'adapte à ce cas de l'optimisation robuste. Cela nous a permet d'avoir la borne inférieure assez efficace en temps très inférieur a celui de CPLEX. La borne inférieure obtenue par notre implémentation sur GPU était identique à la borne inférieure finale trouvée par le CPLEX (pour les instances $\leq 40$ nœuds), tandis que les autres instances sont insolvables par le CPLEX. À partir des résultats expérimentaux obtenus, nous pouvons dire que notre implémentation sur GPU est bien plus performante que CPLEX en terme d'efficacité (temps de calcul) et a le potentiel de résoudre des instances plus importantes $\leq 4000$ nœuds (limitées par la capacité de la mémoire du GPU, mais pas par le temps d'exécution).

# Conclusion et Perspectives

Cette thèse est une contribution à la parallélisation et l'implémentation sur GPU d'applications réelles de grandes tailles. Nous nous sommes intéressés aux problèmes de la logistique de distribution, notamment les problèmes de tournées de véhicules et ceux de localisation des hubs. Ces deux problèmes d'optimisation combinatoire sont considérés NP-difficiles.

Les applications réelles de tels problèmes sont nombreuses, dès lors qu'il s'agit d'applications réelles, la taille des problèmes devient considérable. Dans ce cas, les problèmes ne peuvent pas être résolus à l'aide de systèmes classiques. Par conséquent, l'usage des architectures haute performance telles les GPUs devient nécessaire. Nous étions parmi les premiers à utiliser les GPUs pour ces problèmes.

Concrètement, nous avons implémenté des heuristiques sur GPU pour les deux problèmes cités précédemment et leurs variantes.Dans un premier temps, nous avons traité différentes variantes du VRP. Nous avons proposé quatre implémentation sur GPU pour le résoudre. La première implémentation sur GPU est une heuristique basée sur l'algorithme CW pour résoudre le VRP unique et multi dépôts. Les tests montrent que cette implémentation atteint un facteur d'accélération optimal.
La deuxième implémentation pour la variante multi capacités du VRP, a permit d'améliorer les solutions publiées dans la littérature et aussi de résoudre des instances non résolues jusqu'à présent.
La troisième implémentation est une méthode simple pour le VRP dynamique, qui insère en temps réel des requêtes dynamiques dans des routes déjà planifiées.
La quatrième implémentation basée sur un algorithme génétique permet de résoudre de grandes instances du DVRP (jusqu'à 3000 nœuds).
Les tests numériques montrent que les implémentations proposées exploitent au mieux le parallélisme et la puissance de calcul du GPU et permettent de résoudre des instances non résolues et d'améliorer les solutions proposées dans la littérature.

Ces travaux ont donné lieu à cinq publications [24, 23, 25, 16, 17].

Dans un second temps, le deuxième problème traité est celui de la localisation des hubs avec diverses variantes. L'approche de résolution est basée sur un algorithme génétique dont le codage, la solution initiale, les opérateurs génétiques et l'implémentation GPU sont adaptés à chaque variante traitée.
La première variante traitée du HLP est celle où le nombre des hubs à localiser est connu à l'avance. Aussi nous avons traité le problème où le nombre des hubs est une décision à prendre. Nous avons abordé également la variante où les hubs ont une capacité limitée et la variante avec multi affectations où chaque non-hub peut-être affecté à plusieurs hubs.
Les résultats expérimentaux obtenus sont comparés aux meilleures solutions connues sur des instances jusqu'à 1000 nœuds et sur des instances plus larges jusqu'à 6000 nœuds générées par nous-même. Ces résultats montrent que les implémentations GPU surpassent les heuristiques les plus connues en termes de qualité de la solution et en temps d'exécution. Aussi, elles ont permis de résoudre des instances non résolues jusqu'à présent. Nous pensons que les implémentations proposées ont le potentiel de résoudre des instances plus importantes (limitées par la capacité de la mémoire du GPU, mais pas par le temps d'exécution). Ces travaux ont donné lieu à quatre publications [22, 21, 19, 18]

Au cours des dernières années, les problèmes d'optimisation combinatoire ont été étendus pour gérer des données incertaines, ce qui donne lieu à l'optimisation robuste. Nous avons adapté notre GA parallèle sur GPU pour résoudre le RUSApHMP avec des demandes incertaines et des scénarios discrets.
Nous avons développé une approche de résolution basée sur le GA pour le RUSApHMP [20]. les tests numériques montrent la supériorité de notre approche par rapport à CPLEX.

Nous souhaitons approfondir cette recherche dans plusieurs directions :

- en résolvant d'autres variantes du HLP, surtout avec une optimisation robuste.
- en développant d'autres aspects en parallèle, comme un système de colonies de fourmis.
- en faisant un lien entre le solveur CPLEX et un cluster de GPUs afin de lancer les fonctions du solveur CPLEX comme des kernels sur un ensemble de GPUs pour accélérer le calcul du CPLEX.
- en utilisant les GPUs dans l'intelligence artificielle, pour résoudre le problème SLAM (Simultaneous localization and mapping).

# Bibliographie


[1] Abbatecola, L., Fanti, M. P., and Ukovich, W. A review of new approaches for dynamic vehicle routing problem. *In Automation Science and Engineering (CASE), 2016 IEEE International Conference* (2016), 361–366. 66

[2] Abdinnour-Helm, S. A hybrid heuristic for the uncapacitated hub location problem. *European Journal of Operational Research 106(2-3)* (1998), 489–499. 29

[3] Abdinnour-Helm, S. Using simulated annealing to solve the p-hub median problem. *International Journal of Physical Distribution and Logistics Management 31(3)* (2001), 203–220. 24

[4] Abdinnour-Helm, S., and Venkataramanan, M. A. Solution approaches to hub location problems. *Annals of Operations Research 78* (1998), 31–50. 28

[5] Abyazi-Sani, R., and Ghanbari, R. An efficient tabu search for solving the uncapacitated single allocation hub location problem. *Computers and Industrial Engineering 93* (2016), 99–109. 29, 105, 108

[6] AlBdaiwi, B. F., and AboElFotoh, H. M. A gpu-based genetic algorithm for the p-median problem. *The Journal of Supercomputing. 73(10)* (2017), 4221–4244. 98

[7] Alumur, S., and Kara, B. Y. Network hub location problems : The state of the art. *European journal of operational research 190(1)* (2008), 1–21. 24, 28

[8] Alumur, S. A., Nickel, S., and Saldanha-da Gama, F. Hub location under uncertainty. *Transportation Research Part B : Methodological 46(4)* (2012), 529–543. 136

[9] Anderson, J. A., Lorenz, C. D., and Travesset, A. General purpose molecular dynamics simulations fully implemented on graphics processing units. *Journal of Computational Physics 227(10)* (2008), 5342–5359. 45

[10] Arbelaez, A., and Codognet, P. A gpu implementation of parallel constraint-based local search. *In Parallel, Distributed and Network-Based Processing (PDP), 2014 22nd Euromicro International Conference IEEE* (2014), 648–655. 47

[11] Augerat, P., Belenguer, J. M., Benavent, E., Corbéran, A., and Naddef, D. Separating capacity constraints in the cvrp using tabu search. *European Journal of Operational Research 106(2-3)* (1998), 546–557. 21

[12] Aykin, T. Networking policies for hub-and-spoke systems with application to the air transportation system. *Transportation Science 29(3)* (1995), 201–221. 24, 28

[13] Beasley, J. E. Route first—cluster second methods for vehicle routing. *Omega 11(4)* (1983), 403–408. 23

[14] Bellman, R. Dynamic programming and the numerical solution of variational problems. *Operations Research* (1957), 277–288. 9

[15] Ben-Tal, A., El Ghaoui, L., and Nemirovski, A. *Robust optimization.* Princeton University Press, 2009. 135

[16] Benaini, A., and Berrajaa, A. Solving the dynamic vehicle routing problem on gpu. *Logistics Operations Management (GOL), 2016 3rd International Conference on IEEE* (2016), 1–6. 2, 21, 67, 73, 75, 149

[17] Benaini, A., and Berrajaa, A. Genetic algorithm for large dynamic vehicle routing problem on gpu. *In Logistics Operations Management (GOL), 2018 4th International Conference on IEEE* (2018), 1–9. 2, 80, 94, 149

[18] Benaini, A., and Berrajaa, A. Gpu based approach for solving the multiple allocation p-hub median problem. *Submitted to J. soft computing* (April 2017). 2, 124, 126, 149

[19] Benaini, A., and Berrajaa, A. Gpu based algorithm for the capacitated single allocation hub location problem. *Submitted to Computers and Industrial Engineering Journal* (February 2017). 2, 119, 139, 149

[20] Benaini, A., and Berrajaa, A. Genetic algorithm for the robust usaphmp with discrete scenarios. *working paper* (June 2018). 2, 137, 149

[21] Benaini, A., Berrajaa, A., Boukachour, J., and Oudani, M. Parallel genetic algorithm for the uncapacited single allocation hub location problem on gpu. *In Computer Systems and Applications (AICCSA), 2016 IEEE/ACS 13th International Conference of IEEE* (2016), 1–8. 2, 99, 149

[22] Benaini, A., Berrajaa, A., Boukachour, J., and Oudani, M. Solving the uncapacitated single allocation p-hub median problem on gpu. *Bioinspired Heuristics for Optimization. Studies in Computational Intelligence vol 774. Springer* (2019). 2, 99, 102, 106, 146, 149

[23] Benaini, A., Berrajaa, A., and Daoudi, E. M. Gpu implementation of the multi depot vehicle routing problem. *In Computer Systems and Applications (AICCSA), 2015 IEEE/ACS 12th International Conference of IEEE* (2015), 1–7. 2, 48, 149

[24] Benaini, A., Berrajaa, A., and Daoudi, E. M. Solving the vehicle routing problem on gpu. *In Proceedings of the Mediterranean Conference on Information and Communication Technologies 2015 Springer* (2016), 239–248. 2, 48, 51, 63, 149

[25] Benaini, A., Berrajaa, A., and Daoudi, E. M. Parallel implementation of the multi capacity vrp on gpu. *In Europe and MENA Cooperation Advances in Information and Communication Technologies Springer* (2017), 353–364. 2, 58, 61, 149

[26] Bertsimas, D., Brown, D. B., and Caramanis, C. Theory and applications of robust optimization. *SIAM review. 53(3)* (2011), 464–501. 135

[27] Boland, N., Krishnamoorthy, M., Ernst, A. T., and Ebery, J. Preprocessing and cutting for multiple allocation hub location problems. *European Journal of Operational Research 155(3)* (2004), 638–653. 26, 28, 29, 124

[28] Boukani, F. H., Moghaddam, B. F., and Pishvaee, M. S. Robust optimization approach to capacitated single and multiple allocation hub location problems. *Computational and Applied Mathematics 35(1)* (2016), 45–60. 136

[29] Campbell, J. F. Location and allocation for distribution systems with transshipments and transportion economies of scale. *Annals of operations research 40(1)* (1992), 77–99. 24

[30] Campbell, J. F., and O'Kelly, M. E. Twenty-five years of hub location research. *Transportation Science 46(2)* (2012), 153–169. 24

[31] Campeotto, F., Dovier, A., Fioretto, F., and Pontelli, E. A gpu implementation of large neighborhood search for solving constraint optimization problems. *In ECAI* (2014), 189–194. 47

[32] Candia-Véjar, A., Álvarez Miranda, E., and Maculan, N. Minmax regret combinatorial optimization problems : an algorithmic perspective. *RAIRO-Operations Research 45(2)* (2011), 101–129. 136

[33] Cardon, S., Dommers, S., Eksin, C., Sitters, R., Stougie, A., and Stougie, L. A ptas for the multiple depot vehicle routing problem. *SPOR Report 3* (2008). 57

[34] Catanzaro, B., Sundaram, N., and Keutzer, K. Fast support vector machine training and classification on graphics processors. *In Proceedings of the 25th international conference on Machine learning* (2008), 104–111. 44

[35] Cekmez, U., Ozsiginan, M., and Sahingoz, O. K. Adapting the ga approach to solve traveling salesman problems on cuda architecture. *In Computational Intelligence and Informatics (CINTI), 2013 IEEE 14th International Symposium IEEE* (2013), 423–428. 47

[36] Chen, J. F. A heuristic for the uncapacitated multiple allocation hub location problem. *Journal of the Chinese Institute of Industrial Engineers 23(5)* (2006), 371–381. 121

[37] Chen, J. F. A hybrid heuristic for the uncapacitated single allocation hub location problem. *Omega 35(2)* (2007), 211–220. 29, 93

[38] Chen, S., Chen, R., and Gao, J. A monarch butterfly optimization for the dynamic vehicle routing problem. *Algorithms 10(3)* (2017), 107. 80, 89, 93

[39] Cherif Khettaf, W. R., Rachid, M. H., Bloch, C., and Chatonnay, P. New notation and classification scheme for vehicle routing problems. *RAIRO Operations Research 49(1)* (2015), 161–194. 20

[40] Christofides, N., and Beasley, J. E. The period routing problem. *Networks 14(2)* (1984), 237–256. 92

[41] Clarke, G., and Wright, J. W. Scheduling of vehicles from a central depot to a number of delivery points. *Operations research 12(4)* (1964), 568–581. 22, 49

[42] Coates, A., Huval, B., Wang, T., Wu, D., Catanzaro, B., and Andrew, N. Deep learning with cots hpc systems. *In International Conference on Machine Learning* (2013), 1337–1345. 44

[43] Contreras, I., Diaz, J. A., and Fernández, E. Lagrangean relaxation for the capacitated hub location problem with single assignment. *OR spectrum 31(3)* (2009), 483–505. 28

[44] Corberán, ., Peiró, J., Campos, V., Glover, F., and Martí, R. Strategic oscillation for the capacitated hub location problem with modular links. *Journal of Heuristics. 22(2)* (2016), 221–244. 122

[45] Cordeau, J. F., Gendreau, M., Hertz, A., Laporte, G., and Sormany, J. S. New heuristics for the vehicle routing problem. *Logistics systems : design and optimization Springer, Boston, MA* (2005), 279–297. 20

[46] Cordeau, J. F., and Laporte, G. Tabu search heuristics for the vehicle routing problem. *Metaheuristic Optimization via Memory and Evolution Springer, Boston, MA* (2005), 145–163. 23

[47] Crainic, T. G., Gendreau, M., Hansen, P., and Mladenović, N. Cooperative parallel variable neighborhood search for the p-median. *Journal of Heuristics. 10(3)* (2004), 293–314. 98

[48] Crainic, T. G., and Semet, F. Recherche opérationnelle et transport de marchandises. *Centre de recherche sur les transports (CRT)= Centre for Research on Transportation* (2005). 18

[49] Dantzig, G. B., and Wolfe, P. Decomposition principle for linear programs. *Operations Research 8(1)* (1960), 101–111. 11

[50] Desrochers, M., and Verhoog, T. W. A matching based savings algorithm for the vehicle routing problem. *Cahiers du GERAD* (1989). 22

[51] Diego, F. J., Gomez, E. M., Ortega Mier, M., and Garcia Sanchez, A. Parallel cuda architecture for solving de vrp with aco. *In Industrial Engineering : Innovative Networks. Springer* (2012), 385–393. 47

[52] Drigo, M. The ant system optimization by a colony of cooperating agents. *IEEE Transactions on Systems, Man, and Cybernetics Part B 26(1)* (1996), 1–13. 14

[53] Duhamel, C., Lacomme, P., Quilliot, A., and Toussaint, H. A multi-start evolutionary local search for the two-dimensional loading capacitated vehicle routing problem. *Computers and Operations Research 38(3)* (2011), 617–640. 12

[54] Eilon, S., and Watson-Gandy, C. D. T. Distribution management. *mathematical modelling and practical analysis No. 658.80184 E5* (1971). 21

[55] Erken, M. Genetic algorithm based solution for the hub location problem. *ISBN : 978-1-61804-307-8*. 29, 100

[56] Ernst, A. T., and Krishnamoorthy, M. Efficient algorithms for the uncapacitated single allocation p-hub median problem. *Location science 4(3)* (1996), 139–154. 25, 28

[57] Ernst, A. T., and Krishnamoorthy, M. An exact solution approach based on shortest-paths for p-hub median problems. *INFORMS Journal on Computing 10(2)* (1998), 149–162. 124

[58] Ernst, A. T., and Krishnamoorthy, M. Solution algorithms for the capacitated single allocation hub location problem. *Annals of Operations Research 86* (1999), 141–159. 27, 28, 29, 111, 118, 122

[59] Farahani, R. Z., Hekmatfar, M., Arabani, A. B., and Nikbakhsh, E. A review of models, classification, solution techniques, and applications. *Computers & Industrial Engineering 64(4)* (2013), 1096–1109. 24

[60] Fischetti, M., Toth, P., and Vigo, D. A branch-and-bound algorithm for the capacitated vehicle routing problem on directed graphs. *Operations Research 42(5)* (1994), 846–859. 21

[61] Fisher, M. Vehicle routing. *Handbooks in operations research and management science. vol. 8* (1995), 1–33. 92

[62] Fisher, M. L., and Jaikumar, R. A generalized assignment heuristic for vehicle routing. *Networks 11(2)* (1981), 109–124. 23

[63] Fosin, J., Davidovic, D., and Caric, T. A gpu implementation of local search operators for symmetric travelling salesman problem. *Promet Traffic and Transportation 25(3)* (2013), 225–234. 47

[64] Frazzoli, E., and Bullo, F. Decentralized algorithms for vehicle routing in a stochastic time-varying environment. *In Decision and Control, 2004. CDC. 43rd IEEE Conference Vol. 4* (2004), 3357–3363. 16

[65] Garcia-López, F., Melián-Batista, B., Moreno-Pérez, J. A., and Moreno-Vega, J. M. Parallelization of the scatter search for the p-median problem. *Parallel Computing. 29(5)* (2003), 575–589. 98

[66] Garland, M., Le Grand, S., Nickolls, J., Anderson, J., Hardwick, J., Morton, S., and ... Volkov, V. Parallel computing experiences with cuda. *IEEE micro (4)* (2008), 13–27. 44

[67] Gendreau, M., Hertz, A., and Laporte, G. A tabu search heuristic for the vehicle routing problem. *Management science 40(10)* (1994), 1276–1290. 23

[68] Gendreau, M., and Potvin, J. Y. Issues in real-time fleet management. *Transportation Science. 38(4)* (2004), 397–398. 67

[69] Ghaderi, A., and Rahmaniani, R. Meta-heuristic solution approaches for robust single allocation p-hub median problem with stochastic demands and travel times. *The International Journal of Advanced Manufacturing Technology 82(9-12)* (2016), 1627–1647. 137, 139

[70] Ghaffari-Nasab, N., Ghazanfari, M., and Teimoury, E. Robust optimization approach to the design of hub-and-spoke networks. *The International Journal of Advanced Manufacturing Technology 76(5-8)* (2015), 1091–1110. 136

[71] Gillett, B. E., and Miller, L. R. A heuristic algorithm for the vehicle-dispatch problem. *Operations research 22(2)* (1974), 340–349. 22

[72] Giosa, I. D., Tansini, I. L., and Viera, I. O. New assignment algorithms for the multi-depot vehicle routing problem. *Journal of the operational research society 53(9)* (2002), 977–984. 57

[73] Glover, F. Tabu search and adaptive memory programming advances, applications and challenges. *In Interfaces in computer science and operations research Springer, Boston, MA* (1997), 1–75. 13

[74] Golden, B., Assad, A., Levy, L., and Gheysens, F. The vehicle routing problem : latest advances and new challenges. *Computers and Operations Research 11(1)* (1984), 49–66. 64

[75] Golden, B. L., Raghavan, S., and Wasil, E. A. The vehicle routing problem : latest advances and new challenges. *The vehicle routing problem, Society for Industrial and Applied Mathematics Vol. 43* (2008). 21

[76] Gutiérrez-Jarpa, G., Desaulniers, G.and Laporte, G., and Marianov, V. A branch-and-price algorithm for the vehicle routing problem with deliveries, selective pickups and time windows. *European Journal of Operational Research 206(2)* (2010), 341–349. 21

[77] Hall, R. W. Vehicle routing software survey. *Retrieved August 16, 2013* (2013). 21

[78] Hanshar, F. T., and Ombuki-Berman, B. M. Dynamic vehicle routing using genetic algorithms. *Applied Intelligence 27(1)* (2007), 89–99. 67, 80, 81, 89, 92

[79] Hemmelmayr, V. C., Cordeau, J. F., and Crainic, T. G. An adaptive large neighborhood search heuristic for two-echelon vehicle routing problems arising in city logistics. *Computers and operations research 39(12)* (2012), 3215–3228. 23

[80] Holland, J. Adaption in natural and artificial systems. *The University of Michigan Press, Ann Arbor, MI* (1975). 15

[81] Ilić, A., Urošević, D., Brimberg, J., and Mladenović, N. A general variable neighborhood search for solving the uncapacitated single allocation p-hub median problem. *European Journal of Operational Research 206(2)* (2010), 289–300. 26, 29, 104, 105, 106, 107, 109, 129, 130, 131, 132, 146

[82] Juan, A. A., Faulin, J., Jorba, J., Riera, D., Masip, D., and Barrios, B. On the use of monte carlo simulation, cache and splitting techniques to improve the clarke and wright savings heuristics. *Journal of the Operational Research Society 62(6)* (2011), 1085–1097. 48

[83] Kalai, R., Lamboray, C., and Vanderpooten, D. Lexicographic p-robustness : An alternative to min–max criteria. *European Journal of Operational Research 220(3)* (2012), 722–728. 136

[84] Karagul, K. A new heuristic routing algorithm for fleet size and mix vehicle routing problem. *Gazi university Journal of Science 27(3)* (2014), 979–986. 20, 58, 59, 64

[85] Kirkpatrick, S., Gelatt, C. D., and Vecchi, M. P. Optimization by simulated annealing. *science 220(4598)* (1983), 671–680. 13

[86] Kratica, J. An electromagnetism-like metaheuristic for the uncapacitated multiple allocation p-hub median problem. *Computers and Industrial Engineering 66(4)* (2013), 1015–1024. 26

[87] Kratica, J., Milanović, M., Stanimirović, Z., and Tošić, D. An evolutionary-based approach for solving a capacitated hub location problem. *Applied Soft Computing 11(2)* (2011), 1858–1866. 99, 116

[88] Kratica, J., Stanimirovic, Z., Tosic, D., and Filipovic, V. Genetic algorithm for solving uncapacitated multiple allocation hub location problem. *Computing and Informatics 24(4)* (2012), 415–426. 29, 99, 124, 129, 131, 132

[89] Krizhevsky, A., Sutskever, I., and Hinton, G. E. Imagenet classification with deep convolutional neural networks. *In Advances in neural information processing systems.* (2012), 1097–1105. 44

[90] Kuby, M. J., and Gray, R. G. The hub network design problem with stopovers and feeders : The case of federal express. *Transportation Research Part A : Policy and Practice 27(1)* (1993), 1–12. 24

[91] Land, A. H., and Doig, A. G. An automatic method of solving discrete programming problems. *Econometrica : Journal of the Econometric Society* (1960), 497–520. 10

[92] Laporte, G. The vehicle routing problem : An overview of exact and approximate algorithms. *European journal of operational research 59(3)* (1992), 345–358. 21, 50

[93] Laporte, G., Gendreau, M., Potvin, J. Y., and Semet, F. Classical and modern heuristics for the vehicle routing problem. *International transactions in operational research 7(4-5)* (2000), 285–300. 20

[94] Li, J., Pardalos, P. M., Sun, H., Pei, J., and Zhang, Y. Iterated local search embedded adaptive neighborhood selection approach for the multi-depot vehicle routing problem with simultaneous deliveries and pickups. *Expert Systems with Applications 42(7)* (2015), 3551–3561. 23

[95] Li, J. M., Tan, H. S., Li, X., and Liu, L. L. A parallel simulated annealing solution for vrptw based on gpu acceleration. *In Advances in Intelligent Decision Technologies Springer* (2010), 201–208. 47

[96] Lim, G. J., and Ma, L. Gpu-based parallel vertex substitution algorithm for the p-median problem. *Computers and Industrial Engineering. 64(1)* (2013), 381–388. 98

[97] Makui, A., Rostami, M., Jahani, E., and Nikui, A. A multi-objective robust optimization model for the capacitated p-hub location problem under uncertainty. *Management Science Letters 2(2)* (2002), 525–534. 136

[98] Melab, N., and Talbi, E. G. Gpu-based island model for evolutionary algorithms. *In Proceedings of the 12th annual conference on Genetic and evolutionary computation. ACM* (2010), 1089–1096. 47

[99] Meyer, T., Ernst, A. T., and Krishnamoorthy, M. A 2-phase algorithm for solving the single allocation p-hub center problem. *Computers and Operations Research. 36(12)* (2009), 3143–3151. 104, 105, 118

[100] Michalakes, J., and Vachharajani, M. Gpu acceleration of numerical weather prediction. *Parallel Processing Letters 18(04)* (2012), 531–548. 45

[101] Mingozzi, A., and Valletta, A. An exact algorithm for period and multi-depot vehicle routing problems. *Department of Mathematics, University of Bologna, Bologna, Italy* (2003). 20

[102] Mladenovic, N., and Hansen, P. Variable neighborhood search. *Computers and operations research 24(11)* (1997), 1097–1100. 14

[103] Montemanni, R., Gambardella, L. M., Rizzoli, A. E., and Donati, A. V. A new algorithm for a dynamic vehicle routing problem based on ant colony system. *In Second international workshop on freight transportation and logistics. Vol. 1* (2003), 27–30. 80, 81, 88, 89, 92

[104] Naeem, M., and Ombuki-Berman, B. An efficient genetic algorithm for the uncapacitated single allocation hub location problem. *In Evolutionary Computation (CEC), 2010 IEEE Congress on IEEE.* (2010), 1–8. 84, 100, 116, 124, 127, 128

[105] Nemhauser, G. L., and Wolsey, L. A. Integer and combinatorial optimization. *Interscience series in discrete mathematics and optimization ed : John Wiley and Sons* (1988). 16

[106] Nickolls, J., and Kirk, D. *Graphics and computing GPUs.*, vol. Morgan Kaufmann. 2009. 37, 38

[107] Ochi, L. S., Vianna, D. S., Drummond, L. M., and Victor, A. O. A parallel evolutionary algorithm for the vehicle routing problem with heterogeneous fleet. *In International Parallel Processing Symposium. Springer, Berlin, Heidelberg* (1998), 216–224. 58, 59, 64

[108] Ogryczak, W. On the lexicographic minimax approach to location problems. *European Journal of Operational Research 100(3)* (1997), 566–585. 135

[109] O'kelly, M. E. A quadratic integer program for the location of interacting hub facilities. *European Journal of Operational Research 32(3)* (1987), 393–404. 28, 104

[110] Ombuki-Berman, B. M., Runka, A., and Hanshar, F. Waste collection vehicle routing problem with time windows using multi-objective genetic algorithms. *Proceedings of the Third IASTED International Conference on Computational Intelligence ACTA Press* (2007), 91–97. 24, 28

[111] Osman, I. Metastrategy simulated annealing and tabu search algorithms for the vehicle routing problem. *Annals of Operations Research 41* (1993), 421–451. 23

[112] Padberg, M., and Rinaldi, G. Optimization of a 532-city symmetric traveling salesman problem by branch and cut. *Operations Research Letters 6(1)* (1987), 1–7. 11

[113] Paessens, H. The savings algorithm for the vehicle routing problem. *European Journal of Operational Research 34(3)* (1988), 336–344. 22

[114] Park, N., Okano, H., and Imai, H. A path-exchange-type local search algorithm for vehicle routing and its efficient search strategy. *Journal of the Operations Research Society of Japan 43(1)* (2000), 197–208. 50

[115] Peker, M., Kara, B. Y., Campbell, J. F., and Alumur, S. A. Spatial analysis of single allocation hub location problems. *Networks and Spatial Economics 16(4)* (2016), 1075–1101. 112, 113

[116] Rahmaniani, R., Ghaderi, A., Mahmoudi, N., and Barzinepour, F. Stochastic p-robust uncapacitated multiple allocation p-hub location problem. *International Journal of Industrial and Systems Engineering 14(3)* (2013), 296–314. 137

[117] Ralphs, T. K. Parallel branch and cut for capacitated vehicle routing. *Parallel Computing 29(5)* (2003), 607–629. 21

[118] Randall, M. Solution approaches for the capacitated single allocation hub location problem using ant colony optimisation. *Computational Optimization and Applications 39(2)* (2008), 239–261. 29

[119] Rego, C., and Roucairol, C. Le problème de tournées de vehicules. *Étude et résolution approchée (Doctoral dissertation, INRIA)* (1994). 21

[120] Santos, L., Madeira, D., Clua, E., Martins, S., and Plastino, A. A parallel grasp resolution for a gpu architecture. *In International Conference on Metaheuristics and Nature Inspired Computing META10* (2010). 98

[121] Sariklis, D., and Powell, S. A heuristic method for the open vehicle routing problem. *Journal of the Operational Research Society 51(5)* (2000), 564–573. 20

[122] Sasaki, M., and Fukushima, M. On the hub-and-spoke model with arc capacity conatraints. *Journal of the Operations Research Society of Japan 46(4)* (2013), 409–428. 112

[123] Savelsbergh, M. A branch-and-price algorithm for the generalized assignment problem. *Operations research 45(6)* (1997), 831–841. 11

[124] Silva, M. R., and Cunha, C. B. New simple and efficient heuristics for the uncapacitated single allocation hub location problem. *Computers and Operations Research 36(12)* (2009), 3152–3165. 29, 105, 109, 110

[125] Solano-Charris, E., Prins, C., and Santos, A. C. Local search based metaheuristics for the robust vehicle routing problem with discrete scenarios. *Applied Soft Computing 32* (2015), 518–531. 19

[126] Solomon, M. M. Algorithms for the vehicle routing and scheduling problems with time window constraints. *Operations research 35(2)* (1987), 254–265. 89, 90

[127] Stanimirovic, Z. An efficient genetic algorithm for the uncapacitated multiple allocation p-hub median problem. *Control and Cybernetics 37(3)* (2008). 26, 29

[128] Stanimirovic, Z. A genetic algorithm approach for the capacitated single allocation p-hub median problem. *Computing and Informatics 29(1)* (2012), 117–132. 27, 99, 111, 113

[129] Subramanian, A., Uchoa, E., and Ochi, L. S. A hybrid algorithm for a class of vehicle routing problems. *Computers and Operations Research. 40(10)* (2013), 2519–2531. 94

[130] Szymon, J., and Dominik, Z. Solving multi-criteria vehicle routing problem by parallel tabu search on gpu. *Procedia Computer Science 18* (2013), 2529–2532. 47

[131] Taillard, . Parallel iterative search methods for vehicle routing problems. *Networks 23(8)* (1993), 661–673. 23, 92

[132] Talbi, E. G., and Hasle, G. Metaheuristics on gpus. *Journal Parallel Distrib. Comput. 73(1)* (2013), 1–3. 47, 63

[133] Tan, K. C., Lee, L. H., Zhu, Q. L., and Ou, K. Heuristic methods for vehicle routing problem with time windows. *Artificial intelligence in Engineering 15(3)* (2001), 281–295. 21

[134] Tas, D., Dellaert, N., Van Woensel, T., and De Kok, T. Vehicle routing problem with stochastic travel times including soft time windows and service costs. *Computers and Operations Research 40(1)* (2013), 214–224. 20

[135] Tasan, A. S., and Gen, M. A genetic algorithm based approach to vehicle routing problem with simultaneous pick-up and deliveries. *Computers and Industrial Engineering 62(3)* (2012), 755–761. 23

[136] Taylor, G. D., Harit, S., English, J. R., and Whicker, G. Hub and spoke networks in truckload trucking : Configuration, testing and operational concerns. *Logistics and Transportation Review 31(3)* (1995), 209. 24

[137] Thomas, A., and Resende, M. G. Set covering problem. *Operations research letters 8* (1989), 67–71. 13

[138] Topcuoglu, H., Corut, F., Ermis, M., and Yilmaz, G. Solving the uncapacitated hub location problem using genetic algorithms. *Computers and Operations Research 32(4)* (2005), 967–984. 29, 100

[139] Toth, P., and Vigo, D. Models, relaxations and exact approaches for the capacitated vehicle routing problem. *Discrete Applied Mathematics 123(1-3)* (2002), 487–512. 11

[140] Toth, P., and Vigo, D. The vehicle routing problem. *SIAM monographs on discrete mathematics and applications. Society for Industrial and Applied Mathematics* (2002). 20, 21, 22

[141] Toth, P., and Vigo, D. The granular tabu search and its application to the vehicle-routing problem. *Informs Journal on computing 15(4)* (2003), 333–346. 23

[142] Uchoa, E., Pecin, D., Pessoa, A., Poggi, M., Vidal, T., and Subramanian, A. New benchmark instances for the capacitated vehicle routing problem. *European Journal of Operational Research. 257(3)* (2017), 845–858. 80, 89, 93, 94

[143] Uthayopas, P., Srimool, G., Pichitlamken, J., and Benjamas, N. Speeding up the pickup and delivery problem with time windows using gpu cluster. *International Journal of Engineering and Industries 4(2)* (2013), 53. 47

[144] Van Luong, T., Melab, N., and Talbi, E. G. Gpu computing for parallel local search metaheuristic algorithms. *IEEE transactions on computers. 62(1)* (2013), 173–185. 97, 99

[145] Vidal, T., Crainic, T. G., Gendreau, M., and Prins, C. A unified solution framework for multi-attribute vehicle routing problems. *European Journal of Operational Research 234(3)* (2014), 658–673. 20

[146] Vidal, T., Crainic, T. G., Gendreau, M., and Prins, C. A unified solution framework for multi-attribute vehicle routing problems. *European Journal of Operational Research 234(3)* (2014), 658–673. 94

[147] Volkov, V., and Demmel, J. W. Benchmarking gpus to tune dense linear algebra. *In High Performance Computing, Networking, Storage and Analysis, 2008. SC 2008. International Conference. IEEE* (2008), 1–11. 42

[148] Wald, A. Contributions to the theory of statistical estimation and testing hypotheses. *The Annals of Mathematical Statistics 10(4)* (1939), 299–326. 135

[149] Xu, Y., Wang, L., and Yang, Y. Dynamic vehicle routing using an improved variable neighborhood search algorithm. *Journal of Applied Mathematics* (2013). 80, 81, 88, 89, 90, 91

[150] Zetina, C. A., Contreras, I., Cordeau, J. F., and Nikbakhsh, E. Robust uncapacitated hub location. *Transportation Research Part B : Methodological 106* (2017), 393–410. 136