Projet de mémoire pour l'obtention du titre de

Docteur en Systèmes Informatiques et Automatiques

de l'Ecole Doctorale EDSYS

Université Toulouse 3 Paul Sabatier

Présenté par :

Mohamed Esseghir LALAMI

# Titre :

Contribution à la résolution de problèmes d'optimisation combinatoire: méthodes heuristiques et parallèles

# Table des matières

# Table des figures

# Liste des tableaux

# Chapitre I
## Introduction générale

Le problème du sac à dos fait partie des problèmes d'optimisation combinatoire les plus étudiés ces cinquante dernières années, en raison de ces nombreuses applications dans le monde réel. En effet, ce problème intervient souvent comme sous-problème à résoudre dans plusieurs domaines : la logistique comme le chargement d'avions ou de bateaux, l'économie comme la gestion de portefeuille ou dans l'industrie comme la découpe de matériaux.

Ce problème en variables 0-1, dont l'énoncé est assez simple, fait partie des problèmes mathématiques NP-complets. Cela explique que le nombre d'ouvrages qui lui sont consacrés est important, on peut notamment citer les ouvrages de référence [KEL 04] et [MAR 90], mais aussi les différents travaux proposant diverses méthodes pour résoudre ce problème (cf [BEL 57], [GIL 65], [KOL 67], [BAL 80], [PLA 85] , [ELK 02] , [MEL 05] , [HIF 08] et [BEL 08a]).

De nos jours le problème du sac à dos se résout de manière assez efficace. Les travaux actuels portent sur différentes variantes du problème du sac à dos qui sont beaucoup plus difficiles à résoudre. On peut en citer :

- Le problème du sac à dos multidimensionnel (MKP) : on notera les travaux de Freville et Plateau [FRE 94], Hanafi et al. [HAN 96] et Boyer et al.[BOY 10].
- Le problème du partage équitable (KSP) : on notera les travaux de Hifi et al. [HIF 05] Belgacem et Hifi [BEL 08b], Boyer et al. [BOY 11].
- Le problème du sac à dos multiple (MKP) : on notera les travaux de Hung et Fisk [HUN 78], Martello et Toth [MAR 80].

- Le problème du sac à dos disjonctif (DCKP) : on notera les travaux de Yamada et Kataoka [YAM 01], Hifi et Michrafy [HIF 07].
- Le problème de bin packing: on notera les travaux de Bekrar et al.[BEK 10] et Hifi et al. [HIF 10].

Les approches proposées dans la littérature, pour résoudre les problèmes de la famille du sac à dos sont des méthodes exactes capables de résoudre un problème à l'optimalité ou des heuristiques qui fournissent une solution approchée de bonne qualité dans des temps de résolution très raisonnables.

Les méthodes classiques telles que la programmation dynamique et le Branch and Bound peuvent être combinées de manière efficace afin de donner naissance à des méthodes coopératives ou hybrides. On note par exemple le travail de Viader [VIA 98] sur une méthode coopérative pour le problème du sac à dos. On note aussi le travail de Boyer et al. [BOY 10] sur une méthode coopérative pour le problème du sac à dos multidimensionnel. Les tests numériques présentés montrent l'efficacité des méthodes coopératives par rapport aux méthodes classiques.

Le parallélisme constitue une autre approche afin d'accélérer la résolution de problèmes d'optimisation combinatoire. L'apparition de nouvelles architectures comme les Graphics Processing Units ou GPUs semble particulièrement intéressante afin de diminuer les temps de résolution de manière économique. cf. [LUO 10], [BOY 11].
On s'est intéressé dans ce mémoire à la résolution d'une variante du problème du sac à dos à savoir le problème du sac à dos multiple (MKP). Ce problème compte plusieurs applications industrielles dont on peut citer quelques exemples : le chargement de fret sur les navires, où il s'agit de choisir certains conteneurs, dans un ensemble de $n$ conteneurs à charger dans $m$ navires de différentes capacités de chargement (cf [EIL 71]), le chargement de $n$ réservoirs par $m$ liquides qui ne peuvent pas être mélangés (cf [MAR 80]); l'affectation de tâches. Le problème MKP est NP-complet et la nécessité de trouver des algorithmes donnant une bonne solution heuristique se justifie par la complexité de ce type de problème.

La deuxième partie de notre contribution porte sur l'utilisation des GPUs pour la mise en œuvre parallèle de méthodes d'optimisation combinatoire en variables 0-1. Ces travaux font suite à une série d'études effectuées dans l'équipe CDA du LAAS-CNRS sur la mise en

œuvre parallèle de la méthode de programmation dynamique sur GPU, cf. Boyer et al. [BOY 11] et [BOY 10]. Notre travail, s'est concentré sur la mise en œuvre parallèle sur GPU de la méthode de Branch and Bound ainsi que de la méthode du Simplexe.

## Organisation de la thèse

Dans le **chapitre II**, nous commençons par présenter le problème du sac à dos ainsi que certaines de ces variantes comme le problème du sac à dos multiple. Nous nous intéressons en particulier à des méthodes de résolution classiques, comme la programmation dynamique et le Branch and Bound.

Nous proposons au **chapitre III** une méthode heuristique pour résoudre le problème du sac à dos multiple. Nous commençons d'abord par un état de l'art du MKP et détaillons en particulier une heuristique qui à été proposée pour le problème du sac à dos multiple, à savoir l'heuristique MTHM de Martello et Toth [MAR 81]. Nous présentons en détail notre contribution à savoir l'heuristique RCH pour Recursive Core Heuristic. Dans cette dernière méthode, nous considérons le problème du sac à dos multiple comme une succession de problème de sac à dos à résoudre. Nous définissons alors pour chaque problème KP un noyau. Nous résolvons alors pour chaque noyau excepté le dernier, un problème de *subset sum* par l'approche basée sur la programmation dynamique proposée par Elkihel [ELK 84] tandis que le dernier noyau est résolu en utilisant la programmation dynamique classique.

Le **chapitre IV** est consacré au GPU. Nous commençons d'abord par donner un état de l'art du calcul sur GPU. Puis, nous nous intéresserons à l'architecture CUDA (Compute Unified Device Architecture) proposée par NVIDIA. Les performances de cette architecture reposent sur deux éléments fondamentaux : la mémoire et la décomposition du travail en tâches.

Au **chapitre V**, nous présentons l'approche que nous avons suivie pour la mise en œuvre parallèle de l'algorithme de Branch and Bound sur GPU. Nous donnons un bref état de l'art relatant les différentes implémentations parallèles existantes pour l'algorithme de Branch and Bound, qu'elles soient sur des machines multi-cœurs, grilles de calculs ou sur architecture GPU. Nous expliquons les différents choix que nous avons faits pour aboutir à la mise en œuvre proposée. Ceux-ci concernent tout aussi bien la stratégie de séparation des nœuds, les données sauvegardés pour chaque nœuds, les mémoires utilisées mais aussi les différentes

techniques et synchronisations utilisées pour diminuer les temps de latence d'accès en mémoire. Pour finir, nous présentons nos résultats et les analysons.

Enfin, nous présentons au **chapitre VI**, l'approche que nous avons suivie pour la mise en œuvre parallèle de la méthode du Simplexe sur GPU et sur un système Multi-GPU. Nous commençons d'abord par présenter un bref état de l'art. Nous expliquons ensuite les différents choix que nous avons faits pour aboutir à la mise en œuvre proposée. Ceux-ci concernent aussi bien l'identification des tâches de cet l'algorithme qui peuvent se paralléliser de manière performante, que le choix des mémoires utilisées et le moyen utilisé pour réduire l'effet de chemins divergents induits par des instructions conditionnelles. Nous proposons, ensuite, une mise en œuvre multi-GPU de l'algorithme du Simplexe mettant à contribution plusieurs cartes GPUs disponibles dans un seul système pour résoudre un problème de programmation linéaire. Nous expliquons comment partager le tableau du Simplexe entres les différents GPUs et comment diminuer ainsi les échanges entre les GPUs et le CPU. Enfin, nous présentons et analysons les résultats obtenues pour la mise en œuvre séquentielle, sur GPU et la mise en œuvre parallèle sur un système multi-GPU.

Nous terminons ce mémoire en présentant nos conclusions générales et les perspectives de recherche.

# Chapitre II

## Généralités sur le sac à dos

## Sommaire



## II.1 Introduction

Dans le présent chapitre, nous présentons le contexte dans lequel vont s'inscrire nos travaux de recherche. Ces travaux s'articulent autour de la résolution de problèmes d'optimisation.

Dans la sous-section II.2, Nous définirons le problème du sac à dos ainsi que le problème du *sac à dos multiple* (MKP). Nous définirons aussi la notion de noyau d'un problème de sac à dos.

Dans la sous-section II.3, Nous présenterons les deux méthodes exactes utilisées pour la résolution de problème de type sac à dos : le Branch and Bound et la programmation dynamique. Nous parlerons aussi de la manière de calculer les bornes supérieures et inferieures pour le problème du sac à dos et présenterons la méthode de réduction de variables qui est une étape de prétraitement d'un problème, afin de réduire sa cardinalité. Enfin nous présenterons quelques solveurs utilisés dans le domaine de l'optimisation.

## II.2 Le problème du sac à dos (KP)

De manière générale, un problème d'optimisation peut être formulé de la façon suivante :

$$(P)\begin{cases}\max f(x)\\ s.c.\ x \in X\end{cases} \qquad \text{(II.1)}$$

où:

- $f(.)$ est une fonction d'utilité à maximiser (ou minimiser en remplaçant max par min),
- X est un ensemble fini, défini par un ensemble de contraintes sur les variables.

On recherche alors une solution optimale $x^* \in X$ telle que $f(x^*) \geq f(x), \forall x \in X$ .

Plusieurs problèmes théoriques ou réels peuvent être écrits suivant (II.1), avec une fonction objectif et un ensemble de contraintes et de variables bien déterminés. Ces dernières peuvent être linéaires, entières ou mixtes.

On s'intéresse ici aux problèmes à variables entières 0-1 à savoir des problèmes de la famille du sac à dos.

Le problème du sac à dos est un problème classique d'optimisation combinatoire appartenant à la classe des problèmes NP-complets [FRE 04]. L'énoncé de ce problème est simple : étant donné un ensemble de *n* objets, où chaque objet *i* est caractérisé par un poids $w_i$ et un profit $p_i$ on cherche le sous-ensemble d'objets à charger dans un sac de capacité *c* afin de maximiser la somme des profits. Ainsi, le problème du sac à dos se présente sous la forme mathématique suivante :

$$(KP)\begin{cases} \max \sum_{i=1}^{n} p_i.x_i, \\ \sum_{i=1}^{n} w_i.x_i \le c, \\ x_i \in \{0,1\}, i \in \{1,..,n\} \end{cases} \quad \text{(II.2)}$$

Les poids $w_i$ et les profits $p_i$ ainsi que la capacité $c$ sont des entiers positifs $i \in \{1,..., n\}$.

La variable $x_i$ est la variable de décision ; elle prend la valeur 1 si l'objet $i$ est chargé dans le sac, sinon elle prend la valeur 0.

Résoudre ainsi le problème du sac à dos revient à trouver le vecteur solution $x^* = (x_1^*,...,x_n^*)^T$ qui optimise (maximise dans ce cas) la fonction objectif définie en (II.2).

Le problème du sac à dos, *Knapsack Problem* (KP) en anglais, et ces différentes variantes ont été longuement étudiés depuis le travail pionnier de Dantzig [DAN 57] en 1957. L'intérêt porté au sac à dos est dû au fait qu'il permet de modéliser de très nombreux problèmes comme les problèmes de gestion de capital, de chargement de cargaison, de rotation d'équipage, de tournées et de livraison ; par ailleurs, ce problème apparaît comme un sous-problème de nombreux problèmes d'optimisation combinatoire.

Les problèmes de type sac à dos apparaissent aussi fréquemment comme une relaxation de problèmes de programmation en nombre entier. Dans ce type d'application, on résout souvent un problème KP afin d'obtenir une borne supérieure. On présente dans ce qui suit le problème du *sac à dos multiple* que nous serons amenés à étudier au chapitre III. Nous définirons aussi une notion importante qui est le « noyau » du sac à dos.

## II.2.1 Le problème du sac à dos multiple

Le problème du *sac à dos multiple* (MKP) est une généralisation du problème standard du *sac à dos* en variable 0–1, où on essaie de remplir $m$ sacs à dos de différentes capacités au lieu de considérer un seul sac à dos.

Soit $N = \{1, ..., n\}$ l'ensemble des indices d'articles à charger où chaque article d'indice $j$ est caractérisé par un profit $p_j$ et un poids $w_j$. Nous considérons $m$ sacs à dos où chaque sac d'indice $i$, $i \in \{1, ..., m\}$ est de capacité de chargement $c_i$. Alors le problème du *sac à dos*

*multiple* consiste à remplir tous les sacs à dos de façon à maximiser le profit total et de sorte que la somme des poids dans chaque sac à dos d'indice *i* ne dépasse pas la capacité $c_i$.

Nous notons par $x_{ij}$ la variable binaire de décision qui prend la valeur 1 si l'article d'indice $j$ est affecté au sac à dos d'indice *i* et 0 dans le cas contraire. Le problème *du sac à dos multiple* (MKP) peut être formulé de la manière suivante :

$$(MKP)\begin{cases} \max \sum_{i=1}^{m}\sum_{j=1}^{n} p_j x_{ij}, & \text{(II.3)} \\ s.c.\ \sum_{j=1}^{n} w_j x_{ij} \le c_i, i \in \{1,\dots,m\}, & \text{(II.4)} \\ \sum_{i=1}^{m} x_{ij} \le 1, j \in \{1,\dots,n\}, & \text{(II.5)} \\ x_{ij} \in \{0,1\}, i \in \{1,\dots,m\}, j \in \{1,\dots,n\}; \end{cases}$$

où $p_j$, $c_i$ et $w_j$ sont des entiers positifs et les contraintes (II.4) et (II.5), assurent respectivement, que le remplissage du sac à dos *i* ne dépasse pas sa capacité correspondante $c_i$ et que chaque article sélectionné est attribué, au plus, à un seul sac à dos.

## II.2.2 Le noyau du sac à dos

De nombreux tests expérimentaux sur une large variété d'instances de problèmes de sac à dos ont montré que seul un sous-ensemble contenant un nombre relativement faible d'articles est crucial pour la détermination de la solution optimale, ceci est particulièrement vrai si le nombre d'articles est très grand [PIS 97].

Nous faisons l'hypothèse que les *n* articles sont triés par ratio profit sur poids décroissant, on dit aussi efficacité décroissante, comme illustré par l'inégalité (II.6).

$$\frac{p_1}{w_1} \ge \frac{p_2}{w_2} \ge \dots \frac{p_n}{w_n}. \quad \text{(II.6)}$$

Nous désignons par *s* l'indice de rupture ou de base du problème (*KP*) qui est donné par l'inégalité suivante :

$$\sum_{j=1}^{s-1} w_j \le c < \sum_{j=1}^{s} w_j . \tag{II.7}$$

Nous introduisons la notion de coût réduit pour un article *j* comme suit :

$$d_j = p_j - p_s . \frac{w_j}{w_s}, j \in \{1,...,n\}. \tag{II.8}$$

Les articles les plus attractifs sont ceux qui on le coût réduit le plus grand en valeur absolue

$$|d_1| \le |d_2| \le \cdots \le |d_n|. \tag{II.9}$$

Le noyau ou *core*, en anglais, correspond au sous-ensemble d'articles avoisinant l'article de base, d'indice *s*.

Balas et Zemel [BAL 80] ont donné une définition précise du noyau d'un problème de sac à dos, cette définition présentée ci-dessous est basée sur la connaissance d'une solution optimale du problème *LKP*.

**Définition II.1:** Supposons que les éléments sont triés suivant un ratio profit sur poids décroissant et notons par $x^*$ le vecteur solution optimale du problème *KP*. Posons

$$a := \min\{j \mid x_j^* = 0\}, \qquad b := \max\{j \mid x_j^* = 1\}. \tag{II.10}$$

L'ensemble $C = \{a, ..., b\}$ représente le noyau du problème (KP). De plus, si on pose $\tilde{p} = \sum_{j=1}^{a-1} p_j$ et $\tilde{w} = \sum_{j=1}^{a-1} w_j$ le problème du knapsack sur le noyau est formulé ainsi :

$$(KPC)\begin{cases} \max \sum_{j \in C} p_j x_j + \tilde{p}, \\ \text{s.c.} \sum_{j \in C} w_j x_j \le c - \tilde{w}, \\ x_j \in \{0,1\}. \end{cases} \tag{II.11}$$

Pour de nombreuses classes d'instances, la taille du noyau est petite par rapport à *n*. Par conséquent, si les valeurs *a* et *b* sont connues a priori, le problème initial peut facilement être résolu en posant $x_j^* = 1$ pour $j = 1, ..., a-1$ et $x_j^* = 0$ pour $j = b+1,..., n$ et en résolvant tout simplement le knapsack sur le noyau par la méthode de Branch and Bound ou la programmation dynamique. Nous rappelons que la notation $|C|$ correspond au cardinal de l'ensemble *C*.

En pratique, les valeurs de *a* et *b* ne sont pas connues a priori, la plupart des algorithmes utilisant la notion de noyau reposent sur le choix de $|C|$ articles autour de l'article de base. Plusieurs propositions ont été faites quant à la taille $|C|$ du noyau. Balas et Zemel [BAL 80] ont proposé de prendre une valeur constante $|C| = 50$ ; Martello et Toth [MAR 88] ont proposé une valeur $|C| = \sqrt{n}$ , puis $|C| = 2\sqrt{n}$ [MAR 97]. Ces différentes valeurs de $|C|$ se rapportent à des interprétations différentes du problème.

Des algorithmes de recherche de la taille du noyau ont aussi été proposés par Balas et Zemel [BAL 80] et Martello et Toth [MAR 88]. Pisinger a proposé un algorithme basé sur la méthode du *Quick-sort* modifiée (cf. Hoare [HOA 62]), qui détermine l'indice de la variable de rupture *s* et trie les $|C|$ articles autour de *s* (voir aussi référence [PIS 95]).

Plusieurs expérimentations ont été menées par Pisinger [PIS 97] sur différents types de problèmes (KP) pour déterminer la taille minimale du noyau. Celles-ci sont présentées dans le tableau ci-après :

| *N* | Problèmes Non corrélés | Problèmes faiblement corrélés | Problèmes Fortement Corrélés | Subset sum |
|---|---|---|---|---|
| 100 | 5 | 12 | 13 | 14 |
| 500 | 8 | 17 | 25 | 13 |
| 1000 | 11 | 17 | 36 | 13 |
| 5000 | 14 | 21 | 79 | 13 |
| 10000 | 17 | 25 | 104 | 13 |

TABLEAU II.1 – Taille minimale du noyau $|C|$ suivant le nombre d'articles (moyenne sur 100 instances).

La taille du noyau en pratique doit être plus grande que les valeurs présentées dans le tableau ci-contre, si l'on veut prouver l'optimalité de la solution obtenue.

De cette étude sur le noyau, plusieurs algorithmes exacts de résolution de problèmes (KP), appelés algorithmes noyau, ont été proposés. On dénombre deux types d'algorithmes :

- les algorithmes à noyau fixe (cf Balas et Zemel [BAL 80], Fayard et Plateau [FAY 82], Martello et Toth [MAR 88] [MAR 97]).

- les algorithmes à noyau extensible, comme l'algorithme *Expknap* présenté par Pisinger [PIS 95] qui utilise la méthode de Branch and Bound. Dans cette méthode, le noyau contient initialement la variable de base, mais cet ensemble est étendu à chaque fois que la méthode de Branch and Bound atteint les bords du noyau ou la solution du problème initiale.

Ces algorithmes utilisent généralement la méthode de Branch and Bound pour l'énumération.

## II.3 Méthodes de résolution exactes

Nous présentons dans cette sous-section deux méthodes sur lesquelles se basent un grand nombre d'algorithmes pour la résolution des problèmes de type sac à dos : la méthode de séparation et d'évaluation (Branch and Bound) et la méthode de programmation dynamique.

### II.3.1 La méthode de Branch and Bound

La méthode de Branch and Bound (B&B) [LAN 60] est l'une des méthodes les plus connues pour la résolution de problèmes d'optimisation combinatoire NP-difficiles. Cette méthode est basée sur une recherche arborescente d'une solution optimale par séparation et évaluation. Ainsi la résolution d'un problème combinatoire formulé par (II.1) consiste à trouver la solution optimale $x^* \in X$ telle que $f(x^*) \geq f(x), \forall x \in X$, où $f$ est la fonction objectif à maximiser.

Il est à noter que l'énumération de l'ensemble des éléments de $X$ est très souvent peu réaliste en raison de l'importance de son cardinal. La méthode de Branch and Bound tente d'explorer intelligemment l'ensemble des solutions admissibles en éliminant de l'espace de recherche les sous-ensembles de solutions qui ne peuvent pas fournir une solution optimale.

**Présentation de l'algorithme**

La recherche par décomposition de l'ensemble des solutions peut être représentée graphiquement par un arbre (voir la Figure II.1). C'est de cette représentation que vient le nom de "méthode de recherche arborescente".

- Chaque sous-problème créé au cours de l'exploration est symbolisé par un nœud de l'arbre (ou sommet), le nœud racine représentant le problème initial.

- Les branches de l'arbre symbolisent le processus de séparation. Elles représentent la relation entre les nœuds.
- Lors de la séparation, un nœud «père» crée un ensemble de nœuds «fils».

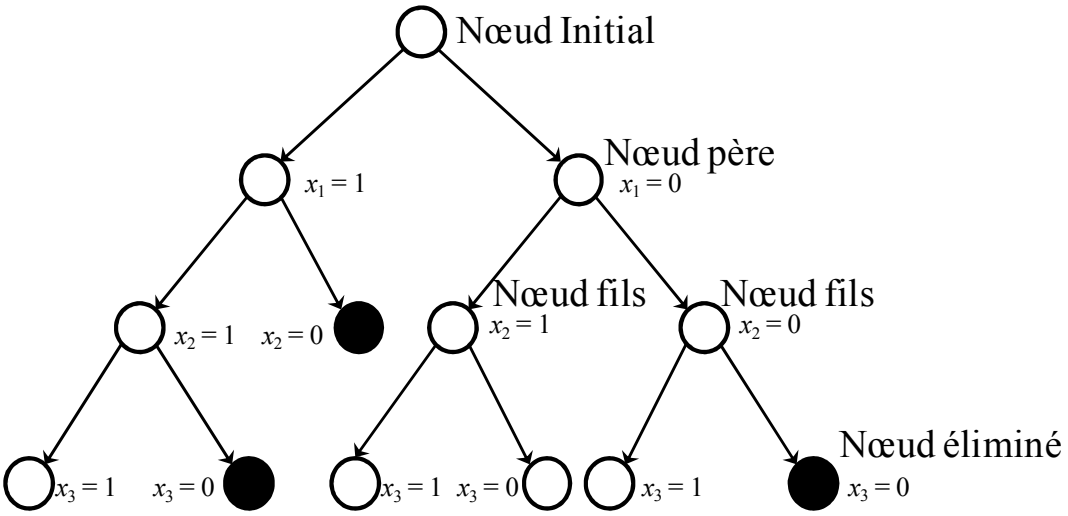



FIGURE II.1 Arbre engendré par décomposition d'un problème.

Le Branch and Bound est basé sur trois principes :

Principe d'évaluation

Le principe d'évaluation a pour objectif de connaître la qualité des nœuds à traiter.

La méthode de Branch and Bound utilise deux types de bornes

- une borne inférieure de la fonction d'utilité du problème initial,
- une borne supérieure de la fonction d'utilité.

La connaissance d'une borne inférieure du problème et d'une borne supérieure de la fonction d'utilité de chaque sous-problème permet de stopper l'exploration d'un sous-ensemble de solutions qui ne sont pas candidates à l'optimalité : si pour un sous-problème la borne supérieure est plus petite que la borne inférieure du problème, l'exploration du sous-ensemble correspondant est inutile. D'autre part, lorsque le sous-ensemble est suffisamment « petit », on procède à une énumération explicite : on résout alors le sous-problème correspondant.

Le principe de séparation :

Le principe de séparation consiste à diviser le problème en un certain nombre de sous-problèmes qui ont chacun leur ensemble de solutions réalisables. En résolvant tous les sous-problèmes et en prenant la meilleure solution trouvée, on est assuré d'avoir résolu le problème initial. Ce principe de séparation est appliqué de manière récursive à chacun des sous-ensembles tant que celui-ci contient plusieurs solutions.

**Remarque :** La procédure de séparation d'un ensemble s'arrête lorsqu'une des conditions suivantes est vérifiée :

- on connaît la meilleure solution de l'ensemble ;
- on connaît une solution meilleure que toutes celles de l'ensemble ;
- on sait que l'ensemble ne contient aucune solution admissible.

Stratégie de parcours :

La stratégie de parcours est la règle qui permet de choisir le prochain sommet à séparer parmi l'ensemble des sommets de l'arborescence.

Parmi les stratégies de parcours les plus connues, on peut citer :

- La profondeur d'abord :

  L'exploration privilégie les sous-problèmes obtenus par le plus grand nombre de séparations appliquées au problème de départ, c'est-à-dire aux sommets les plus éloignés de la racine (de profondeur la plus élevée).

  L'obtention rapide d'une solution admissible (pour les problèmes où il est difficile d'obtenir une heuristique de bonne qualité) et le peu de place mémoire nécessaire en sont les avantages. L'inconvénient est l'exploration de sous-ensembles qui peuvent s'avérer peu prometteurs à l'obtention d'une solution optimale.

- La largeur d'abord :

  Cette stratégie favorise les sous-problèmes obtenus par le moins de séparations du problème de départ, c'est-à-dire les sommets les plus proches de la racine (de profondeur la moins élevée).

- Le meilleur d'abord :

Cette stratégie favorise l'exploration des sous-problèmes possédant les plus petites bornes inferieures. Elle dirige la recherche là où la probabilité de trouver une meilleure solution est la plus grande.

## II.3.2 La Programmation Dynamique

La programmation dynamique [BEL 54], [BEL 57] est une méthode classique de résolution exacte qui peut-être utilisée pour la résolution d'un grand nombre de problèmes d'optimisation dont le problème du sac à dos. C'est une méthode récursive.

Nous introduisons les notions suivantes pour l'étape *k*, avec $k \in \{1,..., n\}$ où *n* est le nombre d'étapes :

- $X_k$ est l'ensemble des « décisions » que l'on prend à l'étape $k$ ,
- $E_k$ est l'ensemble des « états » dans lesquels le système se trouve à l'étape $k$ ,
- $C(e_{k-1}, x_k)$ est le profit attaché à la décision $x_k$ qui fait passer le système de l'état $e_{k-1}$ à l'état $e_k$ .
- $f(k, e_k) = \sum_{j=1}^{k} c(e_{j-1}; x_j)$ est le profit total de la séquence de décisions $x = (x_1, x_2, ..., x_k)$ qui fait passer le système de l'état $e_0$ à $e_k$ ,
- $F(e_{k-1}, x_k) = e_k$ est transition d'état.

Le profit associé à la décision $x_k$ lorsqu'on se trouve dans l'état $e_{k-1}$ ne dépend que de ces deux éléments et non de toutes les décisions antérieures qui ont permis d'aboutir à l'état $e_{k-1}$ (*propriété* Markovienne).

La programmation dynamique est basée sur le principe d'optimalité de Bellman.

Quelle que soit la décision optimale prise à l'étape $k$ qui amène le système de l'état $e_{k-1}$ à l'état $e_k$ la portion de la politique entre $e_0$ et $e_{k-1}$ est optimale.

La résolution du problème revient donc à trouver une séquence optimale $x^* = (x_1^*, x_2^*, ..., x_n^*)$ qui à partir de l'état initial $e_0$ nous amène à l'état $e_n$ tout en maximisant la fonction profit :

$$f(n, e_n) = \max\left\{\sum_{j=1}^{n} C(e_{j-1}, x_j) \middle| x_j \in X_j; e_j = F(e_{j-1}, X_j); j \in \{1,...,n\}\right\}. \quad \text{(II.12)}$$

En appliquant le principe d'optimalité, nous pouvons calculer de proche en proche $f(n, e_n)$ en utilisant l'équation récursive qui suit :

$$f(k, e_k) = \max_{x \in X_k; e_k = F(e_{k-1}, X_k)} \{C(e_{k-1}, x_k) + f(k-1, e_{k-1})\}, \quad \text{(II.13)}$$

où $f(k, e_k)$ représente le profit optimal du problème sur un horizon de k périodes.

**La programmation dynamique pour le problème du sac à dos**

Considérant la formulation du problème KP donnée par (II.2), nous définissons pour chaque couple $(k, g)$, $k = 1, \ldots, n$, $g = 0, \ldots, c$, le sous-problème $(KP(k, g))$ tel qu'illustré ci-contre :

$$(KP(k,g)) \begin{cases} \max \sum_{j=1}^{k} p_j . x_j, \\ s.c. \sum_{j=1}^{k} w_j . x_j \le g, \\ x_j \in \{0,1\}, j \in \{1,...,n\}. \end{cases} \quad \text{(II.14)}$$

Le profit $f(k, g)$ *est* obtenu en chargeant le sac à dos de capacité $g$ avec les $k$ premiers objets. De là, la solution du problème KP est donnée par la solution de $(KP(n, c))$.

Dans ce problème on a à l'étape $k$ :

- L'ensemble des décisions : $X_k = \{0, 1\}$, (II.15)
- l'ensemble des états dans lesquels le système peut être :

$$E_k = \left\{ e_k = g \middle| g \in \left\{0,1,...,\min\left\{c, \sum_{j=1}^{k} w_j\right\}\right\}\right\}, \quad \text{(II.16)}$$

- le profit de la décision $x_k$ qui fait passer le système de l'état $e_{k-1}$ à l'état $e_k$ :

$$C_{(}e_{k-1,}\, x_{k)} = p_{k.}x_{k,} \tag{II.17}$$

- la transition d'état :

$$F(e_{k-1}, x_k) = e_k = e_{k-1} + w_k x_k . \tag{II.18}$$

le profit total de la séquence de décisions $x = (x_1, \ldots, x_k)$ qui fait passer le système de l'état $e_0$ *à* $e_k$ :

$$\sum_{j=1}^{k} c(e_{j-1}, x_j) = \sum_{j=1}^{k} p_{j.}x_j, \tag{II.19}$$

La liste de la programmation dynamique à l'étape $k$ est constituée des éléments suivants :

- l'ensemble $E_k$ des états dans lesquels le système peut se trouver à l'étape $k$,
- la séquence de décisions optimales $x_g = (x_{g,1}, \ldots, x_{g,k})$ qui maximise le profit pour un sac à dos de capacité $g$.

On a alors pour chaque $g \in \{1, \ldots, c\}$ :

$$f(k,g) = \sum_{j=1}^{k} C(e_{k-1}, x_{g,j}). \tag{II.20}$$

La relation qui relie l'ensemble des éléments de la liste de la programmation dynamique à l'étape $k$, aux éléments de la liste à l'étape $k-1$, nous permet de construire récursivement la liste contenant l'ensemble des solutions des problèmes $f(k, g)$ pour un $k$ fixé :

$$f(k,g) = \begin{cases} f(k-1,g) \quad pour \quad g \in \{0,\ldots,w_k - 1\} \\ \max\{f(k-1,g), c_k + f(k-1, g-w_k)\} pour \quad g \in \{w_k, \ldots, \underline{c}\} \end{cases} \tag{II.21}$$

avec $\underline{c} = \min\left\{c, \sum_{j=1}^{k} w_j\right\}$.

**Le concept de listes**

A l'étape $k$, la liste récursive de la programmation dynamique $L_k$ est définie comme l'ensemble de couples suivant :

$$L_k = \left\{ (w,p) \middle| w = \sum_{j=1}^{k} w_j x_j \le c, p = \sum_{j=1}^{k} p_j x_j, \quad avec \quad \forall j \in \{1,..,k\} \; x_j \in \{0,1\} \right\}. \qquad \text{(II.22)}$$

*Exemple :*

Soit le problème (KP) avec $n = 4$ objets [(3,2), (5,4), (8,7), (10,10)] et une capacité $c = 15$:

$$(Ex) = \begin{cases} \max 2.x_1 + 4.x_2 + 7.x_3 + 10.x_4, \\ s.c.\ 3.x_1 + 5.x_2 + 8.x_3 + 10.x_4 \le 15 \\ x_j \in \{0,1\} \quad pour\ j \in \{1,...,4\}. \end{cases} \qquad \text{(II.23)}$$

La résolution du problème (II.23) via la méthode de programmation dynamique avec les listes, est illustré dans le tableau II.2.

| L0 | | L1 | | L2 | | L3 | | L4 | |
|---|---|---|---|---|---|---|---|---|---|
| (0, 0) | | (3,2) | | (5,4) | | (8,7) | | (10,10) | |
| *w* | *P* | *w* | *p* | *w* | *p* | *W* | *p* | *w* | *p* |
| | | | | | | | | 15 | 14 |
| | | | | | | | | 13 | 12 |
| | | | | | | | | 13 | 11 |
| | | | | | | 13 | 11 | 11 | 9 |
| | | | | | | 11 | 9 | 10 | 10 |
| | | | | | | 8 | 7 | 8 | 7 |
| | | | | 8 | 6 | 8 | 6 | 8 | 6 |
| | | | | 5 | 4 | 5 | 4 | 5 | 4 |
| | | 3 | 2 | 3 | 2 | 3 | 2 | 3 | 2 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

TABLEAU II.2– Listes de la programmation dynamique (en gris, les états éliminés).

*Le principe de dominance*

Soit un couple $(w,p)$. Si $\exists (w', p')$ tel que $p \le p'$ et $w \ge w'$, alors on dit que le couple $(w, p)$ est dominé par le couple $(w', p')$.

Le principe de dominance découle directement du principe d'optimalité. Par conséquent les éléments dominés sont supprimés de la liste de la programmation dynamique, réduisant ainsi considérablement la taille des listes et donc le travail de recherche d'une solution optimale.

Le Tableau II.2 montre la suppression des éléments dominés (en gris) dans la liste de la programmation dynamique.

Dans l'exemple ci-dessus, le couple (11,9) correspondant à la combinaison des objets 1 et 3, est dominé par le couple (10,10) car il a un poids supérieur mais un profit inférieur.

L'algorithme de programmation dynamique a une complexité égale à $O(min(2^n, n.c))$ (cf Ahrens et Finke [AHR 75]), (voir aussi Horowitz et Sahni [HOR 74], Toth [TOT 80]) et Plateau et Elkihel [ELK 84].

**Décomposition du problème (méthode des deux listes)**

Horowitz et Sahni [HOR 74] ont proposé en 1974, une méthode de programmation dynamique basée sur la décomposition du problème KP en 2 sous-problèmes (KP1 et KP2) de tailles égales $n_1 = \lfloor n/2 \rfloor$ *et* $n_2 = n - n_1$ variables respectivement :

$$(KP_1)\begin{cases} \max \sum_{j=1}^{n_1} p_j.x_j, \\ s.c. \sum_{j=1}^{n_1} w_j.x_j \le c, \\ x_j \in \{0,1\}, j \in \{1,...,n_1\}, \end{cases} \quad \text{et} (KP_2)\begin{cases} \max \sum_{j=n_1+1}^{n} p_j.x_j, \\ s.c. \sum_{j=n_1+1}^{n} w_j.x_j \le c, \\ x_j \in \{0,1\}, j \in \{n_1+1,...,n\}. \end{cases} \qquad \text{(II.24)}$$

Fig. II.3 – Les deux problèmes de la méthode des listes

La programmation dynamique est appliquée à chacun des deux sous-problèmes, donnant alors deux listes contenant l'ensemble des états non dominés que nous noterons $L_1$ et $L_2$.

Résoudre le problème initial du (KP) revient à chercher, parmi l'ensemble des combinaisons réalisables possibles entre les états des deux listes, celle qui maximise le profit. Cela revient à fusionner les deux listes.

Pour cette recherche, la liste $L_1$ est parcourue selon les poids décroissants et $L_2$ selon les poids croissants. On se sert d'un pivot $P$ qui sera alternativement dans $L_1$ puis dans $L_2$.

P = ($w$, $p$) étant dans $L_1$, on parcourt $L_2$ jusqu'à ($w'$, $p'$) tel que $w + w' \geq$ c. Le nouveau pivot est alors P = ($w'$, $p'$). On continue alors le parcours de la liste $L_1$ à partir de ($w'$, $p'$) jusqu'à

($w''$, $p''$) tel que $w' + w'' \leq c$ et on prend P = ($w''$, $p''$) et ainsi de suite. La recherche s'arrête lorsque la fin d'une des deux listes est atteinte.

La complexité temporelle de construction des deux listes est de $O(\min(2^{\frac{n}{2}}, \frac{n}{2}.c))$, et celle de la fusion des deux listes est de $O(\min(2^{\frac{n}{2}}, c))$. En ce qui concerne la complexité spatiale, elle passe de $O(\min(2^n, c))$ à $O(\min(2^{\frac{n}{2}}, c))$.

La complexité temporelle et la complexité spatiale de l'algorithme de programmation dynamique, se trouvent ainsi diminuées.

## II.4 Calcul de bornes et méthode de réduction de variables

Il est évident d'après la description de la méthode de Branch and Bound que le calcul des bornes inférieures et supérieures est un point important permettant d'affiner les performances de cette méthode.

Dans ce qui suit, nous passerons brièvement en revue quelques-unes des méthodes de calcul de bornes dans la section suivante.

### II.4.1 Bornes supérieures pour le problème KP

La plupart des bornes supérieures pour le problème (KP) se basent sur le tri des objets par ordre décroissant d'efficacité, voir la relation (II.6), où l'efficacité d'un objet *j* est représentée par le ratio $e_j = \frac{p_j}{w_j}$.

La borne supérieure la plus simple du problème du sac-à-dos (KP), peut être obtenue en prenant la solution optimale de la relaxation continue du problème du sac à dos (*LKP*) et en considérant son arrondi à la valeur entière inférieure.

$$(LKP)\begin{cases} \max \sum_{i=1}^{n} p_i.x_i, \\ \sum_{i=1}^{n} w_i.x_i \le c, \\ 0 \le x_i \le 1, i \in \{1,..,n\} \end{cases} \qquad \text{(II.25)}$$

Ainsi on aboutit à la borne supérieure présentée par Dantzig [DAN 57] qui est exprimée comme suit :

$$U_1 = U_{LKP} = \lfloor Z^{LKP} \rfloor, \tag{II.26}$$

où $Z^{LKP} = \sum_{j=1}^{s-1} p_j + (c - \sum_{j=1}^{s-1} w_j)\frac{p_s}{w_s}$, et $s$ est l'indice de la variable de rupture, (voir l'équation II.8).

Les articles étant triés en fonction de (II.6). $U_1$ peut être obtenu de façon simple en $O(n)$.

Une autre borne supérieure peut être obtenue au moyen de la relaxation Lagrangienne. Pour le problème KP, nous pouvons relaxer la contrainte sur la capacité ce qui conduit au problème $L(KP, \lambda)$ , illustré si dessous :

$$L(KP,\lambda)\begin{cases} \max \sum_{j=1}^{n} p_j x_j + \lambda\left(c - \sum_{j=1}^{n} w_j x_j\right) \\ s.c. \quad x_j \in \{0,1\} \quad j = 1,\ldots,n. \end{cases} \tag{II.27}$$

Cependant, la relaxation Lagrangienne ne peut fournir une meilleur borne que $U_1$, voir [WOL 98]. En effet le meilleur choix pour le multiplicateur, $\lambda$ serait la valeur $\frac{p_s}{w_s}$.

Martello et Toth [MAR 77] ont proposé une meilleure borne $U_2$ en considérant séparément le cas où l'article de base $s$ est chargé ou exclu, ce qui mène à la borne supérieure suivante :

$$U_2 = \max\left\{\left\lfloor \sum_{j=1}^{s-1} p_j + \left(c - \sum_{j=1}^{s-1} w_j\right)\frac{p_{s+1}}{w_{s+1}} \right\rfloor, \left\lfloor \sum_{j=1}^{s-1} p_j + p_s + \left(c - \sum_{j=1}^{s-1} w_j - w_s\right)\frac{p_{s-1}}{w_{s-1}} \right\rfloor\right\} \tag{II.28}$$

Il est facile de voir que $U_2$ peut être calculée en $O(n)$ , de plus, nous avons $U_2 \le U_1$.

Plusieurs autres bornes ont été proposées dans la littérature, à savoir Martello et Toth [MAR 88], Fayard et Plateau [FAY 82], Müller-Merbach [MUL 79], et Dudzinski et Walukiewicz [DUD 87]. Dans le cadre de ce travail, nous avons utilisé la borne $U_2$ dans le chapitre III et $U_1$ dans le chapitre V.

## II.4.2 Bornes inferieures pour le problème KP

Afin de résoudre le problème (KP) notamment par la méthode de Branch and Bound, il est nécessaire de disposer d'une borne inferieure $L$ avec laquelle les bornes supérieures peuvent être comparées. Bien que l'on puisse initialement fixer $L$ à 0, de bonnes mises en œuvre de la méthode de Branch and Bound reposent sur de bonnes bornes inferieures proches de la solution optimale.

La borne inférieure la plus simple peut être obtenue par l'algorithme Glouton (voir Chapitre III) comme suit :

$$L_1 = \max_{i=s+1,\ldots,n} \left\{ \sum_{j=1}^{s-1} p_j + p_i \;\middle|\; \sum_{j=1}^{s-1} w_j + w_i \le c \right\} \tag{II.29}$$

Une autre borne analogue à la borne ci-dessus a été proposée par Pisinger [PIS 95], cette fois-ci l'article correspondant à l'indice $s$ est chargé et on applique l'algorithme Glouton pour remplir le sac de capacité $c - w_s$. Ainsi la borne inferieure obtenue est la suivante :

$$L_2 = \max_{i=1,\ldots,s-1} \left\{ \sum_{j=1}^{s} p_j - p_i \;\middle|\; \sum_{j=1}^{s} w_j - w_i \le c \right\} \tag{II.30}$$

Ces bornes inferieures sont particulièrement adaptées aux problèmes fortement corrélés. De plus, la complexité en temps des ces bornes est égale à $O(n)$.

Plusieurs autres bornes inferieures pour le problème KP peuvent être déduites à l'aide de techniques d'approximation, voir [KEL 04].

## II.4.3 Réduction de variables

Il est avantageux de réduire la taille d'un problème (KP) avant qu'il ne soit résolu. Cela vaut en particulier si le problème doit être résolu de manière exacte, mais aussi lors de la recherche de solutions approchées à l'aide d'heuristiques ou d'algorithmes d'approximation qui sont, en général, plus efficaces pour de petites instances.

La réduction de variables réduit le temps d'exécution des algorithmes pour presque tous les cas survenant dans la pratique. Cependant, dans le pire des cas, les techniques de réduction de variables s'avèrent inefficaces.

Les algorithmes de réduction de variables peuvent être considérés comme un cas particulier de la méthode de Branch and Bound où l'on effectue la séparation sur une seule variable au niveau du nœud de départ. Ainsi, toutes les variables sont tour à tour choisies comme variables de séparation au niveau du nœud de départ.

Dans le cas du problème du sac à dos (KP), la séparation se fait sur chacune des variables $x_j$, avec $j$ = 1, ...., $n$ créant à chaque fois 2 sous-problèmes c'est-à-dire un sous-problème avec $x_j$ = 1 et un autre avec $x_j = 0$.

L'intérêt de la réduction de variables est de fixer définitivement le plus grand nombre de variables à leur valeur optimale, et ceci en un temps de calcul peu couteux.

Le premier algorithme de réduction à été proposé par Ingargiola et Korsh [ING 73]. Il se déroule comme suit :

A chaque variable $x_j$ ( $j \in \{1,..., n\}$), on associe deux bornes supérieures, à savoir :

- $U_j(x_j = 1)$, est la borne supérieure calculée pour le sous problème correspondant au cas où $x_j$=1.
- $U_j(x_j = 0)$, est la borne supérieure calculée pour le sous-problème correspondant au cas où $x_j = 0$.

Notons alors $L$ la borne inférieure du problème du sac à dos. La réduction de variables se fait comme suit :

Pour $j \in \{1,\cdots,n\}$,

- si $U_j\,(x_j = 0) \leq L$, alors on peut fixer $x_j$ à 1,
- si $U_j\,(x_j = 1) \leq L$, alors on peut fixer $x_j$ à 0.

Le calcul des bornes est effectué comme indiqué aux sous-sections II.4.1 et II.4.2. De plus, on note $\overline{U}$ la borne supérieure du problème KP et $\bar{x} = (\bar{x}_1,\ldots,\bar{x}_n)$ la solution associée.

Notons alors $J_1 = \{j \in \{1,\ldots,n\} \mid \bar{x}_j = 1\}$ et $J_0 = \{j \in \{1,\ldots,n\} \mid \bar{x}_j = 0\}$, nous avons alors :

- pour $j \in J_1$, $U_j\,(x_j = 1) = \overline{U}$ et
- pour $j \in J_0$, $U_j\,(x_j = 0) = \overline{U}$.

Il restera donc à calculer $2.n - |J_1| - |J_0| + 1$ bornes supérieures.

Une autre méthode se basant sur les couts réduits a été présentée par Fayard et Plateau [FAY 75]. Celle-ci garde la même structure précédente, excepté le fait qu'elle utilise les coûts réduits optimaux pour le calcul des bornes supérieures.

Rappelons que les coûts réduits sont donnés par la relation II.8, les bornes supérieures sont alors calculées comme suit :

$$U_j\left(x_j = 1 - \bar{x}_j\right) = \left\lfloor \overline{U} - \left| d_j \right| \right\rfloor \tag{II.31}$$

Ces bornes sont comparées de la même manière à la borne inferieure $L$ :

- si $U_j\left(x_j = 1 - \bar{x}_j\right) \leq L$, alors on peut fixer $x_j$ à $\bar{x}_j$,

Les bornes supérieures ainsi obtenues sont de moins bonnes qualités que les bornes supérieures calculées dans les 2 sections précédentes, néanmoins, elles sont calculées plus rapidement.

## II.5 Les solveurs existants

De nombreux solveurs ont été développés pour la résolution du problème (KP). Les principaux d'entre eux sont repris ci-dessous. Des logiciels libres sont aussi mis à la disposition de la communauté scientifique, notamment par le biais d'internet.

*a. Les logiciels commerciaux*

Les logiciels commerciaux présentent souvent des performances supérieures aux solveurs libres. Ils disposent de plus de mises à jour régulières et d'une assistance en cas de problèmes. Ce sont des outils aboutis et complets présentant une bonne flexibilité (intégration dans différents langages de programmation sous la forme de librairie et une interface utilisateur plus conviviale).

Les deux principaux logiciels existants sont :

- CPLEX de IBM ILOG :

*http : //www.ilog.com/products/cplex*

On notera que l'outil CPLEX est aussi disponible gratuitement pour l'usage académique (université).

- XPRESS-MP de Dash Optimization :

*http : //www.dashoptimization.com/home/products/products optimizer.html*

Les algorithmes utilisés sont à priori robustes et permettent de traiter une grande variété de problèmes différents.

*b. Les logiciels libres*

Une des plus ferventes communautés du logiciel libre regroupée autour du Projet GNU propose un ensemble d'outils, libres pour l'optimisation de programmes linéaires. Ces outils, regroupés sous le nom de GLPK (GNU linear programming kit) sont disponibles sur le site internet :

*http : //www.gnu.org/software/glpk/glpk.html.*

GLPK se présente sous la forme d'une librairie pouvant être utilisée par des programmes en langage C/C++. Il intègre les composants suivants :

– la méthode du Simplexe révisé,

– la méthode des points intérieurs primal-dual,

– la méthode de Branch and Bound,

– un solveur de programmes linéaires et programmes mixtes en nombres entiers.

Bernard Le Cun et François Galea du laboratoire PRISM de l'Université de Versailles ont développé et continuent de faire évoluer un outil appelé Bob++, programmée en langage C++. Cet outil inclut des bibliothèques servant d'appui lors de développement d'applications de type séparation et évaluation, notamment le Branch and Bound ou la programmation dynamique.

Bob++ est disponible via le site internet :

*http://www.prism.uvsq.fr/blec/BOBO/main.html.*

Bob++ comprend, entre autres, les composants suivants :

- différents algorithmes de recherche,
- différentes méthodes heuristiques,
- une gestion automatique des différents paramètres,
- une gestion des erreurs,
- une gestion transparente du parallélisme.

## II.6 Conclusion

Nous avons présenté dans ce chapitre quelques problèmes d'optimisation combinatoire appartenant à la famille des problèmes du sac à dos. Certains de ces problèmes seront détaillés dans les chapitres suivants, à savoir le problème du sac à dos multiple. Nous avons aussi présenté les méthodes classiques de résolution de problèmes de la famille du sac à dos à savoir, la méthode de la programmation dynamique qui a été détaillée dans ce chapitre, et la méthode de Branch and Bound qui sera plus approfondie au chapitre V.

Nous avons présenté sommairement les solveurs les plus utilisés dans la résolution des problèmes d'optimisation combinatoire.

Dans le chapitre suivant nous proposons des heuristiques pour le problème du sac à dos multiple.

# Chapitre III

## Le problème du sac à dos multiple

## Sommaire



## III.1 Introduction

Dans ce chapitre, nous présentons une heuristique qui fournit une solution réalisable pour le problème du *sac à dos multiple* (MKP). L'heuristique proposée appelée *RCH*, pour *Recursive Core Heuristic*, est une méthode récursive qui utilise le noyau de chaque sac à dos. L'heuristique *RCH* est comparée à l'heuristique *MTHM* de Martello et Toth. Des résultats de calcul sur des instances engendrées aléatoirement montrent que l'approche proposée aboutit à de meilleures solutions réalisables dans des temps de calcul plus faibles.

## III.2 Le problème du sac à dos multiple MKP

Le chargement de fret sur les navires est une application du monde réel du MKP, voir Eilon et Christofides [EIL 71]. Le problème est de choisir certains conteneurs dans un ensemble de *n* conteneurs à charger dans *m* navires de différentes capacités de chargement. Le problème *MKP* est parfois appelé le problème de chargement multiple.

Un autre problème MKP du monde réel a été présenté par Wirsching [WIR 98]. Une cargaison de *m* plaques de marbre provenant d'une carrière, est reçue par une entreprise. Ces plaques ont une taille uniforme et sont beaucoup plus longues que larges. Cette entreprise fabrique différents produits, qui doivent d'abord être découpés à partir des plaques de marbre puis traités ultérieurement. Selon le contenu et la disponibilité du stock, l'entreprise prépare une liste de produits qu'il pourrait être intéressant de produire. Sur cette liste, certains produits doivent être sélectionnés et découpés à partir des plaques, de manière à minimiser la quantité totale d'éléments de marbre gaspillés. Ce problème a été modélisé comme un cas particulier du problème MKP où les profits sont identifiés au poids, les dalles aux sacs à dos et les longueurs aux poids.

D'autres applications industrielles concernent le chargement de *n* réservoirs avec *m* liquides qui ne peuvent pas être mélangés, voir [MAR 80], le chargement de véhicules, voir [HIF 09], l'affectation de tâches et l'ordonnancement des tâches dans les multiprocesseurs, voir [LAB 03].

Le problème MKP est NP-complet. Nous rappelons la formulation du problème du sac à dos multiple :

$$(MKP)\begin{cases} z(MKP) = \max \sum_{i=1}^{m}\sum_{j=1}^{n} p_j x_{ij}, & \text{(III.1)} \\ s.c.\ \sum_{j=1}^{n} w_j x_{ij} \le c_i, i \in \{1,\dots,m\}, & \text{(III.2)} \\ \sum_{i=1}^{m} x_{ij} \le 1, j \in \{1,\dots,n\}, & \text{(III.3)} \\ x_{ij} \in \{0,1\}, i \in \{1,\dots,m\}, j \in \{1,\dots,n\}; \end{cases}$$

où $p_j$, $c_i$ et $w_j$ sont des entiers positifs. On note par $N = \{1,\dots,n\}$ ensemble d'articles à charger dans les *m* sacs à dos.

Afin d'éviter les cas triviaux, nous faisons les hypothèses suivantes :

- Tous les articles peuvent être chargés (au moins dans le plus grand sac) :

$$\max_{j \in N} w_j \leq \max_{i \in \{1,\ldots,m\}} c_i . \quad \text{(III.4)}$$

- Le plus petit article peut être chargé au moins dans le plus petit sac :

$$\min_{j \in N} w_j \leq \min_{i \in \{1,\ldots,m\}} c_i . \quad \text{(III.5)}$$

- Aucun sac à dos ne peut être rempli avec tous les articles de $N$ :

$$\sum_{j=1}^{n} w_j \geq \max_{i \in \{1,\ldots,m\}} c_i . \quad \text{(III.6)}$$

Nous supposons également que les articles sont triés par ordre de ratio profit sur poids décroissant.

$$\frac{p_1}{w_1} \geq \frac{p_2}{w_2} \geq \cdots \geq \frac{p_n}{w_n}, \quad \text{(III.7)}$$

de plus les sacs sont triés par ordre croissant

$$c_1 \leq c_2 \leq \cdots \leq c_m . \quad \text{(III.8)}$$

## III.3 Etat de l'art

### III.3.1 Différentes relaxations du problème MKP

Les méthodes de relaxation sont généralement utilisées afin d'estimer un majorant de la valeur optimale du problème *MKP*. Dans cette section nous présentons les principales relaxations.

**a. La relaxation surrogate**

Soit $\mu = (\mu_1, ..., \mu_m)$ un vecteur de multiplicateurs non négatifs correspondant aux $m$ contraintes de capacité du problème *MKP*, le problème relaxé $S(MKP, \mu)$ est alors donnée par:

$$S(MKP, \mu)\begin{cases} z(S(MKP, \mu)) = \max \sum_{i=1}^{m} \sum_{j=1}^{n} p_j x_{ij}, \\ s.c. \sum_{i=1}^{m} \mu_i \sum_{j=1}^{n} w_j x_{ij} \le \sum_{i=1}^{m} \mu_i c_i, \\ \sum_{i=1}^{m} x_{ij} \le 1, \quad j = 1, ..., n, \\ x_{ij} \in \{0, 1\}, i = 1, .., m, j = 1, ..., n. \end{cases} \quad \text{(III.9)}$$

Martello et Toth [MAR 81] ont montré que le meilleur choix de vecteur des multiplicateurs $\mu$ peut être déduit du théorème III.1:

**Théorème III.1:** Pour toute instance du problème *MKP*, le choix optimal des multiplicateurs $\mu_1, ..., \mu_m$ dans $S(MKP, \mu)$ est $\mu_i = k$, $i = 1, ..., m$, où $k$ est une constante positive (voir une démonstration dans la référence [KEL 04]).

Ainsi le problème relaxé $S(MKP, \mu)$ devient un problème de sac à dos (*KP*) standard, de la forme :

$$(KP)\begin{cases} z(KP) = \max \sum_{j=1}^{n} p_j x'_j, \\ s.c. \sum_{j=1}^{n} w_j x'_j \le c, \\ x'_j \in \{0, 1\}, \quad j = 1, ..., n. \end{cases} \quad \text{(III.10)}$$

Où $c = \sum_{i=1}^{m} c_i$ et la variable binaire introduite $x'_j = \sum_{i=1}^{m} x_{ij}$ indique si l'article $j$ a été chargé dans l'un des sacs à dos $i = 1, ..., m$.

**b. La relaxation continue**

Une approche différente est la relaxation continue des variables $x_{ij}$ du problème *MKP*, conduisant à un problème *C*(*MKP*).

$$C(MKP)\begin{cases} z(C(MKP)) = \max \sum_{i=1}^{m}\sum_{j=1}^{n} p_j x_{ij}, \\ s.c. \sum_{j=1}^{n} w_j x_{ij} \le c_i, i \in \{1,\dots,m\}, \\ \sum_{i=1}^{m} x_{ij} \le 1, j \in \{1,\dots,n\}, \\ 0 \le x_{ij} \le 1, i \in \{1,\dots,m\}, j \in \{1,\dots,n\}. \end{cases} \quad \text{(III.11)}$$

Martello et Toth [MAR 90] ont prouvé que $z(C(MKP)) = z(C(S(MKP, 1)))$, ainsi la valeur de la fonction objectif du problème relaxé $z(C(MKP))$ peut être trouvée en un temps $O(n)$ en utilisant l'algorithme *Split* proposé par Balas et Zemel [BAL 80] pour résoudre la relaxation surrogate continue du problème $S(MKP, 1)$. La qualité des deux bornes est cependant la même.

**c. La relaxation Lagrangienne**

Deux relaxations Lagrangienne du problème *MKP* sont aussi possibles. Par la relaxation des contraintes $\sum_{i=1}^{m} x_{ij} \le 1$ et un vecteur $\lambda = (\lambda_1, \dots, \lambda_n)$ de multiplicateurs non négatifs nous obtenons le problème relaxé $L_1(MKP, \lambda)$ donné par:

$$L_1(MKP,\lambda)\begin{cases} z(L_1(MKP,\lambda)) = \max \sum_{i=1}^{m}\sum_{j=1}^{n} p_j x_{ij} - \sum_{j=1}^{n} \lambda_j \left( \sum_{i=1}^{m} x_{ij} - 1 \right) \\ s.c. \sum_{j=1}^{n} w_j x_{ij} \le c_i, \qquad i = 1,\dots,m, \\ x_{ij} \in \{0,1\}, \ i = 1,\dots,m, j = 1,\dots,n. \end{cases} \quad \text{(III.12)}$$

En posant $\tilde{p}_j := p_j - \lambda_j$ pour $j = 1, \dots, n$, le problème relaxé peut être décomposé en $m$ problèmes de sac à dos indépendants, où chaque problème est de la forme :

$$(KP_i)\begin{cases} z_i(KP) = \max \sum_{j=1}^{n} \widetilde{p}_j x_{ij} \\ s.c. \sum_{j=1}^{n} w_j x_{ij} \le c_i, \\ x_{ij} \in \{0,1\},\ j = 1, ..., n. \end{cases} \quad \text{(III.13)}$$

Pour $i$ = 1, ..., $m$, tous les problèmes ont les mêmes profits et poids. Cependant seule la capacité est différente d'un problème à un autre. La solution optimale du problème relaxé est alors obtenue ainsi :

$$z(L_1(MKP, \lambda)) = \sum_{i=1}^{m} z_i + \sum_{j=1}^{n} \lambda_j \ .$$

Contrairement à la relaxation surrogate, il n'existe dans ce cas aucun moyen analytique qui permette de trouver un choix optimal pour les multiplicateurs $\lambda$. Cela signifie que des valeurs approximatives doivent être trouvées.

Ainsi, en supposant que les articles sont triés suivant l'ordre donné par la relation (III.4), Hung et Fisk [HUN 78] ont utilisé des multiplicateurs prédéfinis donnés par

$$\overline{\lambda}_j := \begin{cases} p_j - w_j p_s / w_s \ \textit{if} \ \ j < s, \\ 0 \qquad\qquad\qquad \textit{if} \ \ j \ge s, \end{cases} \quad \text{(III.14)}$$

où $s$ est l'indice de rupture du problème $S(MKP, 1)$, 1 étant une abréviation du vecteur de dimension $m$ (1, ..., 1). Avec ce choix de $\lambda_j$, nous aboutissons à l'égalité suivante (voir la démonstration dans la référence [HUN 78] ) :

$$z(C(L_1(MKP, \overline{\lambda}))) = z(C(S(MKP, 1))) = z(C(MKP)). \quad \text{(III.15)}$$

Ce qui montre que ce choix du vecteur multiplicateur $\overline{\lambda}$ est le meilleur pour le problème $C(L_1(MKP))$. En outre, les deux relaxation $L_1(MKP, \lambda)$ et $S(MKP, 1)$ fournissent de meilleures bornes que la relaxation continue. Il n'existe cependant pas de dominance entre cette relaxation Lagrangienne et la relaxation surrogate.

Une autre relaxation Lagrangienne $L_2(MKP, \lambda)$ est aussi possible en relaxant cette fois-ci les contraintes de poids. En utilisant un vecteur $\lambda = (\lambda_1, ..., \lambda_m)$ de multiplicateurs non négatifs, le problème devient relaxé, comme suit :

$$L_2(MKP, \lambda)\begin{cases} z(L_2(MKP, \lambda)) = \max \sum_{i=1}^{m}\sum_{j=1}^{n} p_j x_{ij} - \sum_{i=1}^{m} \lambda_i \left( \sum_{j=1}^{n} w_j x_{ij} - c_i \right) \\ s.c. \ \sum_{i=1}^{m} x_{ij} \le 1, \ \ j = 1, ..., n, \\ x_{ij} \in \{0, 1\}, \ \ i = 1, ..., m, \ j = 1, ..., n. \end{cases} \quad \text{(III.16)}$$

Si l'un des multiplicateurs $\lambda_i$ est nul, alors la solution optimale du problème (III.13) revient à mettre tous les articles dans le sac à dos *i*, ce qui donne une borne supérieure inutile. Ainsi, on peut supposer que $\lambda_i > 0$ pour tout $i = 1, ..., m$.

La fonction objectif devient alors

$$z(L_2(MKP, \lambda)) = \max \sum_{i=1}^{m}\sum_{j=1}^{n} (p_j - \lambda_i w_j) x_{ij} + \sum_{i=1}^{m} \lambda_i c_i . \quad \text{(III.17)}$$

Ce qui montre que la solution optimale peut être trouvée en choisissant le sac à dos avec la plus petite valeur associée de $\lambda_i$ dont l'indice sera noté *t* et en choisissant tous les articles avec $p_j - \lambda_t w_j > 0$ pour ce sac à dos. Cependant, cette relaxation fournit une borne supérieure moins bonne que celle fournie par la relaxation continue *C(MKP)* du problème *MKP*.

## III.3.2 Algorithmes pour le problème du MKP

Plusieurs algorithmes pour la résolution exacte ou approchée du problème MKP ont été proposés, parmi lesquels il convient de mentionner Neebe et Dannenbering [NEE 77], Christofides, Mingozzi et Toth [CHR 79], Hung et Fisk [HUN 78], Martello et Toth [MAR 80], et Pisinger [PIS 99]. Les deux premiers algorithmes sont destinés à des problèmes avec de nombreux sacs à dos et peu d'objets à charger, tandis que les derniers sont plus adaptés à des problèmes avec un grand nombre d'objets à charger dans chaque sac à dos.

Hung et Fisk [HUN 78] ont proposé une méthode à base de Branch and Bound avec comme stratégie de parcours profondeur d'abord. Les bornes supérieures sont obtenues à l'aide de la relaxation Lagrangienne et la séparation se fait sur l'objet qui – dans le cas du problème

relaxé – a été chargé dans la pluspart des sacs à dos. A chaque étape de séparation sur un article, ce dernier est alternativement attribué à chaque sacs à dos et ce parcours se fait dans le sens décroissant des capacité $c_i$ c'est à dire $c_1 \geq c_2 \geq \cdots \geq c_m$. Lorsque tous les sacs ont été considérés, une dernière branche de l'arborescence de la méthode de Branch and Bound est alors crée pour le cas où l'objet est exclu de tous les sacs à dos.

Un autre algorithme à base de Branch and Bound a été proposé par Martello et Toth [MAR 80], où à chaque nœud de séparation, un problème MKP relaxé de la contrainte $x_{ij} \leq 1$ est résolu. La séparation se fait alors sur l'article qui a été sectionnée dans $k > 1$ sac à dos, dans le cas du problème MKP relaxé. Ainsi, l'opération de séparation engendre *k* nœuds en assignant l'article à l'un des $k-1$ sacs à dos ou en l'excluant de tous ces derniers. Une technique paramétrique a été utilisée pour accélérer le calcul de la borne supérieure à chaque nœud.

Dans un article ultérieur, Martello et Toth [MAR 81] se sont concentrés sur deux faits qui rendent difficile la résolution du problème MKP:

- Généralement, il est difficile de vérifier la faisabilité de la solution fournie par la borne supérieure obtenue soit par relaxation surrogate, soit par relaxation Lagrangienne.
- L'algorithme de Branch and Bound requiert de bonnes bornes inférieures pour le parcours des nœuds dans l'énumération.

Afin d'éviter ces problèmes, Martello et Toth [MAR 81] ont proposé un algorithme « bound and bound » pour le problème du MKP nommé MTM. A chaque nœud de séparation, une borne supérieure et une borne inférieure sont calculées. Cette technique est bien adaptée pour des problèmes où il est facile de trouver une solution heuristique rapide qui donne de bonnes bornes inférieures et où il est difficile de vérifier la faisabilité de la solution fournie par la borne supérieure.

Pisinger [PIS 99] a amélioré cet algorithme par l'incorporation d'un algorithme performant pour le calcul des bornes supérieures à l'aide de la relaxation surrogate ainsi que de meilleures règles de réduction qui permettent de déterminer les articles qui peuvent être fixés à zéro et une méthode qui tente de réduire la capacité des sacs à dos.

**L'heuristique de Hung et Fisk**

Une approche heuristique différente a été proposée par Hung et Fisk [HUN 78] ; elle est basée sur la solution exacte de la relaxation surrogate $S(MKP)$ du problème MKP. Notons $Z_s$ le sous-ensemble d'articles dont la somme des profits est égale à $z(S(MKP))$.

L'algorithme proposé considère les articles de $Z_s$ dans l'ordre décroissant de poids, et tente d'insérer chaque article dans un sac à dos choisis au hasard ou, si il ne s'y insère pas, dans l'un des sacs à dos restants. Lorsqu'un article ne peut être inséré dans aucun sac à dos, l'algorithme tente, pour chaque paire de sacs à dos, des échanges entre les articles (un pour un, puis deux pour un, puis un pour deux), jusqu'à ce qu'un échange, qui utilise pleinement l'espace disponible dans un des sacs à dos, soit trouvé. Si tous les articles de $Z_s$ sont insérés, la solution optimale est alors trouvée, sinon, l'actuelle solution (sous-optimale) réalisable peut être améliorée à l'aide de l'algorithme glouton sur le sous-ensemble d'articles restants $N - Z_s$.

**L'heuristique (MTHM) de Martello et Toth**

Dans [MAR 81], Martello et Toth ont proposé une heuristique beaucoup plus efficace que celle présentée au paragraphe précédent. Cet algorithme nommé MTHM se déroule en quatre phases.

Les articles ainsi que les sacs à dos sont supposés triés suivant les relations (III.7) et (III.8).

La première phase de MTHM permet d'obtenir une solution en appliquant l'algorithme glouton (voir la *Procédure* 1 à la sous-section III.3.1, paragraphe a.) au premier sac; un ensemble d'articles résiduels est alors obtenu; puis la même procédure est appliquée au second sac à dos et ainsi de suite jusqu'au $m^{ème}$ sac à dos. Une nouvelle variable $y_j$, associée à chaque article et utilisée dans l'algorithme MTHM, est modifiée en même temps que $x_j$ comme suit :

$$y_j := \begin{cases} \text{indice du sac dans lequel à été affecté l'article } j, \\ 0 \text{ si l'article } j \text{ n'a pas été affecté.} \end{cases}$$

Dans la deuxième phase l'algorithme améliore la solution initiale à travers des échanges locaux. Premièrement, il considère toutes les paires d'articles affectées à différents sacs à dos et, si possible, les échanges devraient permettre l'insertion d'un nouvel article dans la solution.

Lorsque toutes les paires d'articles ont été considérées, la troisième phase de l'algorithme MTHM tente d'exclure à tour de rôle chaque article sélectionné, s'il est possible de le remplacer par un ou plusieurs articles restants, de sorte que le profit total soit augmenté.

Les tests expérimentaux ont montré que les échanges ont tendance à être beaucoup plus efficaces lorsque dans la solution initiale, obtenue avant la phase deux, chaque sac à dos contient des articles avec des ratios profit sur poids dissimilaires. Ceci n'est pas le cas pour la solution initiale déterminée à l'aide du glouton par la première phase. En effet, pour les premiers sacs à dos, les meilleurs objets sont d'abord insérés et, après que l'article critique correspondant à l'indice de rupture, voir la relation (III.20) ait été rencontré, généralement d'autres «bons» articles de poids faibles sont sélectionnés. Il s'ensuit que seuls de «mauvais» articles sont disponibles pour les derniers sacs à dos.

Les phases d'échanges dans l'algorithme MTHM sont alors précédées par une phase de réarrangement de la solution initiale. Cette phase est réalisée de la manière suivante : d'abord on retire des sacs à dos tous les articles de la solution initiale. Puis ceux-ci sont reconsidérés en fonction de leurs ratios profit sur poids croissant, en essayant d'assigner chaque article au sac à dos suivant, de façon cyclique. (De cette manière, les articles avec un faible poids sont considérés lorsque les capacités résiduelles sont petites).

L'avantage de l'heuristique MTHM est que certains articles peuvent être échangés d'un sac à l'autre ou retirés complètement si cela permet l'augmentation du profit total.

Cela peut conduire à une solution rapide et efficace lorsque la solution donnée par la phase de réarrangement est bonne. Le principal inconvénient de l'heuristique MTHM est qu'elle ne considère que les échanges entre paires d'articles au lieu de combinaisons d'articles.

L'heuristique présentée ci-après (cf. [LAL 12a]) donne de meilleures solutions réalisables dans un temps de calcul raisonnable en exploitant efficacement le noyau de chaque sac à dos.

## III.4 Une Nouvelle heuristique RCH pour le problème MKP

L'objectif de l'approche proposée consiste à charger un nombre maximal de «meilleurs» articles au regard du classement des articles, illustré par la relation (III.7), dans un faible temps de calcul. A cet effet, une borne inférieure efficace $\underline{z}$ du problème du MKP peut être construite comme suit :

Nous considérons le problème du *sac à dos multiple* comme une série de problèmes de sac à dos standards ($KP_i$):

$$(KP_i)\begin{cases} z_i(KP_i) = \max \sum_{j \in N_i} p_j x_j, \\ \text{s.c.} \sum_{j \in N_i} w_j x_j \le c_i, \\ x_j \in \{0,1\}, j \in N_i. \end{cases} \quad \text{(III.18)}$$

avec $i = 1, \ldots, m$, $N_1 = N$, $N_i = N_{i-1} - \{ l \in N_{i-1} \mid x_l = 1 \}$ et $\underline{z}_i$ est la borne inférieure du problème ($KP_i$) tel que $\underline{z}_i = \sum_{j \in N_i} p_j x_j$ .

Une borne inférieure du problème MKP est alors déduite comme suit :

$$\underline{z} = \sum_{i=1}^{m} \underline{z}_i . \quad \text{(III.19)}$$

Afin de simplifier la notation et sans perte de généralité, nous considérons que les articles de l'ensemble $N_i$ sont indexés de 1 à $Card(N_i) = n_i$, c'est-à-dire, $N_i = \{1, ..., n_i\}$.

Le principe de l'approche proposée peut être représenté à l'aide d'un exemple de problème MKP avec deux sacs. Les articles ainsi que les sacs seront classés suivant l'ordre donné par les inégalités (III.7) et (III.8).

On remplit le premier sac, qui est le plus petit puis le deuxième sac. La borne inferieure est alors déduite de la relation (III.19). Supposons maintenant que les deux sacs ne soient pas complètement remplis et qu'en « échangeant » quelques articles du premier sac avec quelques uns du deuxième sac, on arrive à mieux remplir le premier sac (et donc à diminuer l'espace restant dans ce dernier). La borne inferieure ne change pas mais l'espace restant dans le deuxième sac est plus grand. Par conséquent, en considérant les articles qui n'ont pas été chargés on ne peut qu'améliorer la borne inférieure.

L'échange des articles entre le premier et le deuxième sac se fait, dans notre cas, en résolvant un problème de type *Subset sum* sur un sous-ensemble d'articles appelé *noyau* du premier sac.

L'heuristique proposée est illustrée par la procédure *RCH* où on peut voir le déroulement des différentes étapes qui seront présentées par la suite.

## III.4.1 Remplissage classique : algorithme Glouton

**Recherche de la borne inférieure pour les m – 1 premiers sac à dos**

Les $m$ – 1 premiers problèmes de sac à dos ($KP_i$) sont traités successivement. A l'étape $i$, l'algorithme *glouton* illustré par la procédure 1 ci-dessous est exécuté pour remplir le sac à dos ($KP_i$). Ceci permet d'obtenir une borne inférieure $\underline{z}_i = \sum_{j \in N_i} p_j x_j$.

Si le sac est entièrement rempli, c.-à-d. la capacité résiduelle est nulle, on passe au sac suivant. Sinon on résout un problème du *Subset Sum* sur le noyau du sac à dos considéré et on obtient une nouvel valeur pour $\underline{z}_i$.

Enfin, la méthode de la programmation dynamique est utilisée pour obtenir une borne inférieure pour le dernier sac à dos ($KP_m$).

**Calcul d'une borne inférieure d'un sac à dos avec l'algorithme Glouton**

On suppose que les articles et les sacs à dos sont triés en fonction des inégalités (III.7) et (III.8), respectivement. L'algorithme glouton suivant est appliqué au sac ($KP_i$), $i \neq m$.

---

**Procédure 1**. Algorithme Glouton.

---

**Entrée:** $N_i$, $(p_j)$, $(w_j)$, $(x_j)$, $c_i$

**Sortie:** $\underline{z}_i$, $\bar{c}_i$;

$\underline{z}_i = 0$ ;

$\bar{c}_i = c_i$ ;

**for** $j := 1$ to $j := n_i$ **do**

**if** $w_j \leq \bar{c}_i$ **then**

$x_j := 1$ ;

$\bar{c}_i := \bar{c}_i - w_j$ ;

$\underline{z}_i := \underline{z}_i + p_j$ ;

**end if**

**end for**

**end ;**

---

A la fin de cette procédure, on obtient un premier remplissage du sac à dos $i$ avec une première borne inferieure $\underline{z}_i$ et une capacité résiduelle $\bar{c}_i$ (les deux sorties de la Procédure 1). La borne $\underline{z}_i$, qui est une partie de la solution, voir équation (III.19), sera intéressante si le sac à dos $i$ est complètement rempli. Dans le cas contraire, on essayera de réduire cette capacité résiduelle à l'étape suivante.

**Le problème du noyau du sac à dos ($KP_i$)**

Nous rappelons que les articles sont triés selon l'inégalité (III.7). Nous désignerons par $s$ l'indice de rupture ou de base du problème ($KP_i$) correspondant à l'inégalité suivante :

$$\sum_{j=1}^{s-1} w_j \leq c_i < \sum_{j=1}^{s} w_j . \qquad \text{(III.20)}$$

Le noyau du problème ($KP_i$) qui est un sous-ensemble d'articles noté $C_i$ est défini comme suit:

$$C_i = \left\{ j \in N_i \;\middle|\; s - r \leq j \leq s + r - 1 \right\}, i = 1,..., m, \qquad \text{(III.21)}$$

où $2r$ est une constante qui représente la taille du noyau $|C_i|$. Comme indiqué au chapitre II, plusieurs choix ont été proposés pour $|C_i|$. Pour l'heuristique proposée, de meilleurs résultats ont été obtenus avec $|C_i| = 2\sqrt{n}$ soit $r = \sqrt{n}$ .

## III.4.2 Remplissage efficace : utilisation du noyau

**La résolution du problème Subset Sum ($SSP_i$)**

Un problème de la somme d'un sous-ensemble, «S*ubset sum problem*» en anglais (SSP) est un cas particulier du problème de sac à dos où les profits sont identiques aux poids. La formulation de ce problème est donnée ci-dessous.

$$(SSP)\begin{cases} \max \sum_{j=1}^{n} w_j . x_j , \\ s.c. \sum_{j=1}^{n} w_j . x_j \leq c, \\ x_j \in \{0,1\}, j \in \{1,..,n\}. \end{cases} \qquad \text{(III.22)}$$

Ce problème a de nombreuses applications. Par exemple Diffie et Hellman [DIF 76] ont conçu un système de cryptographie publique dont la sécurité repose sur la difficulté à résoudre des problèmes SSP. Martello et Toth [MAR 82], ont mentionné que le problème d'ordonnancement sur deux machines est un problème du *Subset Sum*.

Les solutions du problème SSP peuvent aussi être utilisées pour obtenir des bornes inférieures de bonne qualité pour des problèmes de planification (Guéret et Prins [GUE 99]).

Dans le cas de l'heuristique RCH, nous résolvons un problème du *Subset Sum* ($SSP_i$) restreint au noyau $C_i$ afin de réduire la capacité résiduelle $\overline{c}_i$ du problème de sac à dos ($KP_i$).

Par la suite, nous utilisons la nouvelle capacité résiduelle $\overline{c}_i = c_i - \sum_{j=1}^{s-r-1} w_j$.

Alors nous résolvons le problème suivant:

$$(SSP_i)\begin{cases} z(SSP_i) = \max \sum_{j \in C_i} w_j x_j, \\ \text{s.c.} \sum_{j \in C_i} w_j x_j \le \overline{c}_i, \\ x_j \in \{0,1\}. \end{cases} \qquad \text{(III.23)}$$

Pour résoudre ce problème ($SSP_i$), notre choix s'est porté sur la méthode Elkihel [ELK 84]. Nous désignerons par $z(SSP_i)$ la valeur optimale du problème ($SSP_i$). Dans la méthode de Elkihel [ELK 84], les articles sont pris alternativement de part et d'autre de l'indice $s$, c'est-à-dire que nous considérons les articles $s$, $s-1$, $s+1$, $s-2$, ... (voir aussi [PIS 95] et [MAR 97]). Plus précisément, la méthode consiste à construire par itération les listes $L_1^k$ et $L_2^q$ comme suit :

$$L_1^k = \left\{ w \;\middle|\; w = \sum_{j=k}^{s-1} w_j x_j \le \overline{c}_i, x_j \in \{0,1\} \right\}, k = s-1, s-2, ..., s-r, \qquad \text{(III.24)}$$

$$L_2^q = \left\{ w' \;\middle|\; w' = \sum_{j=s}^{q} w_j x_j \le \overline{c}_i, x_j \in \{0,1\} \right\}, q = s, s+1, ..., s+r-1. \qquad \text{(III.25)}$$

Les états sont classés par poids $w$ croissants dans les listes. La technique de dominance est appliquée. Les listes $L_1^k$ et $L_2^q$ sont fusionnées deux par deux après chaque étape.

Le processus de construction des listes est arrêté lorsqu'une des deux conditions suivantes: $\max\ w + w' = \overline{c}_i$ ou $k = s - r$ et $q = s + r - 1$, est réalisée. Ainsi, nous avons :

$$\underline{z}_i = \sum_{j=1}^{s-r-1} p_j + \sum_{j \in C_i} p_j x_j^*, \tag{III.26}$$

où les $x_j^*$ correspondent à la solution du problème *SSP*, c'est-à-dire $z(SSP) = \sum_{j \in C_i} w_j x_j^*$ .

Nous pouvons écrire cet algorithme comme suit:

---

**Procédure 2**. Résoudre $SSP_i$.

---

**Entrée:** $C$, $(w_j)$, $s$, $c_i$, $r$ ;

**Sortie:** $N_i+1$, $\underline{z}_i$ ;

**For** $l$ := 1 to l := $r$ **do**

$k := s - l$ ;

$q := s + l - 1$ ;

Construire $L_1^k$ ;

Construire $L_2^q$ ;

Fusionner $L_1^k$ et $L_2^q$;

**If** max $w + w' = \overline{c}_i$ **then** STOP

**End for**

Mise à jour $\underline{z}_i$ ;

Mise à jour $N_{i+1}$ ;

**End**

---

**Recherche de la borne inférieure pour le dernier sac à dos**

La programmation dynamique est appliquée sur le noyau du dernier sac à dos ($KPC_m$). Le problème est résolu ainsi

$$(KPC_m)\begin{cases} z(KPC_m) = \max \sum_{j \in C_m} p_j x_j \\ \text{s.c.} \sum_{j \in C_m} w_j x_j \le \bar{c}_m, \\ x_j \in \{0,1\}. \end{cases} \quad \text{(III.27)}$$

Nous désignerons par $z(KPC_m)$ la valeur optimale du problème ($KPC_m$). Le problème ($KPC_m$) est résolu par la méthode de programmation dynamique en utilisant la technique de dominance (voir [AHR 75], [ELB 05] et [BOY 09]). Dans cette méthode, les articles sont considérés successivement de $s - r$ à $s + r - 1$. Plus précisément, la méthode de programmation dynamique consiste à construire itérativement les listes $L^k$ comme suit :

$$L^k = \left\{ (w,p) \,\middle|\, w = \sum_{j=s-r}^{k} w_j x_j \le \bar{c}_m, x_j \in \{0,1\} \text{ et } p = \sum_{j=s-r}^{k} p_j x_j \right\},\ k = s-r,\ldots,s+r-1. \quad \text{(III.28)}$$

Ici, les états sont classés dans les listes $L_k$ dans l'ordre des profits $p$ croissants. La technique de dominance est appliquée : si $(w, p)$ et $(w', p')$ sont deux états tels que $w \le w'$ et $p \ge p'$, alors l'état $(w', p')$ est dominé et doit être retiré.

On associe à la programmation dynamique, une méthode de réduction de variable (voir la sous-section II.4.3) utilisant la borne supérieure $U_2$ de Martello et Toth (voir la sous-section II.4.1).

Le temps de résolution du problème (KP) par la programmation dynamique est notamment lié au classement de variables adopté. D'après la sous-section II.4.3, les variables susceptibles d'être fixées sont celles qui présentent les bornes $U_j$, $j \in \{1,\ldots,n\}$, les plus petites. Il n'est donc pas très intéressant de traiter ces variables en premier par la programmation dynamique car avec l'amélioration de la borne, elles ont une forte probabilité d'être fixées par le processus de réduction. Par conséquent, en classant les variables selon les $U_j$ décroissants, nous avons le schéma de résolution suivant :

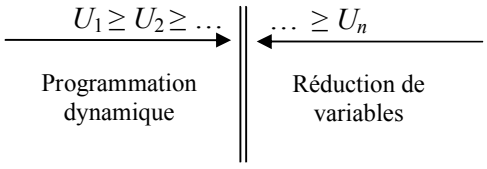



FIGURE III.1 – Déroulement de l'algorithme associant la réduction de variables à la méthode de programmation dynamique.

Nous obtenons alors :

$$\underline{z}_m = \sum_{j=1}^{s-r-1} p_j + z(KPC_m), \qquad \text{(III.29)}$$

avec $z(KPC_m) = \sum_{j \in C} p_j x_j^* = \max \left\{ p \mid (w, p) \in L_{s+r-1} \right\}$ . La procédure de calcul est détaillée dans l'algorithme suivant :

---

**Procédure 3**. Résolution de problèmes KP.

---

**Entrée:** *C,* (*wj*), (*pj*)*, s, cm, r* ;

**Sortie:** $\underline{z}_m$ ;

**For** $k := s - r$ **to** $l := s + r - 1$ **do**

Construire $L_k$ ;

**end for**

Mise à jour de $\underline{z}_m$ ;

**End**

---

L'heuristique proposée est illustrée par la Procédure *RCH* où on peut voir le déroulement des différentes étapes qui ont été présentées précédemment.

## III.5 Résultats expérimentaux

Nous présentons les résultats expérimentaux pour l'heuristique RCH ainsi que la comparaison entre cette dernière, l'heuristique MTHM et le solveur CPLEX version 12.1. Les algorithmes RCH et MTHM ont été écrits en langage C. Tous les tests ont été effectués sur un processeur Intel Core 2 Duo 2,2 GHz avec 2 Go de RAM. Nous avons traité des problèmes engendrés aléatoirement.

Les résultats présentés ont été obtenus sur une moyenne de 10 instances par problème. Nous avons fixé une limite de temps de 600 secondes (10 minutes) pour chaque instance.

**Procédure** *RCH***.**

**Entrée:** $N$, $(p_j)$, $(w_j)$, $(x_j)$, $(c_i)$

**Sortie:** $\underline{z}$ ;

$\underline{z} = 0$ ;
/*Traitement des $m - 1$ premiers sac à dos*/
**for** $i := 1$ to $m-1$ **do** ;

$\underline{z}_i = 0$ ;
$\overline{c}_i = c_i$ ;
**Procédure 1** sur $c_i$ ;

**if** $\overline{c}_i > 0$ **then**

Définir le noyau $C_i$ de $c_i$ ;

**Procédure 2** sur $SSP_i$ ;

**end if**

Mise à jour de $N$ ;

$\underline{z} := \underline{z} + \underline{z}_i$ ;

**end for**

/*Traitement du dernier sac à dos*/

**Procédure 1** sur $c_m$ ;

**if** $\overline{c}_i > 0$ **then**

Définir le noyau $C_m$ de $c_m$ ;

**Procédure 3** sur $KPC_m$ ;

**end if**

$\underline{z} := \underline{z} + \underline{z}_m$ ;

**end ;**

Nous avons traité trois types de problèmes : non corrélés, faiblement corrélés et fortement corrélés ; ce qui nous a permis de considérer des problèmes relativement faciles à très difficiles.

Les poids $w_j$ sont répartis aléatoirement sur l'intervalle des entiers [1, 1000] et les profits $p_j$ sont calculés comme suit :

- Dans le cas de problèmes non corrélés : uniformément aléatoire sur l'intervalle [1, 1000], indépendants de $w_j$.
- Dans le cas de problèmes faiblement corrélés : uniformément aléatoire sur l'intervalle $[w_j - 100, w_j + 100]$.
- Dans le cas de problèmes fortement corrélés : $p_j = w_j + 100$.

Les capacités $c_i$ sont prises de manière uniformément aléatoire dans l'intervalle

$$\left[ 0.4 \frac{\sum_{j=1}^{n} w_j}{m}, 0.6 \frac{\sum_{j=1}^{n} w_j}{m} \right] \text{ pour } i = 1, \ldots, m-1.$$

La capacité du $m^{ème}$ est :

$$c_m = 0.5 \sum_{j=1}^{n} w_j - \sum_{i=1}^{m-1} c_i .$$

Si une instance ne satisfait pas les conditions (III.4) – (III.6), alors une nouvelle instance est engendrée. Les problèmes MKP sont disponibles à l'adresse [LAA 09].

Par la suite, les *gaps* présentés sont calculés comme suit :

$$gap = \frac{100 \ (U - \underline{z})}{U},$$

où $U$ est la meilleure borne supérieure donnée par CPLEX et la relaxation surrogate continue du problème MKP (voir [MAR 81]).

Nous avons testé différents ordres de rangement des articles dans le noyau, à savoir l'ordre naturel selon lequel les articles sont triés par ratio profit sur poids décroissant, voir (III.7) et un ordre différent où les articles sont triés dans l'ordre des poids décroissants, i.e. $w_{s-r} \geq w_{s-r+1} \geq \ldots \geq w_{s-1}$, $w_s \geq w_{s+1} \geq \ldots \geq w_{s+r}$. Nous avons noté que les deux ordres donnent de bons résultats en termes de *gaps*. Néanmoins, le dernier ordre testé s'avère légèrement meilleur en termes de temps de calcul; c'est cet ordre là que nous avons utilisé dans les tests que nous présentons dans ce mémoire.

| *m* | *n* | CPLEX | | MTHM | | RCH | | *U* |
|---|---|---|---|---|---|---|---|---|
| | | Time(s) | GAP(%) | Time(s) | GAP(%) | Time(s) | GAP(%) | |
| 2 | 5 000 | 4.5200 | 0.00000 | 0.1438 | 0.00063 | 0.0064 | 0.00005 | ○ |
| | 10 000 | 7.7600 | 0.00000 | 0.5751 | 0.00023 | 0.0079 | 0.00002 | ○ |
| | 50 000 | 29.010 | 0.00000 | 14.546 | 0.00002 | 0.0157 | 0.00000 | ○ |
| | 100 000 | 138.32 | 0.00000 | 58.530 | 0.00001 | 0.0220 | 0.00000 | ○ |
| 10 | 5 000 | 600.00 | 0.01076 | 0.1766 | 0.00204 | 0.0110 | 0.00020 | ○ |
| | 10 000 | 600.00 | 0.00864 | 0.6842 | 0.00065 | 0.0125 | 0.00011 | ○ |
| | 50 000 | 600.00 | 0.00454 | 17.116 | 0.00006 | 0.0265 | 0.00001 | ○ |
| | 100 000 | 600.00 | 0.00850 | 69.091 | 0.00003 | 0.0359 | 0.00000 | ○ |
| 100 | 5 000 | 600.00 | 0.12789 | 0.1860 | 0.02493 | 0.0657 | 0.00193 | ○ |
| | 10 000 | 600.00 | 0.69317 | 0.7341 | 0.00722 | 0.0828 | 0.00058 | ○ |
| | 50 000 | – | – | 18.016 | 0.00051 | 0.0359 | 0.00008 | ● |
| | 100 000 | – | – | 71.758 | 0.00016 | 0.2704 | 0.00003 | ● |

*U*, la meilleure borne supérieure donnée par : ○ CPLEX.
● la relaxation surrogate continue du *MKP*.
– : CPLEX dépasse la capacité mémoire de la machine.

TABLEAU III.1 – Temps de calcul et *gaps* pour des problèmes non corrélés.

| *m* | *n* | CPLEX | | MTHM | | RCH | | *U* |
|---|---|---|---|---|---|---|---|---|
| | | Time(s) | GAP(%) | Time(s) | GAP(%) | Time(s) | GAP(%) | |
| 2 | 5 000 | 5.1100 | 0.00000 | 0.1139 | 0.00067 | 0.0093 | 0.00000 | ○ |
| | 10 000 | 8.9600 | 0.00000 | 0.4530 | 0.00025 | 0.0108 | 0.00000 | ○ |
| | 50 000 | 31.450 | 0.00000 | 11.359 | 0.00002 | 0.0157 | 0.00000 | ○ |
| | 100 000 | 143.27 | 0.00000 | 42.371 | 0.00000 | 0.0154 | 0.00000 | ○ |
| 10 | 5 000 | 600.00 | 0.01249 | 0.1375 | 0.00541 | 0.0157 | 0.00072 | ○ |
| | 10 000 | 600.00 | 0.00623 | 0.5422 | 0.00174 | 0.0078 | 0.00008 | ○ |
| | 50 000 | 600.00 | 0.00149 | 13.568 | 0.00012 | 0.0375 | 0.00000 | ○ |
| | 100 000 | 600.00 | 0.00544 | 54.250 | 0.00005 | 0.0420 | 0.00000 | ○ |
| 100 | 5 000 | 600.00 | 0.15531 | 0.1453 | 0.09919 | 0.0766 | 0.00072 | ○ |
| | 10 000 | 600.00 | 0.55200 | 0.5658 | 0.03841 | 0.0888 | 0.00016 | ○ |
| | 50 000 | – | – | 13.879 | 0.00301 | 0.2092 | 0.00001 | ● |
| | 100 000 | – | – | 55.446 | 0.00088 | 0.3219 | 0.00000 | ● |

*U*, la meilleure borne supérieure donnée par : ○ CPLEX.
● la relaxation surrogate continue du *MKP*.
– : CPLEX dépasse la capacité mémoire de la machine.

TABLEAU III.2 – Temps de calcul et *gaps* pour les problèmes faiblement corrélés.

| *m* | *n* | CPLEX | | MTHM | | RCH | | *U* |
|---|---|---|---|---|---|---|---|---|
| | | Time(s) | GAP(%) | Time(s) | GAP(%) | Time(s) | GAP(%) | |
| 2 | 5 000 | 5.8700 | 0.00000 | 0.1609 | 0.00000 | 0.1095 | 0.00000 | ○ |
| | 10 000 | 9.3700 | 0.00000 | 0.6625 | 0.00000 | 0.2079 | 0.00000 | ○ |
| | 50 000 | 38.780 | 0.00000 | 15.995 | 0.00000 | 1.0127 | 0.00000 | ○ |
| | 100 000 | 146.54 | 0.00000 | 65.911 | 0.00000 | 2.0140 | 0.00000 | ○ |
| 10 | 5 000 | 600.00 | 0.01924 | 0.1919 | 0.00173 | 0.1235 | 0.00087 | ○ |
| | 10 000 | 600.00 | 0.01755 | 0.7626 | 0.00250 | 0.2267 | 0.00171 | ○ |
| | 50 000 | 600.00 | 0.01000 | 19.185 | 0.00023 | 1.0533 | 0.00021 | ○ |
| | 100 000 | 600.00 | 0.00556 | 76.602 | 0.00010 | 2.0406 | 0.00003 | ○ |
| 100 | 5 000 | 600.00 | 0.46629 | 0.2030 | 0.18344 | 0.7330 | 0.04310 | ○ |
| | 10 000 | 600.00 | 0.56466 | 0.8143 | 0.07015 | 1.3720 | 0.00457 | ○ |
| | 50 000 | – | – | 20.109 | 0.00306 | 1.8797 | 0.00025 | ● |
| | 100 000 | – | – | 80.058 | 0.00096 | 2.8470 | 0.00013 | ● |

*U*, la meilleure borne supérieure donnée par : ○ CPLEX.
● la relaxation surrogate continue du *MKP*.
– : CPLEX dépasse la capacité mémoire de la machine.

TABLEAU III.3 – Temps de calcul et *gaps* pour des problèmes fortement corrélés.

Les Tableaux III.1 à III.3 illustrent les résultats des calculs pour les différents types de problèmes considérés à savoir non corrélés, faiblement corrélés et fortement corrélés.

Au regard des résultats présentés, nous pouvons remarquer que l'heuristique RCH est en général meilleure que CPLEX et aussi de l'heuristique MTHM, en termes de gaps et de temps de calcul.

On peut voir que le temps de calcul de l'heuristique RCH n'est pas très sensible au nombre *m* de sacs et à la taille des problèmes *n*. Nous rappelons que la taille des noyaux est égale à $2\sqrt{n}$. On peut remarquer aussi que plus *n* est grand, meilleur est le *gap*. Nous notons que pour 10 et 100 sacs à dos, CPLEX dépasse la limite de temps de 10 minutes et la différence de *gaps* entre CPLEX et RCH peut atteindre 3 ordres de grandeurs, voir exemple à la Table III.1 ( pour $m = 100$ et $n = 5000$).

Nous notons également que pour des problèmes avec 100 sacs à dos et plus de 50 000 articles, CPLEX dépasse la capacité mémoire de la machine.

Le temps de calcul de la méthode de RCH ne dépasse jamais 3 secondes, alors qu'il peut atteindre 80 secondes avec la méthode MTHM.

Les bons résultats obtenus avec RCH en comparaison avec MTHM peuvent être expliqués comme suit :

- L'échange d'une combinaison d'articles entre deux sacs à dos contigus par la méthode récursive proposée dans l'heuristique RCH est plus efficace que l'échange de paires d'articles entre différents sacs comme dans l'heuristique MTHM.
- La procédure de construction de listes de l'heuristique RCH combine des articles du noyau, de plus faible poids, c'est à dire à gauche de l'indice *s*, avec des articles de plus fort poids, c'est à dire à droite de *s*. Il en résulte une meilleure façon de remplir les sacs.

## III.6 Conclusions et perspectives

Dans ce chapitre, nous avons proposé une heuristique RCH pour le problème du MKP qui consiste à résoudre récursivement chaque noyau des différents sacs à dos. Pour les $m - 1$ premiers noyaux, un problème de *Subset sum* est résolu. Tandis que le dernier noyau est résolu en tant que KP. Dans les deux cas, la méthode de programmation dynamique est utilisée. L'ordre selon lequel les articles sont triés : par poids décroissants dans les deux ensembles $\{s - r, ..., s - 1\}$ et $\{s, ..., s + r - 1\}$ des $m - 1$ premiers noyaux, et l'ordre naturel dans le dernier noyau, a été adopté. Les résultats des expérimentations montrent que l'heuristique RCH aboutit généralement à de meilleurs *gaps* comparée à l'heuristique MTHM de Martello et Toth ou au solveur CPLEX et ceci dans de meilleurs temps de calcul. Cela démontre que l'approche proposée permet de résoudre des problèmes de grandes tailles dans un temps de calcul raisonnable.

# Chapitre IV

# Introduction à CUDA et à l'architecture GPU

## Sommaire



Dans ce chapitre, nous présentons de façon succincte les notions de base de l'architecture GPU et de la programmation CUDA. Pour plus de détails on se reportera à [CUV 11] et [NVI 12].

# IV.1 Introduction

Des problèmes, comme la simulation de phénomènes physiques, biologiques ou chimiques de grande taille, nécessitent des performances que seules les machines massivement parallèles peuvent offrir.

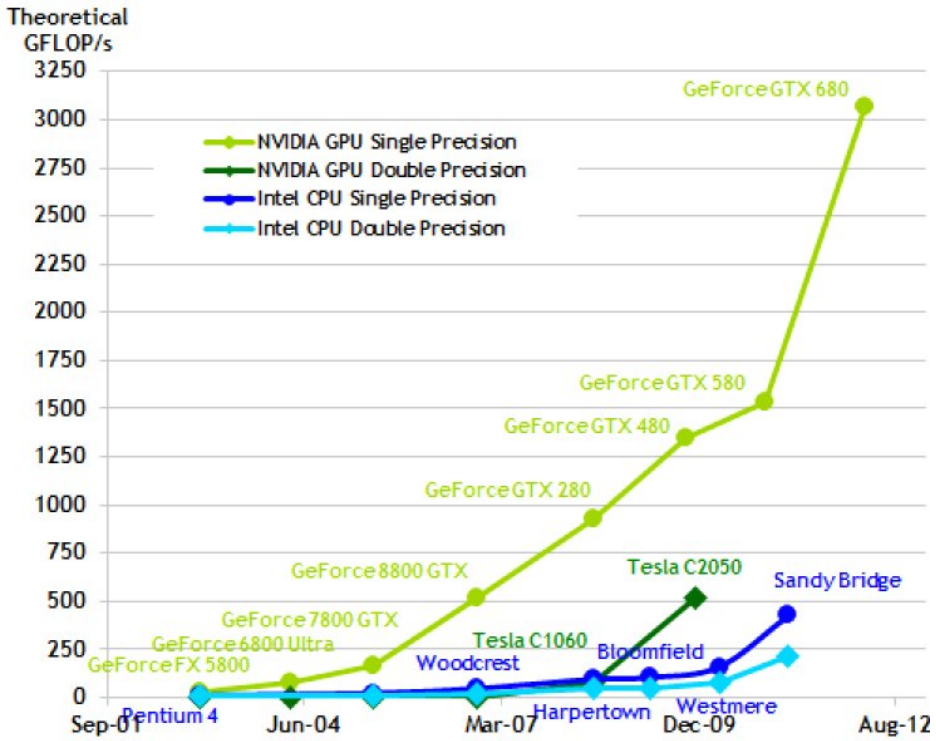



FIGURE IV.1 – Evolution des performances de calcul des CPUs et GPUs (NVIDIA).

Les GPUs (*Graphics Processing Unit*), à l'origine dédiés aux applications graphiques des machines informatiques, ont évolué ces dernières années vers de puissantes architectures parallèles multi-cœurs avec des performances de calculs qui ne cessent d'augmenter (voir Figure. IV.1) pour des applications généralistes. Néanmoins, leur programmation demeure plus complexe que celle des machines séquentielles et des supercalculateurs.

Ainsi, à la différence des CPUs, les GPUs sont orientés vers le calcul hautement parallèle : de par leur conception, ils dédient davantage de transistors au calcul (voir Figure. IV.2), au détriment du stockage et de la gestion des flux de données.

L'architecture des GPUs qui s'appuie sur des multiprocesseurs SIMT (*Single-Instruction, Multiple Threads* c'est-à-dire *une instruction, plusieurs threads*) s'apparente à une architecture SMID (*Single Instruction Multiple Data* c'est-à-dire *une instruction, plusieurs*

*données*). La principale différence entre ces deux architectures est la gestion avancée de tâches permettant des changements rapides de contexte dans le cas des architectures de type SIMT. Ces dernières sont capables d'exécuter plusieurs tâches effectuant la même opération en même temps.

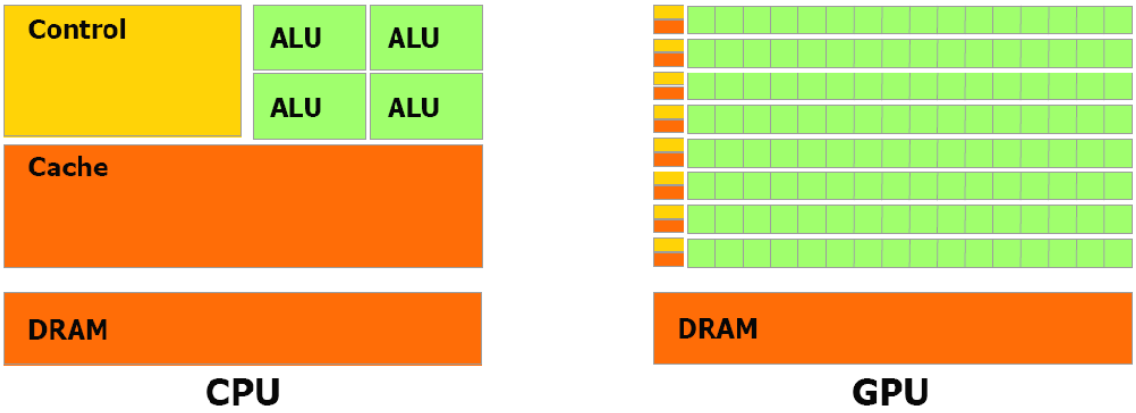



FIGURE IV.2 – Architecture CPU et architecture GPU (NVIDIA).

Les constructeurs de GPUs se sont intéressés à créer des langages, en mesure de tirer profit des possibilités des GPUs. L'arrivée de l'environnement de développement CUDA de NVIDIA [NVI 12] a marqué une avancée significative pour l'utilisation du GPU pour des tâches de calcul polyvalentes de sciences et d'ingénierie. Cet environnement combine un modèle de programmation relativement accessible au travers d'un langage niveau C, et une architecture matérielle permettant les lectures et écritures mémoire arbitraires et offrant un accès à des mémoires locales et des synchronisations entre threads.

L'API CAL d'AMD [AMD 10] est basée sur des concepts similaires, elle fournit un socle sur lequel peuvent s'appuyer des langages de plus haut niveau tels que Brook+, une version reprise par AMD du projet Brook.

Ces API propriétaires ont inspiré des normes permettant la portabilité des applications entre les GPUs des différents constructeurs. L'environnement OpenCL ([OPE 12a] et [OPE 12b]) du consortium Khronos établit une norme multi-plate-forme principalement inspirée par CUDA. Tous ces environnements ont pour point commun d'être basés sur un modèle de programmation similaire. Ce modèle tire son inspiration du modèle Bulk-Synchronous Programming (BSP), qui considère des sections parallèles (où les threads sont indépendants) séparées par des barrières de synchronisation globales.

## IV.2 Algorithmes et applications

Avec l'émergence de langages de programmation standards pour le GPU et l'arrivée de compilateurs pour ces langages, la mise en œuvre d'algorithmes de haut niveau qui exploitent la puissance de calcul des GPUs a connu un intérêt croissant.

Nous donnons ici un bref aperçu des algorithmes de calcul sur GPU et de leurs domaines d'applications; pour plus de détails nous renvoyons le lecteur à la référence [OWE 07] ainsi qu'au site CUDA Zone [CUDA].

### Traitement du Signal

Motivés par les capacités arithmétiques élevées des GPUs modernes, plusieurs mises en œuvre sur GPU de la méthode de la transformée de Fourier rapide (FFT) ont été proposées (cf. [BUC 04], [SUM 05], [GOV 08] et [NUK 12]). Daniel Horn à mis en ligne en 2005 la bibliothèque open-source brookGPU contenant une mise en œuvre optimisée de la méthode FFT sur GPU (cf. [HOR 05]) basée sur le langage Brook [BROOK]. La transformée en ondelettes discrète DWT utilisée dans la norme JPEG2000, est une autre opération fondamentale de traitement du signal dont la mise en œuvre GPU à été proposée dans [WON 07] et [SUX 08]. On peut noter aussi les mises en œuvre sur GPU du filtrage numérique proposé par Smirnov et Chiueh dans [SMI 05] et Rebacz et al. dans [REB 10].

Kerr et Campbell ont présenté dans [KER 08] la libraire GPU VSIPL, une mise en œuvre via CUDA de la libraire VSIPL (Vector Signal Image Processing Library), adaptée aux cartes graphiques NVDIA. La librairie GPU VSIPL permet ainsi d'utiliser les différentes fonctions du traitement du signal tel que : la transformée de Fourier rapide FFT, la transformée en ondelettes DWT, le filtrage numérique, les produits de convolution en obtenant des accélérations de un à deux ordres de grandeur comparé à VSIPL. NVIDIA fournit aussi CuFFT, une librairie optimisée de transformées de Fourier (rapide, discrète, 2D, 3D) sur GPU via CUDA.

### Algèbre linéaire

Les routines de l'algèbre linéaire, dense et creuse, constituent une base de construction d'un grand nombre d'algorithmes numériques, y compris de nombreux solveurs EDP mentionnés ci-dessus.

Les applications de ces algorithmes sont relatives à la simulation des effets physiques (tels que les fluides, la chaleur et le rayonnement). Un exemple représentatif est le travail de Krüger et Westermann [KRU 03], qui portait sur une large classe de problèmes d'algèbre linéaire en mettant l'accent sur la représentation des matrices et des vecteurs dans les GPUs (par exemple l'allocation de matrices denses et creuses en mémoire de textures). On note aussi l'article de Fatahalian et al. [FAT 04] présentant une analyse de multiplication de matrices denses sur GPU, l'article de Gallapo et al. [GAL 05] proposant un solveur sur GPU pour les systèmes linéaires denses capable de surpasser les implémentations ATLAS qui sont déjà hautement optimisées.

L'utilisation de CUDA simplifie et améliore les performances des différentes mises en œuvre d'algorithmes d'algèbre linéaire sur GPU. NVIDIA fournit CUBLAS [CUBLAS], une libraire d'algèbre linéaire pour CUDA respectant les conventions BLAS.

## Algorithme de tri

Les GPUs s'avèrent être très efficaces pour le tri de données si bien que la communauté du calcul sur GPU a adaptée et amélioré les algorithmes de tri existants par exemple l'algorithme de tri bitonique [BAT 68]. Ce réseau de tri est basé sur le principe du tri par fusion deux à deux de listes. Govindaraju et al. ont remporté le prix de la catégorie « PennySort » de la compétition « TeraSort » de 2005 [GOV 05a] en implémentant l'algorithme de tri Bitonic sur GPU toute en apportant de nombreuses améliorations algorithmiques. D'autres algorithmes de tri ont été mis en œuvre avec succès sur GPU, à savoir le tri « Quicksort » [CED 09] et le tri « Radix » [SAT 09].

## Les requêtes de base de données et de recherche

Les chercheurs ont également mis en œuvre plusieurs formes de recherche sur GPU, comme la recherche binaire [HOR 05], la recherche du plus proche voisin [BUS 06] et [QIU 09] ainsi que les opérations de bases de données de haute performance et les algorithmes de tri rapides mentionnés ci-dessus [GOV 04], [GOV 05b].

## Equations différentielles

Les premières tentatives afin d'utiliser les GPUs pour le calcul scientifique se sont focalisées sur la résolution des grands systèmes d'équations différentielles. On pourra citer le tracé de particules [KRU 05]. Les GPUs ont aussi été utilisés pour résoudre des problèmes d'équations

aux dérivées partielles (EDP) tels que les équations de Navier-Stokes pour un écoulement de fluide incompressible. On se référera pour la dynamique des fluides à [BOL 03], [ZAH 07] et [EGL 10] et pour la segmentation de volume à [LEF 03] et [LEE 07].

## Optimisation combinatoire

En raison de leur nature parallèle, les métaheuristiques ont été les premières méthodes à être mises en œuvre sur des architectures GPU.

Un des premiers travaux pionniers sur les algorithmes génétiques a été proposé par Wong et al. dans [WON 05], [WON 06] et [FOK 07]. Dans ces travaux, les auteurs proposent une mise en œuvre CPU-GPU qui comprend un transfert important de données entre le CPU et le GPU. Yu et Al [QIZ 05] ont été les premiers auteurs à proposer une parallélisation d'un algorithme génétique entièrement sur GPU. Plus tard, Luo et Al. [LUO 06] ont été parmi les premiers auteurs à mettre en œuvre un algorithme génétique cellulaire sur GPU pour le problème 3-SAT. Pour effectuer cela, les sémantiques de l'algorithme cellulaire originel sont complètement modifiées pour être adaptées aux contraintes GPU.

Les travaux cités ci-dessus, ont été mis en œuvre sur GPU à l'aide de bibliothèques basées sur Direct3D ou OpenGL. D'autres mises en œuvre sur GPU, utilisant l'outil de développement CUDA ont été proposées. Ainsi Zhu a suggéré dans [ZHU 09] un algorithme de stratégie d'évolution puis Arora et al. ont présenté une mise en œuvre similaire dans [ARO 10] pour les algorithmes génétiques.

Tsutsui et al. ont élaboré des concepts de gestion de mémoire de problèmes d'optimisation combinatoire [TSU 09]. Dans leur mises en œuvre du problème d'affectation quadratique, les accès à la mémoire globale sont coalescents, la mémoire partagée est utilisée pour stocker autant d'individus que possible et les matrices sont associées à la mémoire constante.

Janiak et Al. ont mis en œuvre un algorithme de recherche tabou appliqué au problème du voyageur de commerce et au problème d'ordonnancement flowshop [JAN 08].

Luong et al. ont proposé dans [LUO 10] et [LUO 11] une méthodologie générale pour la conception d'algorithmes multi-démarrage applicable à tous algorithmes de recherche locale tels que la recherche tabou ou le recuit simulé ainsi que les premiers algorithmes multicritères de recherche locale sur GPU.

Boyer et al. ont proposé une mise en œuvre GPU [BOY 11] et Multi-GPU [BOY 12] de l'algorithme de la programmation dynamique pour la résolution de problème du sac à dos.

Chakroun et al. ont proposé dans [CHA 11] de résoudre des problèmes de Flow-shop sur GPU via l'algorithme de Branch and Bound.

Nous avons proposé quant à nous une mise en œuvre GPU de l'algorithme de Branch and Bound sur GPU pour le problème du sac à dos (voir [BOU 12] et [LAL 12b]). Cette mise en œuvre sera détaillée au chapitre VI.

## IV.3 Architecture GPU

En 2006, NVIDIA a proposé CUDA (*Compute Unified Device Architecture*), un environnement de programmation parallèle qui s'apparente à un logiciel permettant de manipuler de façon transparente la puissance d'une carte graphique composée de plusieurs multiprocesseurs, tout en étant facile de prise en main pour des utilisateurs familiers avec le langage C. Les langages C, C++ et Fortran peuvent être utilisés conjointement dans les fonctions (qui sont encore appelées *kernels*), voir la Figure IV.3.

La majorité des cartes graphiques grand public NVIDIA sont compatibles CUDA. CUDA s'est imposé comme un des principaux langages standards de manipulation des GPUs.

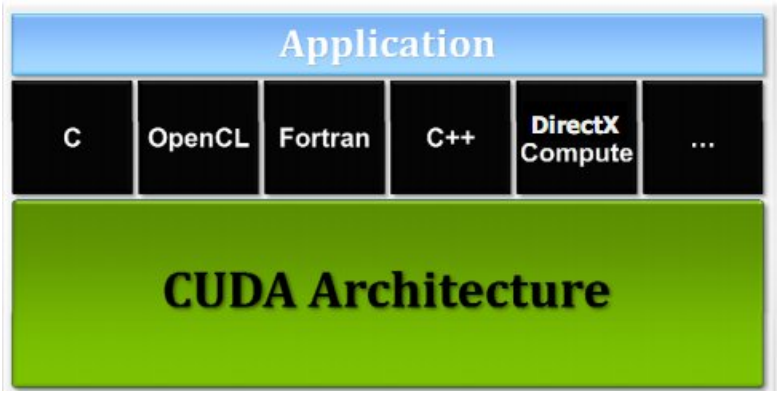



FIGURE IV.3 – Architecture des applications CUDA (NVIDIA).

Trois notions essentielles sont à la base de CUDA : une hiérarchie de groupes de threads, des mémoires partagées et une barrière de synchronisation. Celles-ci sont mises à la disposition du programmeur à travers des extensions, du langage C.

Ces notions donnent la possibilité au programmeur de gérer avec précision le parallélisme offert par la carte graphique. Elles permettent à celui-ci de partitionner le problème en sous-problèmes pouvant être résolus de manière indépendante et parallèle entre un groupe de threads (appelé *bloc*).

Les extensions du langage C apportées par CUDA permettent au programmeur de définir des fonctions appelées « *kernels* », qui se dérouleront en parallèle sur la carte graphique. Ces fonctions sont exécutées *N* fois par un nombre *N* de threads CUDA.

Un *kernel* est déclaré comme une simple fonction C, à l'exception faite du préfixe : *__global__*. D'autre part, pour chaque appel au *kernel*, le nombre de threads est spécifié en utilisant la syntaxe <<< … >>>, comme suit :

```
/* Déclaration du  kernel */
__global__ void Nom_du_kernel(parametres) {Le code à
exécuter;}

/* Programme principal */
int main()
{
/* Appel du kernel */
Nom_du_kernel <<<1, N>>> (parametre);
}
```

Chaque thread, exécutant le code d'un *kernel*, se voit attribuer un identifiant à travers la variable standard « *threadIdx* ». Cette identifiant est accessible à l'intérieur du *kernel*. A titre d'exemple, le code suivant réalise l'addition de deux vecteurs A et B, de dimensions N, et place le résultat dans le vecteur C :

```
__global__ void vecAdd(float* A, float* B, float* C)
{
int i = threadIdx.x;
C[i] = A[i] + B[i];
}

/* Programme principal */
int main()
{
vecAdd<<<1, N>>>(A, B, C);
}
```

### IV.3.1 Les threads

La variable standard *threadIdx* évoquée au paragraphe précédent, est un vecteur de trois composantes. Par conséquent, les *threads* peuvent être identifiés à travers une dimension, deux dimensions, ou trois dimensions formant ainsi des blocs. Ces derniers peuvent être, à leurs tours identifiés à travers une dimension, deux dimensions, ou trois dimensions formant une grille (voir Figure IV.4). Cette caractéristique prend tout son sens dès lors que l'on manipule des tenseurs d'ordre 2 ou 3.

Les *threads* d'un même *bloc* peuvent communiquer en partageant leurs données via la *mémoire partagée* et se synchroniser pour coordonner leurs accès mémoire. Concrètement, le programmeur peut ordonner une synchronisation des *threads* en employant l'instruction *__syncthreads()*.

Cette fonction intrinsèque agit comme une barrière pour les *threads* d'un même *bloc*, barrière qu'aucun d'eux ne pourra franchir seul. En d'autres termes, un *thread* rencontrant cette instruction devra attendre que tous les autres *threads* l'aient atteinte avant de poursuivre.

Le nombre de *threads* par *bloc* est restreint par les ressources mémoire limitées d'un processeur. Ainsi, suivant le modèle de carte NVIDIA utilisé, un *bloc* peut contenir au maximum 512 ou 1024 *threads*.

Un *kernel* peut être exécuté par plusieurs *blocs* de même dimension. Dans ce cas le nombre de *threads* est bien sûr le résultat du produit du nombre de *threads* par *bloc*, par le nombre de *blocs*. Ces *blocs* sont disposés à l'intérieur d'une grille ou *grid* comme illustré Figure IV.4. La dimension de cette grille, qui est inférieure ou égale à trois, peut être spécifiée via le premier argument entre les caractères <<< … >>>.

Là encore, les *blocs* peuvent être identifiés à l'intérieur d'un *kernel* grâce à la variable standard *blockIdx*. La dimension d'un *bloc* peut, quand à elle, être accessible à travers la variable standard *blockDim* (elle correspond au nombre de *threads* présents dans un *bloc*).

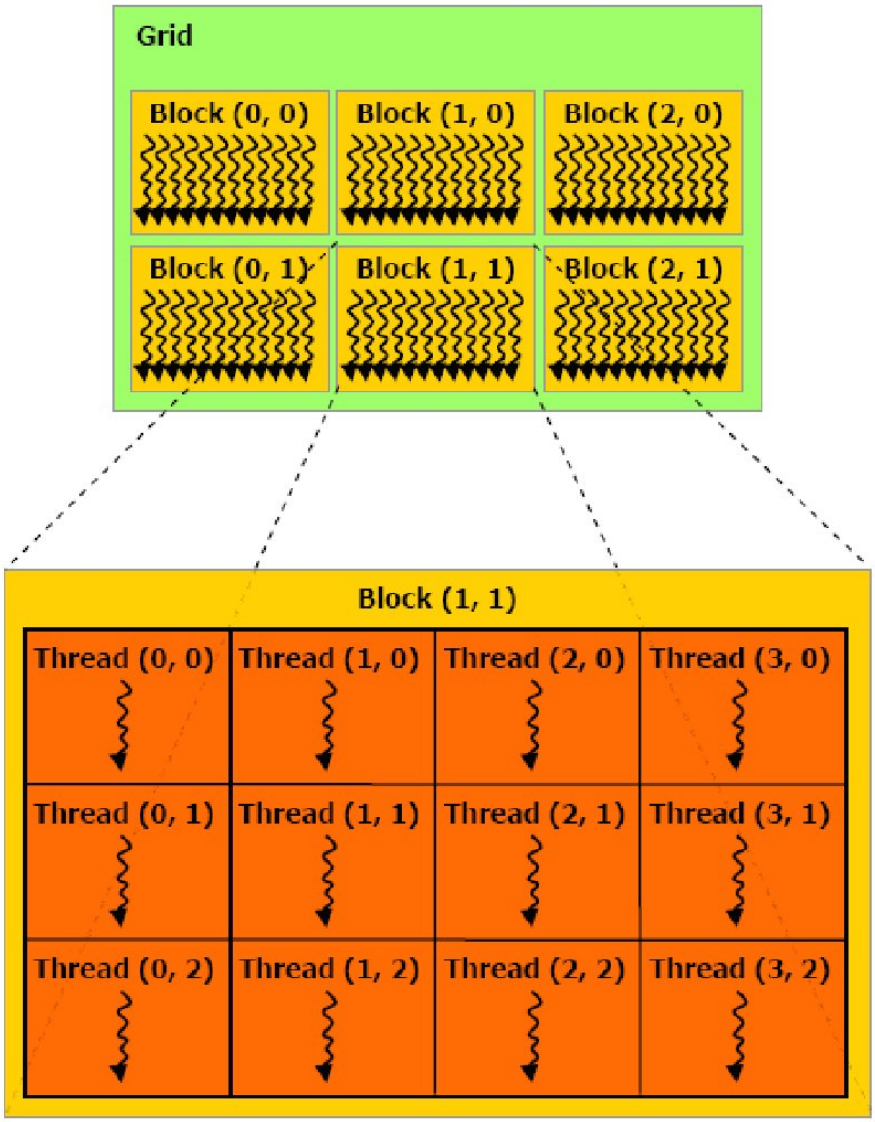



FIGURE IV.4 – Illustration d'une grille de threads (NVIDIA).

Les *blocs* s'exécutent de façon indépendante sans ordre particulier. Le nombre de *blocs* par grille ne dépend pas du nombre de processeurs disponibles mais des données à manipuler. En effet sur une carte GeForce GTX 260, 192 processeurs sont disponibles alors que le nombre maximal de *blocs* par grille est de 65 535.

### IV.3.2 Les mémoires

Les *threads* ont accès à des données se situant dans des mémoires distinctes. Chaque *thread* possède sa propre mémoire appelée *local memory* (mémoire locale). Les *blocs* eux, possèdent une *shared memory* (mémoire partagée) accessible uniquement par les *threads* du *bloc* en

question. On note aussi que chaque bloc dispose d'un banc de registres accessible aux *threads*.

Enfin, tous les *threads* ont accès aux données se trouvant dans la *global memory* (mémoire globale). Il existe également deux autres mémoires qui sont en lecture seule, à savoir la *constant memory* (mémoire constante) et la *texture memory* (mémoire de texture) qui est plus importante que la mémoire constante. Ces deux dernières mémoires sont accessibles à travers un cache, la lecture depuis ces mémoires ne coûte qu'un cycle d'horloge.

### IV.3.3 Host et Device

Comme illustré par la figure IV.5, les *threads* s'exécutent sur une machine physique GPU (qui est encore appelée *device*) séparée, en concurrence avec le reste du code C se déroulant sur ce que l'on appelle le *host* (Le CPU). Ces deux entités possèdent leur propre DRAM (*Dynamic Random Access Memory*) appelées respectivement *device memory* et *host memory*.

Un programme CUDA doit donc manipuler les données de sorte qu'elles soient copiées sur le *device* pour y être traitées, avant d'être recopiées sur le *host*.

Lorsqu'un programme CUDA invoque un *kernel* depuis le *host*, les *blocs* de la grille sont énumérés et distribués aux multiprocesseurs ayant suffisamment de ressources pour les exécuter. Tous les *threads* d'un *bloc* sont exécutés simultanément sur le même multiprocesseur. Dès lors que tous les *threads* d'un *bloc* ont terminé leurs instructions, un nouveau *bloc* est lancé sur le multiprocesseur.

Un multiprocesseur se compose 8 à 192 processeurs suivant le modèle de la carte GPU (32 dans le cas d'une carte C2050). La combinaison d'une barrière de synchronisation *__syncthread*() à moindre coût, d'un procédé de création de *threads* rapide, et l'absence de planification concernant l'ordre d'exécution des *threads* autorise au programmeur de coder avec une faible granularité en assignant par exemple une seule composante d'un vecteur ou encore un pixel d'image à un *thread*.

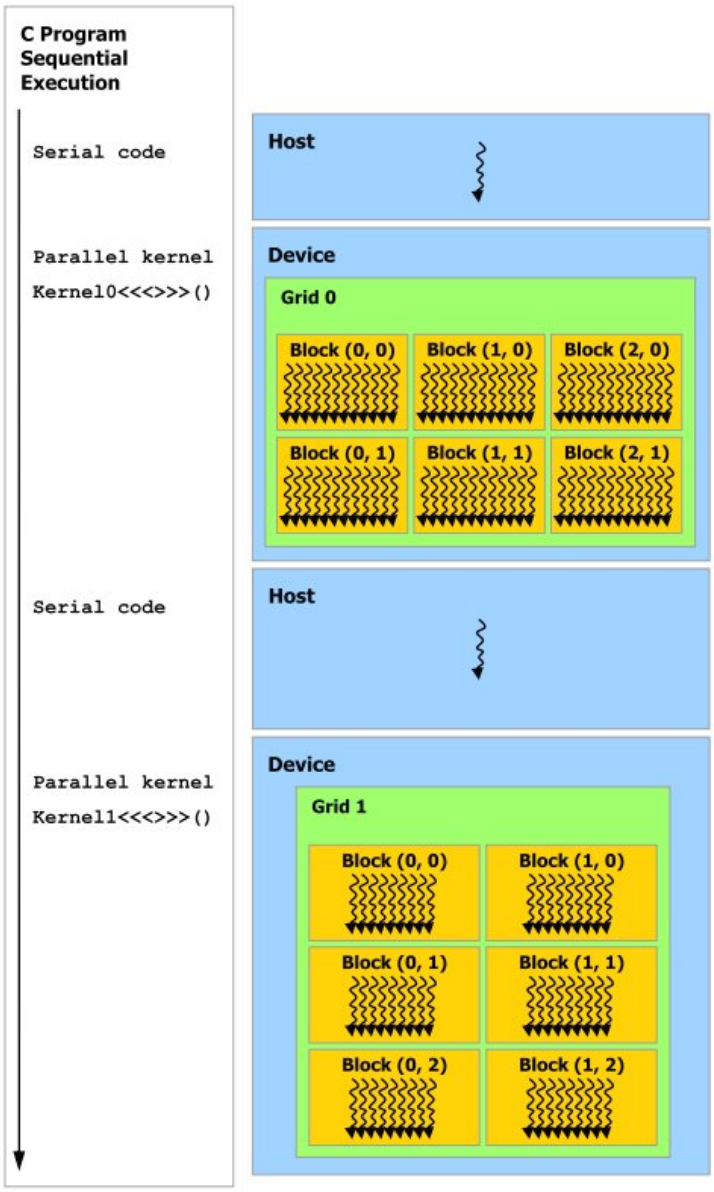



FIGURE IV.5 – Déroulement du programme sur CPU faisant intervenir le GPU (NVIDIA).

Ainsi pour manipuler des centaines de threads, le multiprocesseur emploie l'architecture *SIMT*. C'est donc l'unité SIMT d'un multiprocesseur qui crée, planifie et exécute les threads par groupes de 32. Ces groupes de 32 *threads* sont appelés *warps*, un *half-warp* étant soit la première moitié d'un *warp*, soit la seconde. Les *threads* à l'intérieur d'un *warp* commencent leur exécution à la même instruction dans le programme mais sont libres de s'exécuter indépendamment.

Lorsqu'un multiprocesseur se voit offrir un ou plusieurs *blocs* de *threads* à exécuter, il les divise en *warps* dont l'exécution sera planifiée par l'unité SIMT. Dès lors qu'un *warp* prêt à

être exécuté est sélectionné par l'unité SIMT, la première instruction du *kernel* est lancée pour tous les *threads* actifs du *warp* (certaines instructions de contrôle peuvent rendre inactifs des *threads* à l'intérieur d'un *warp*). Une seule instruction à la fois est exécutée par tout les *threads* d'un *warp*, autrement dit les performances seront maximales si les 32 *threads* composant le *warp* exécutent simultanément la « même » séquence d'instructions. C'est pourquoi il est préférable que les structures de contrôle ne fassent pas diverger les *threads* d'un *warp*. Si toutefois cette situation apparaît, le *warp* exécutera alors tous les chemins (séquences d'instructions) différents pris par les *threads* le composant, et ce en série, c'est-à-dire chemin après chemin. Une fois que tous les chemins sont terminés, les *threads* convergent à nouveau vers le même chemin. Comme le montre la Figure IV.6, la mémoire personnelle d'un multiprocesseur est organisée la manière suivante :

- Un banc de registres par processeur (32768 registres pour la carte Tesla C2050).
- Un bloc mémoire commun à tous les processeurs, où se situe la mémoire partagée.
- Une mémoire cache en lecture seule, conçue pour accélérer l'accès aux données présentes dans la mémoire *constante* (qui est également une mémoire en lecture seule), partagée entre les processeurs.
- Une mémoire cache en lecture seule, conçue pour accélérer l'accès aux données présentes dans la mémoire de *texture* (qui est également une mémoire en lecture seule), partagée entre les processeurs.

Notons qu'il est possible de lire et écrire dans les mémoires dites globale et locale du *device*, et qu'il n'existe pas de mémoire cache permettant d'accélérer l'accès aux données présentes dans les mémoires globale et locale.

Le nombre de *blocs* pouvant être manipulés simultanément par un multiprocesseur – encore appelé nombre de *blocs actifs* par multiprocesseur – dépend des ressources disponibles de ce multiprocesseur. En d'autres termes, le nombre de registres nécessaires par *thread*, mais aussi la taille de la mémoire partagée utilisée par *bloc*, vont avoir un impact direct sur le nombre maximum de *blocs* pouvant être gérés par multiprocesseur. Si d'aventure il n'y avait pas suffisamment de ressources disponibles par multiprocesseur, l'exécution du *kernel* échouerait.

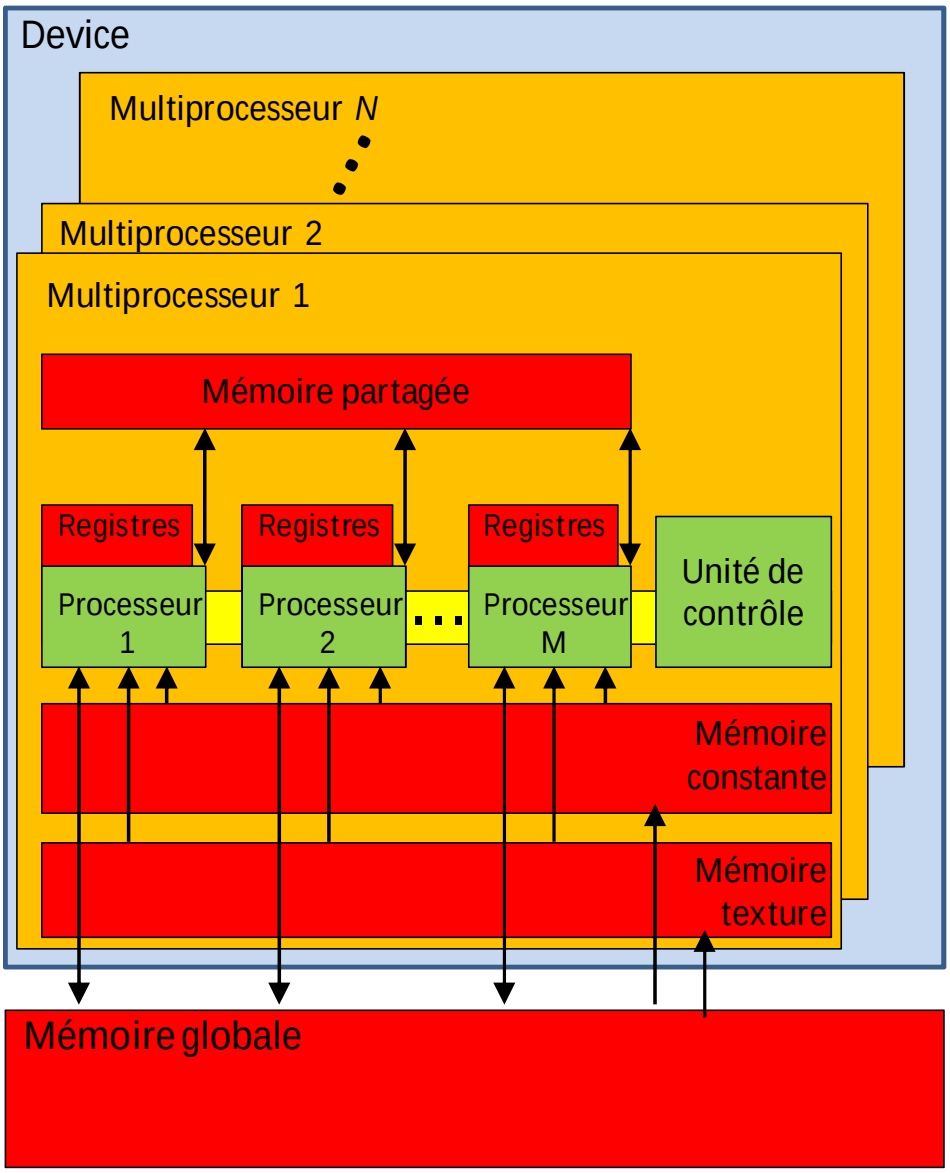



FIGURE IV.6 – Multiprocesseur SIMT avec mémoire partagée embarquée (NVIDIA).

S'il arrive que dans un même *warp*, plusieurs *threads* souhaitent écrire à la même adresse mémoire (que ce soit dans la mémoire globale ou la mémoire partagée), aucune garantie n'est apportée concernant le nombre effectif d'écritures (bien qu'un des *threads* au moins soit certain d'y avoir accès), ni même concernant l'ordre dans lequel les *threads* y écriront. Pour faire face à ce problème majeur, il est possible d'utiliser des fonctions dites *atomiques*. Ces fonctions garantissent un accès exclusif à chaque *thread* sans toutefois qu'un ordre soit défini entre les *threads* pour l'écriture à l'adresse en question.

# IV.4 Règles d'optimisations

## IV.4.1 Instructions de base

La plupart des instructions simples ne nécessitent que quatre cycles d'horloge. C'est le cas par exemple des instructions suivantes :

- Addition et multiplication d'entiers ou de nombre flottants.
- Opération bit-à-bit.
- Comparaison.
- Conversion de type.

Les opérations sur des flottants en double précision nécessitent de 2 à 8 fois plus de cycles que pour les nombres en simple précision, suivant le modèle de GPU utilisé.

En revanche il est recommandé d'éviter les divisions (36 cycles pour les divisions d'entiers ou de flottants) autant que possible. En particulier lors de l'instruction $i/n$ si $n$ est une puissance de 2, alors il convient de remplacer cette instruction par une instruction de décalage de bit équivalente: $i >> \log_2(n)$. De la même façon *i%n* (reste de la division euclidienne de *i* par *n*) sera plutôt mis en œuvre de la façon suivante : $i\&(n-1)$.

## IV.4.2 Instructions de contrôle

Les instructions de contrôle telles que *if*, *switch*, *do*, *for*, ou encore *while* peuvent considérablement ralentir un *kernel*. En effet, pas essence elles créent des chemins différents à l'intérieur d'une même application, autrement dit des divergences au sein d'un *warp*. Chacun de ces chemins créés sera alors exécuté en série jusqu'à ce que tous convergent et que le parallélisme s'applique à nouveau. Malheureusement, ces instructions sont parfois inévitables. Dans de tels cas, il est conseillé, afin d'éviter la divergence des *threads*, de ne pas appliquer la condition sur l'identifiant d'un *thread* mais de faire en sorte qu'elles dépendent plutôt du *warp*. En d'autres termes, il est possible de minimiser les divergences en alignant le conditionnement sur les *warps*, les instructions porteuses de divergences dépendront donc du résultat de *threadIdx/WARP_SIZE* où *WARP_SIZE* est la taille d'un *warp*.

*Exemple :*

- Avec divergence:

$$If\ (threadIdx.x > 2)\ \{\ldots\}$$

Cela crée deux chemins différents pour les threads d'un *bloc*. La granularité est inferieur à la taille du *warp* ; les threads 0 et 1 suivent un chemin différent des autres threads du premier *warp*.

- Sans divergence:

$$If\ (threadIdx.x\ /\ WARP\_SIZE > 2)\ \{\ldots\}$$

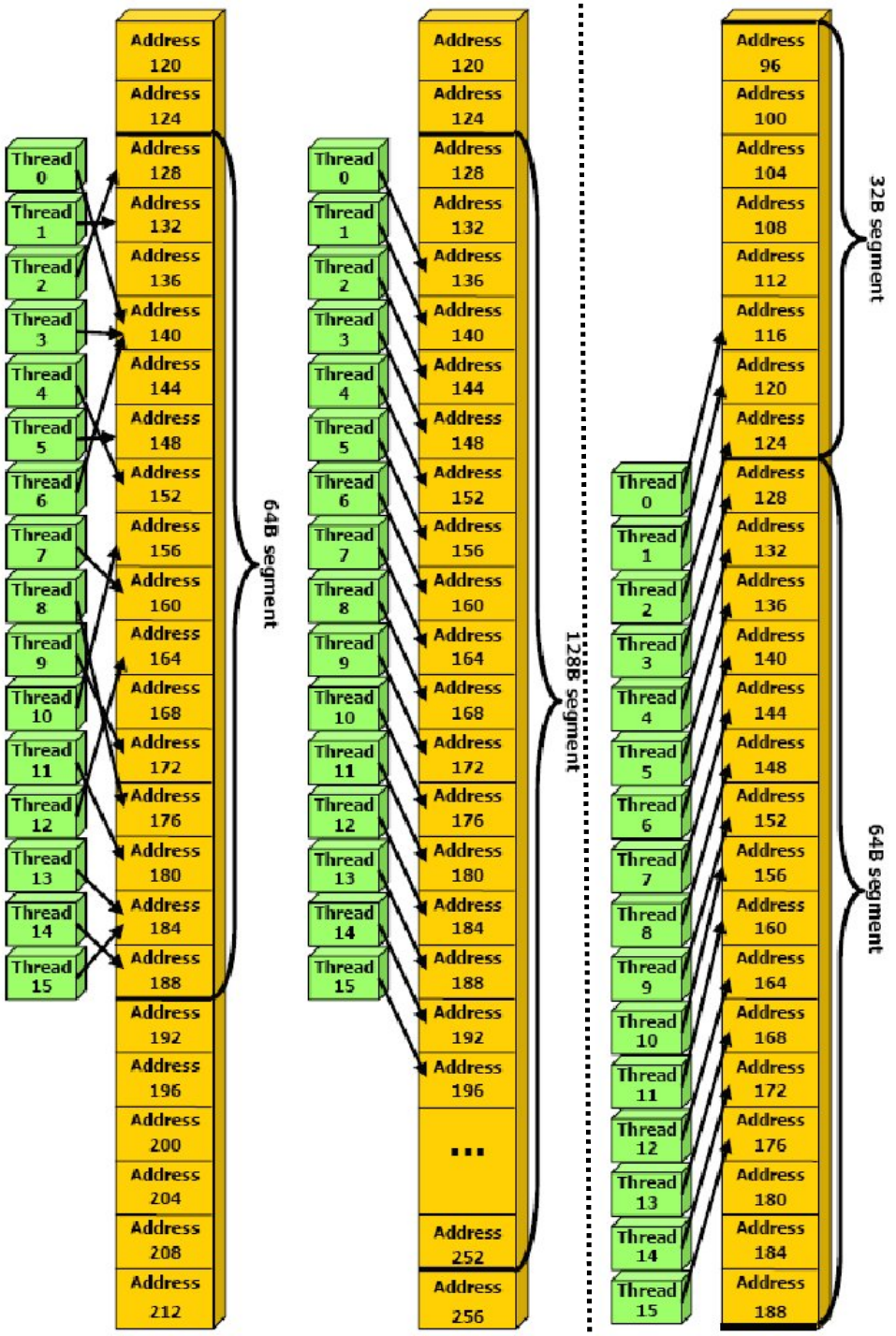
Crée également deux chemins de contrôle différents pour les threads d'un *bloc*. La granularité est un multiple entier de la taille du *warp*; tous les threads dans n'importe quel *warp* donné suivent le même chemin.

### IV.4.3 Instruction de gestion mémoire

Un multiprocesseur requiert 4 cycles d'horloge pour exécuter une instruction de gestion de la mémoire (lecture, écriture) pour un *warp.* Cependant lorsqu'il faut accéder aux mémoires globale ou locale, il faut prévoir entre 400 et 600 cycles d'horloge relatifs au temps de latence. Ce temps de latence peut cependant être dissimulé en codant le programme de sorte que le *kernel* possède un ratio important d'instructions de calculs par rapport aux transactions mémoire. Malgré ces quelques généralités, les différentes mémoires possèdent des caractéristiques très diverses qu'il est bon de maitriser pour éviter une perte de performance non négligeable.

#### a. Mémoire globale

La mémoire globale ne possède aucun cache. Il est donc très important de suivre le mode d'accès aux données prévu afin de maximiser la bande passante. Ce mode d'accès est conçu pour permettre à l'ensemble des *threads* d'un *half-warp* (la moitié d'un *warp*) d'atteindre la mémoire globale en une seule transaction mémoire (on dira que les accès mémoires sont amalgamés). Pour ce faire, ces accès mémoire doivent respecter certaines règles.




Accès mémoires amalgamés:
une seule transaction mémoire.

Accès mémoires non amalgamés:
deux transactions mémoire.

FIGURE IV.7 – Exemples d'accès à la mémoire globale (NVIDIA).

Ainsi il est notamment nécessaire que tous les *threads* d'un *half-warp* accèdent à des données résidant dans un segment (zone d'éléments mémoire contigus) de taille :

- 32 octets si ces *threads* accèdent à des mots de 8 bits,
- 64 octets si ces *threads* accèdent à des mots de 16 bits,
- 128 octets si ces *threads* accèdent à des mots de 32 ou 64 bits.

Si en revanche, le *half-warp* accède à des mots dans *n* segments différents (accès non contigus), *n* transactions mémoires seront réalisées.

A noter que les éléments d'un segment auquel aucun *thread* du *half-warp* n'accède sont tout de même lus. La figure IV.7 illustre quelques exemples d'accès amalgamés et non amalgamés.

### b. Mémoire locale

L'accès à la mémoire locale n'intervient que lorsque des variables automatiques sont affectées non pas aux registres par manque de ressources, mais à la mémoire dite locale. Tout comme la mémoire globale, la mémoire locale ne possède pas de cache, les accès sont donc aussi coûteux que pour la mémoire globale. Les accès sont cependant toujours amalgamés puisque par définition, les accès à la mémoire locale se font par *thread* et non par *warp*.

### c. Mémoire constante

A la différence des deux mémoires précédentes, la mémoire constante possède un cache. Par conséquent, pour tous les *threads* d'un *half-warp*, lire des données situées dans le cache de la mémoire constante est aussi rapide que pour des données situées dans des registres tant que tous ces *threads* lisent à la même adresse. Les coûts d'accès augmentent linéairement avec le nombre d'adresses distinctes accédées par les *threads*.

### d. Registres et mémoire partagée

L'accès aux registres et à la mémoire partagée est plus rapide que celui à la mémoire globale ou locale. Pour les registres, leur nombre par bloc est fixe et dépend du GPU utilisé (32768 registres dans le cas du GPU Tesla C2050). Ainsi le nombre threads par bloc influe directement sur le nombre de registres disponible par thread. Lorsque ce nombre est insuffisant pour une application, il convient alors d'utiliser la mémoire partagée. En effet,

pour chaque *thread* d'un *warp*, accéder à la mémoire partagée est tout aussi rapide que d'accéder aux registres, sous réserve cependant que ces accès mémoires se fassent sans conflit. Nous détaillerons ces conflits plus loin et les appellerons désormais conflits de *bancs*.

Pour offrir des transferts de données efficaces, la mémoire partagée est divisée en segments de tailles égales appelés *bancs*, auxquels il est possible d'accéder simultanément. Cependant, si deux opérations d'accès en mémoire tombent dans le même *banc*, il y a alors création d'un conflit de *banc*, et les accès seront réalisés en série. Il est donc important de comprendre comment manipuler la mémoire partagée pour minimiser ces conflits.

Les *bancs* sont organisées de sorte que des mots successifs de 32 bits chacun soient assignés à des *bancs* successifs, et chaque *banc* possède une bande passante de 32 bits pour 2 cycles d'horloge.

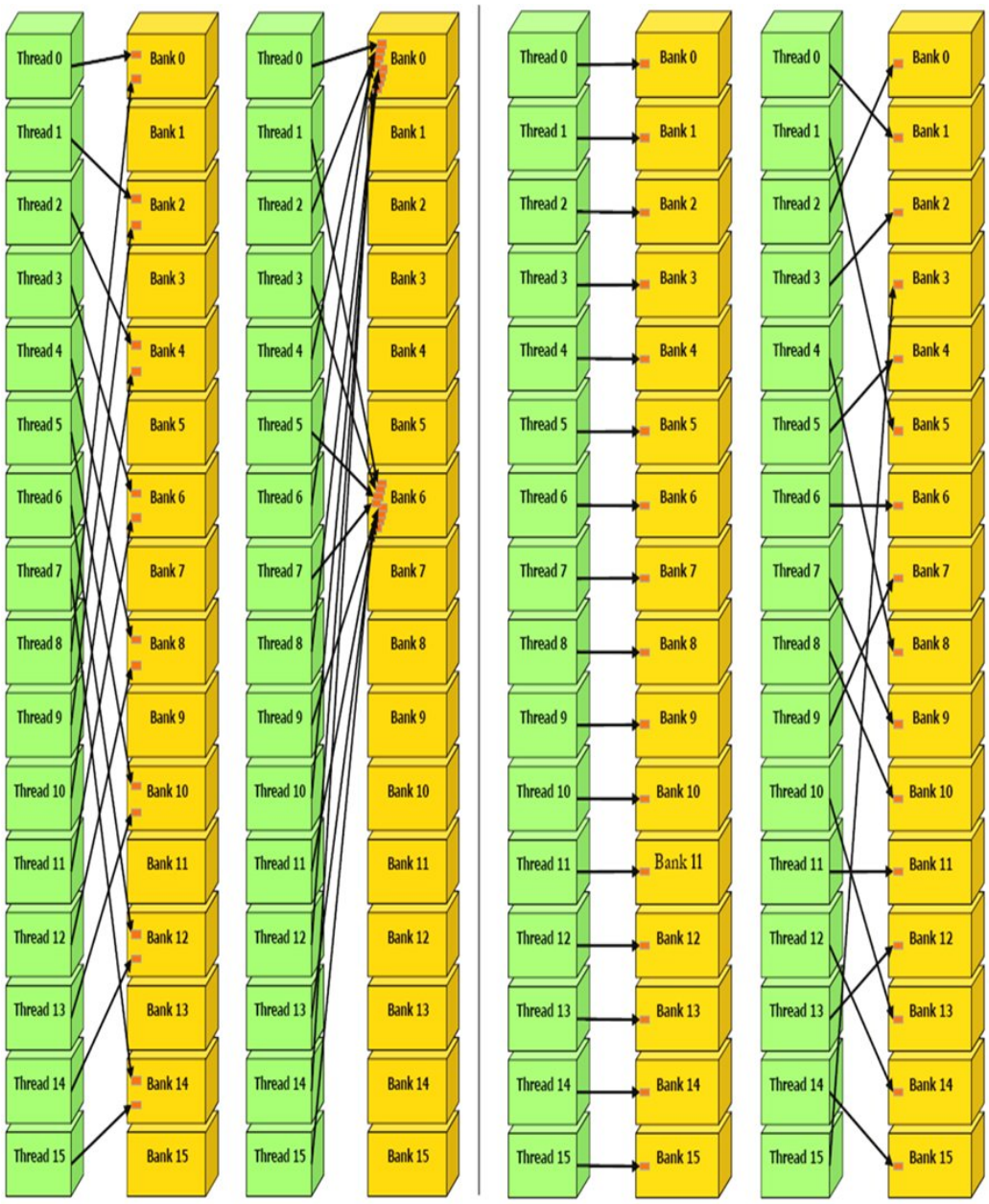
Généralement, chaque thread accède à un mot de 32 bits d'un tableau indexé, grâce à l'identifiant du thread *tid* et à un saut de *s*, comme suit :

```
__shared__ float shared[32];
float data = shared[BaseIndex + s * tid];
```

Dans un cas comme celui-ci, le thread *tid* et le thread *tid*+*n* accèdent au même *banc* chaque fois que $s \times n$ est un multiple du nombre de *bancs m*, soit encore, chaque fois que *n* est un multiple de $m/d$ où *d* est le plus grand diviseur commun de *m* et *s*.

Par conséquent, il n'y aura pas de conflit si et seulement si la taille du *half-warp* est inférieure ou égale à $m/d$. La figure IV.8 illustre quelques exemples d'accès avec et sans conflit de *banc*.

La mémoire partagé dispose aussi d'un mécanisme de diffusion par lequel un mot de 32 bits peut être lu et diffusé à plusieurs threads d'un demi-warp lorsque ceux-ci doivent y accéder simultanément. Ce mot est alors diffusé en une seule opération de lecture réduisant ainsi les conflits. La figure IV.9 illustre un exemple de mécanisme de *diffusion*.




Conflit de *bancs* Pas de conflit de *bancs*

FIGURE IV.8 – Adressages à la mémoire partagée avec et sans conflit de *bancs* (NVIDIA).

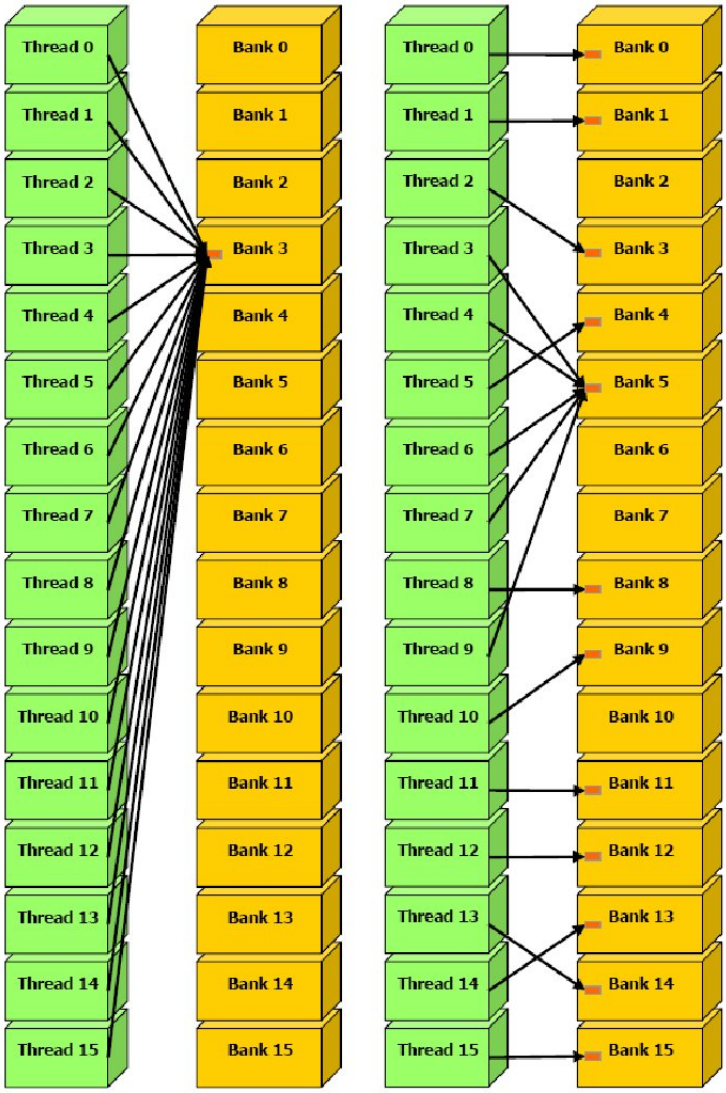



FIGURE IV.9 – Adressages à la mémoire partagée avec diffusion (NVIDIA).

### e. Nombre de threads par bloc

Nous introduisant d'abord la notion d'occupation qui est le nombre de *warps* qu'un multiprocesseur peut exécuter en même temps. Ainsi, la première règle à prendre en considération lors du choix de la configuration est de maximiser l'occupation de chaque multiprocesseur, et donc de choisir un nombre de threads par blocs qui soit suffisamment important. Cela permet de bien cacher le temps de latence lors des accès aux mémoires globale et locale.

Cependant plus le nombre de *threads* par *bloc* est important, moins il y a de registres disponibles. Cet inconvénient peut donc conduire à une utilisation trop importante de la mémoire locale, peu efficace en termes de temps d'accès.

Mark Harris propose dans [HAR 10] l'outil CUDA Occupancy Calculator – sous forme d'une feuille de calcul MS Excel – qui permet de choisir le nombre de threads par blocs pour un *kernel* et un modèle de GPU donnés, de manière à avoir la meilleure occupation du GPU.

### f. Transferts de données CPU ↔ GPU

Il est recommandé de faire le moins de transfert de données possible entre les deux entités. En effet ces transferts sont très lents. Il est préférable a priori, de migrer davantage de code sur carte graphique quitte à avoir une faible occupation des ses multiprocesseurs à condition que le temps de calcul sur GPU puisse masquer ce transfert. Enfin un transfert important de données est plus intéressant que plusieurs transferts moindres, du fait du temps de latence.

Dans les chapitres suivants, nous présentons des mises en œuvre de manière parallèle sur GPU de deux méthodes exactes utilisées dans le domaine de l'optimisation, à savoir la méthode de Branch and Bound pour des problèmes de sac à dos et la méthode du Simplexe pour des problèmes de programmation.

Nous donnons en A*nnexe A* les différentes caractéristiques de la carte GPU (Tesla C2050) utilisée dans cette thèse.

## IV.5 Conclusion

Les architectures GPU sont devenues un moyen économique qui permet de mettre en œuvre efficacement diverses applications avec un parallélisme important. Des environnements comme CUDA et OpenCL permettent de programmer assez facilement ce type d'architecture. De nombreuses mises en œuvre sur GPU ont été proposées dans la littérature pour des applications variées en traitement du signal, en algèbre linéaire et en optimisation.

# Chapitre V

# Mise en œuvre CPU-GPU de la méthode de Branch and Bound

## Sommaire



## V.1 Introduction

Dans ce chapitre, nous présentons un algorithme parallèle de Branch and Bound sur GPU pour le problème du sac à dos. Nous commençons d'abord par un bref rappel mathématique sur la méthode de Branch and Bound et nous présentons les différentes étapes de la méthode parallèle de Branch and Bound que nous avons mise en œuvre. La structure des données en mémoire est détaillée ainsi que le partage de tâches et les différentes communications entre le CPU et le GPU. Finalement, Nous donnons quelques résultats

expérimentaux qui confirment l'intérêt de l'utilisation des GPUs pour résoudre les problèmes de la famille du sac à dos.

## V.2 Branch and Bound pour le sac à dos

Le problème du sac à dos est parmi les plus étudiés des problèmes d'optimisation discrète; c'est également l'un des prototypes les plus simples de problèmes de programmation en variables entières. Le problème du sac à dos intervient aussi comme un sous-problème de beaucoup de problèmes complexes en optimisation combinatoire.

### V.2.1 Formulation du problème

Nous rappelons la formulation donnée au chapitre II.

$$(KP)\begin{cases} z(KP) = \max \sum_{i=1}^{n} p_i.x_i, \\ \sum_{i=1}^{n} w_i.x_i \leq c, \\ x_i \in \{0,1\}, i \in \{1,..,n\} \end{cases} \quad \text{(V.1)}$$

où $p_i \in \mathbb{N}_+^*$ et $w_i \in \mathbb{N}_+^*$ correspondent respectivement au profit et au poids des articles $i \in \{1, ..., n\}$, et $c \in \mathbb{N}_+^*$ est la capacité du sac à dos.

Pour éviter toute solution triviale, nous supposons que nous avons :

$$\forall i \in \{1,\ldots,n\}, w_i \leq c. \quad \text{(V.2)}$$

$$\sum_{i-1}^{n} w_i \geq c. \quad \text{(V.3)}$$

La relation (V.2) assure que chaque article peut être chargé dans le sac à dos.

La relation (V.3) évite le cas trivial où le sac à dos contient la somme de tous les articles.

Nous supposons par la suite, que les articles sont triés par ratio profit sur poids décroissant :

$$\frac{p_1}{w_1} \geq \frac{p_2}{w_2} \geq ..... \geq \frac{p_n}{w_n}. \quad \text{(V.4)}$$

## V.2.2 La méthode de Branch and Bound

Comme cela a été présenté au chapitre II, la méthode de Branch and Bound est une méthode générale qui permet de trouver la solution optimale à différents problèmes d'optimisation, tels ceux de l'optimisation discrète et combinatoire (cf. [BOY 10] et [KAC 08]).

Elle est basée sur l'énumération des solutions candidates et l'élagage de sous-ensembles de solutions candidates par calcul des bornes inférieures et supérieures du critère à optimiser. Nous nous concentrons ici sur la stratégie de recherche *en largeur d'abord* (voir section II.3.1) qui est bien adaptée à la mise en œuvre parallèle sur GPU et pour laquelle les résultats donnés par la méthode parallèle sont identiques à ceux fournis par la méthode séquentielle. A chaque étape de la méthode de Branch and Bound, les mêmes tâches de séparation et de calcul de bornes sont exécutées de manière indépendante sur un grand nombre de données, correspondant ici à la liste des états (qui sont appelés aussi *nœuds)*.

**Définition des bornes**

Soit $k$ l'indice de l'article sur lequel s'effectue l'étape de séparation ; nous noterons par $N_e^k$ l'ensemble des articles chargés dans le sac à l'étape $k$, pour un nœud donné $e$. Un nœud $e$ est généralement caractérisé par le quintuplet ($w_e$, $p_e$, $X_e$, $U_e$, $L_e$) où $w_e$ représente le poids du nœud $e$, $p_e$ le profit du nœud, $X_e$ le sous-vecteur solution associé au nœud $e$ ; $U_e$ et $L_e$ sont respectivement, la borne supérieure et la borne inférieure du nœud.

Nous avons :

$$w_e = \sum_{i \in N_e^k} w_i, \qquad p_e = \sum_{i \in N_e^k} p_i. \tag{V.5}$$

Nous rappelons que la borne supérieure de Dantzig (section II.4.1), obtenue à partir de la relaxation continue LKP du problème du sac à dos (voir relation II.9), est donnée par la relation suivante :

$$U_e = p_e + \sum_{i=k+1}^{s_e-1} p_i + \left\lfloor c \frac{p_{s_e}}{w_{s_e}} \right\rfloor, \tag{V.6}$$

où $s_e$ est l'indice de base donné par la relation suivante :

$$\sum_{i=k+1}^{s_e-1} w_i \le c - w_e < \sum_{i=k+1}^{s_e} w_i, \tag{V.7}$$

On note que l'on peut calculer l'indice $s_e$ pour un nœud $e$ et l'article $k + 1$ par un algorithme Glouton (voir *Procédure* 1 du Chapitre III) appliqué à un problème du sac à dos de capacité $c - w_e$.

On note $\underline{c}$ la capacité résiduelle donnée par

$$\underline{c} = c - w_e - \sum_{i=k+1}^{s_e-1} w_i \tag{V.8}$$

La borne inférieure $L_e$ du problème peut être obtenue à partir d'une solution admissible du problème du sac à dos. Par exemple, on peut avoir une borne inférieure intéressante via un l'algorithme Glouton (voir la *Procédure* 1 du Chapitre III). En effet, en sélectionnant tous les articles d'indice inferieur à l'indice de la variable de base $s_e$, et sachant que les articles sont classés selon la relation (V.4), on tente d'ajouter des éléments d'indice $i \ge s_e + 1$ dans le sac tant que la capacité résiduelle $\underline{c}$, définie par la relation (V.8), le permet.

Ainsi, la borne inferieure $L_e$ d'un nœud $e$, présentée par la relation V.9, est composée de deux termes : le premier terme est le profit $p_e$ du nœud et le deuxième terme, égal à $\sum_{i=k+1}^{s_e-1} p_i + \sum_{i=s_e+1}^{n} p_i.x_i$, est la solution fournie par l'algorithme Glouton appliqué à un problème du sac à dos de capacité $c - w_e$.

$$L_e = p_e + \sum_{i=k+1}^{s_e-1} p_i + \sum_{i=s_e+1}^{n} p_i.x_i, \tag{V.9}$$

où $x_i = 1$ si $w_i \le \underline{c} - \sum_{j=s_e+1}^{i-1} w_j.x_j$ .

Par souci de simplicité, et d'efficacité, un nœud $e$ sera représenté par la suite par le quintuplet $(\hat{w}_e, \hat{p}_e, s_e, U_e, L_e)$ avec

$$\hat{w}_e = w_e + \sum_{i=k+1}^{s_e-1} w_i \tag{V.10}$$

et

$$\hat{p}_e = p_e + \sum_{i=k+1}^{s_e-1} p_i \tag{V.11}$$

A l'étape de $k$ +1, le terme $\sum_{i=k+1}^{s_e-1} p_i$ de la relation (V.11), qui est une partie de la solution fournie par l'algorithme Glouton, est soit diminué de $p_{k+1}$, soit sauvegardé tel quel (voir la sous-section Séparation). On évite ainsi de « recalculer » ce terme à chaque étape de la méthode de Branch and Bound, et les deux bornes $U_e$ et $L_e$ seront fournies par l'algorithme Glouton appliqué cette fois-ci à un problème du sac à dos de capacité plus petite que $c - w_e$, à savoir $c - \hat{w}_e$.

La borne supérieure $U_e$ ainsi que la borne inferieure $L_e$ s'écrivent alors de la manière suivante:

$$U_e = \hat{p}_e + \left\lfloor \underline{c} \frac{p_{s_e}}{w_{s_e}} \right\rfloor, \tag{V.12}$$

$$L_e = \hat{p}_e + \sum_{i=s_e+1}^{n} p_i . x_i , \tag{V.13}$$

Où $x_i = 1$ si $w_i \leq \underline{c} - \sum_{j=s_e+1}^{i-1} w_j . x_j$ .

Nous notons que le calcul de bornes consomme beaucoup de temps dans la méthode de Branch and Bound et obtenir ainsi les bornes $U_e$ et $L_e$ s'avère plus efficace.

**Séparation**

La séparation s'effectue à l'étape $k$ de la méthode de Branch and Bound, sur tous les nœuds de la liste. Nous noterons par $q$ le nombre de nœuds de cette liste.

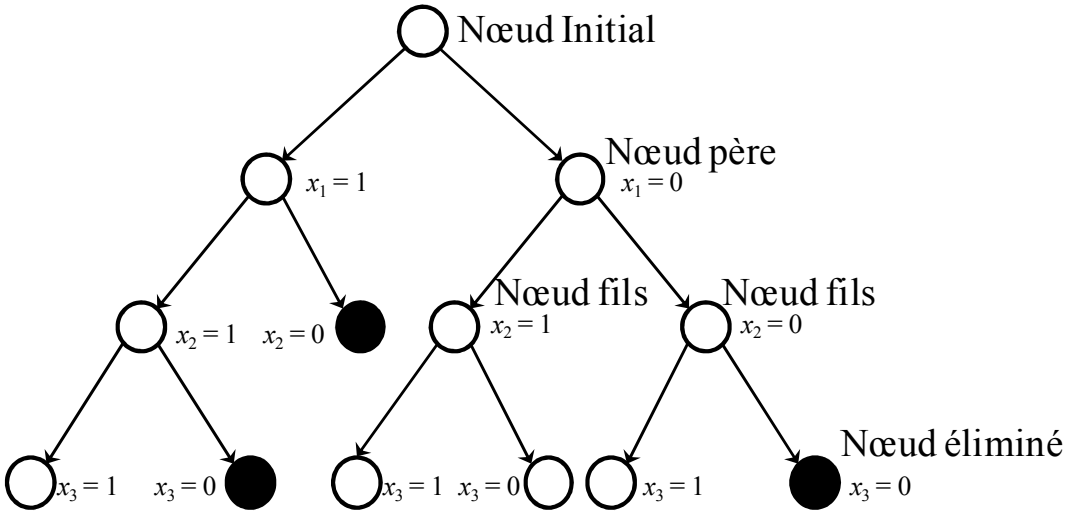



FIGURE V.2 – Arbre de décision

Pour le nœud *e*, qui devient le nœud père, l'étape de séparation produit des nœuds fils (voir la Figure V.1), de la manière suivante :

Si $k < s_e$, alors le nœud *e* engendre deux fils à l'étape *k* :

- un nœud avec $x_k = 1$ (notons que ce nœud est identique au père qui figure déjà dans la liste).
- un nœud avec $x_k = 0$, tel que $\hat{w}_e = \hat{w}_e - w_k$ et $\hat{p}_e = \hat{p}_e - p_k$ qui sera placé dans la liste à la suite des *q* premiers nœuds.

Le cas où $k = s_e$ donne seulement un fils avec $x_k = 0$. Cela revient à garder le nœud père dans la liste et poser $s_e = s_e + 1$.

**Calcul de bornes**

Une fois que la liste des nœuds à été réactualisée l'étape de calcul de borne est lancée sur chaque nœud *e* de la liste.

- Dans le cas d'un nœud fils correspondant à $x_k = 1$, la borne supérieure ainsi que la borne inférieure ne changent pas.

- Dans le cas d'un nœud fils correspondant à $x_k = 0$, on calcule le nouvel indice de la variable de base $s_e$. Par conséquent, $\hat{w}_e$ et $\hat{p}_e$ sont mis à jour et les deux nouvelles bornes $U_e$ et $L_e$, seront déduites.

A la fin de cette étape, on met à jour la meilleure borne inférieure, notée $\overline{L}$ , en prenant le maximum de la précédente meilleure borne inferieure et des bornes inferieures des nœuds nouvellement créés.

**Etape d'élagage**

L'étape d'élagage est lancée pour éliminer les nœuds «non prometteurs» (voir la Figure V.1). Si pour un nœud *e*, nous avons une borne supérieure qui vérifie l'inégalité suivante : $U_e \leq \overline{L}$, alors ce nœud est supprimé de la liste. Dans le cas contraire, le nœud est conservé dans la liste.

Les étapes décrites ci-dessus représentent une itération de la méthode de Branch and Bound. On réitère ces étapes tant que :

- la liste des nœuds n'est pas vide ( $q \neq 0$ ).
- l'on n'a pas considéré tous les articles du problème réduit.

La solution du problème KP est alors obtenue par la borne $\overline{L}$ .

Nous passons maintenant à la partie mise en œuvre parallèle sur un système CPU-GPU de l'algorithme décrit ci-dessus.

## V.3 Etat de l'art

La parallélisation des méthodes d'optimisation combinatoire afin de diminuer les temps de calcul, a retenu l'attention de nombreux chercheurs lors des trente dernières années [OWE 07], [GEN 94], [DEN 96] et [ELB 05].

De plus, l'utilisation de stratégies parallèles de Branch and Bound sur de grand clusters et grilles d'ordinateurs, combinée à des outils avancés de programmation, à savoir le multithreading et des fonctionnalités de tolérance aux fautes, ont fait le succès de la mise en œuvre parallèle de la méthode de Branch and Bound [GOU 02].

Nous présentons dans la suite un bref état de l'art sur les algorithmes parallèles de Branch and Bound.

On peut classer les algorithmes parallèles de Branch and Bound selon le grain de parallélisme utilisé. On peut noter l'existence de trois classes de méthodes parallèles.

- *Parallélisme à grain fin : parallélisation des taches d'évaluation*. Dans cette classe, le parallélisme est introduit lors de l'évaluation des bornes des différents nœuds de l'arbre de Branch and Bound. Cette stratégie dépend fortement de la méthode d'évaluation d'un sous-problème et n'influe pas sur la structure de l'algorithme de Branch and Bound. Un exemple de cette mise en œuvre est l'utilisation de la librairie de routine d'optimisation d'IBM *Optimization Subroutine Library* (OSL) [IBM 95] lorsque la fonction d'évaluation est sous la forme d'un problème de programmation linaire. En effet, cette librairie fournit trois méthodes de résolution en parallèle de programmes linéaires: la méthode de la barrière logarithmique, la méthode de décomposition par bloc et le Simplexe.
- *Parallélisme à grain moyen : explorations concurrentes* : cette stratégie de parallélisation consiste à construire l'arbre de Branch and Bound en parallèle. Plusieurs nœuds de l'arbre de Branch and Bound sont évalués simultanément sur les processeurs de la machine parallèle. Cette stratégie est bien adaptée à des architectures MIMD (*Multiple Instruction Multiple Data*) asynchrones; mais son intérêt reste très limité et donc très peu étudié. Cependant voici quelques références qui utilisent cette stratégie : Miller et Pekny qui ont proposé dans [MIL 93] de construire des arbres de Branch and Bound avec des stratégies de séparation différentes, Jankiram et al. ont proposé dans [JAN 88] une mise en œuvre utilisant la variante du parcours en profondeur d'abords, et Kumar et Kanal qui ont proposé dans [KUM 84] de construire les arbres de Branch and Bound en parallèle avec des tests d'élagage différents.
- *Parallélisme à gros grain :* Dans cette dernière classe on construit plusieurs arbres de Branch and Bound en parallèle avec des stratégies différentes, comme par exemple : *meilleur d'abord* ou *largeur d'abord*. Sur chaque processeur, on peut choisir une stratégie différente de branchement, de sélection ou d'élimination. Les informations recueillies sur un processeur pendant la construction d'un arbre peuvent être utilisées par d'autres processeurs. Cette classe de parallélisation à gros grain de calcul est bien adaptée à des architectures MIMD (*Multiple Instruction Multiple Data*).

Il existe aussi quelques mises en œuvre sur des machines massivement parallèles d'architecture SIMD (*Single Instruction Multiple Data*) de grain fin. Ces mises en œuvre sont plutôt adaptées aux problèmes dont la fonction d'évaluation est triviale et qui s'exécutent en temps constant. Il existe plusieurs travaux qui ont mis en œuvre cette stratégie : par exemple: Roucairol à proposé dans [ROU 87] une mise en œuvre de la méthode de Branch and Bound, utilisant la variante "meilleur d'abord" pour la séparation sur la machine Cray X-MP 48. Kumar et al. ont étudié dans [KUM 88] les performances de différents algorithmes parallèles de Branch and Bound sur une machine BBN Butterfly à 100 processeurs. Quinn a présenté dans [QUI 90] une mise en œuvre sur une machine NCUBE/7 hypercube avec 64 processeurs. Denneulin et Le Cun [DEN 96] ont implémenté un algorithme parallèle de Branch and Bound sur un environnement distribué (DEC ALPHA avec 16 processeurs ou sur le IBM-SP2).

Pour plus de détails sur les différentes mises en œuvre parallèles de la méthode de Branch and Bound, nous renvoyons le lecteur à la référence [GEN 94].

Nous nous intéressons ici à la mise en œuvre de la méthode de Branch and Bound sur GPU.

Nous notons l'existence d'une seule étude en liaison avec ce sujet (cf. [CHA 11]). Dans ce travail, les auteurs proposent de résoudre des problèmes de Flow-shop sur GPU via la méthode de Branch and Bound. L'idée est d'utiliser le GPU pour l'évaluation en parallèle des différentes bornes à chaque étape de la méthode de Branch and Bound. Les auteurs proposent aussi dans cette étude une technique originale pour éviter l'utilisation d'instructions conditionnelles qui causent la divergence de chemins entre threads d'un même *warp.*

Nous nous intéressons, quant à nous, à la résolution de problèmes de la famille du sac à dos sur GPU. En effet, nous développons actuellement dans l'équipe CDA du LAAS-CNRS, une série de codes parallèles sur GPU destinés à être combinés afin de produire des méthodes hybrides parallèles efficaces pour la résolution de problèmes de la famille du sac à dos comme par exemple le sac à dos multidimensionnel. Ainsi, des mises en œuvre via CUDA de la méthode de programmation dynamique dense sur des architectures GPU et multi GPU ont été proposées dans [BOY 11] et [BOY 12] pour résoudre le problème du sac à dos.

# V.4 Calcul hybride

Dans cette section, nous présentons une mise en œuvre parallèle de la méthode de Branch and Bound sur un système CPU – GPU (cf. [LAL 12b]).

Dans la méthode de Branch and Bound, le calcul de bornes est prépondérant. L'idée de départ est de résoudre le problème KP sur CPU tant que le nombre de nœud n'est pas important. Le GPU est sollicité lorsqu'un nombre important de nœuds est engendré. En résumé tant que le nombre nœud est inferieur à un certain seuil, que nous fixerons par la suite, le problème est traité sur CPU. Dans le cas contraire le GPU prend le relai, d'où la notion de mise en œuvre CPU-GPU. De plus, le GPU fait appel au CPU lors de l'étape d'élagage, ceci sera vu plus en détail par la suite.

Dans ce qui suit, nous présentons la mise en œuvre des différentes étapes de la méthode de Branch and Bound via CUDA.

## V.4.1 Initialisation et algorithme général

Au début de l'algorithme il est nécessaire de passer par une étape d'initialisation du GPU. Celle-ci consiste à allouer de l'espace mémoire sur le GPU (mémoire globale) sous forme de 5 tableaux correspondant au quintuple $(\hat{w}_e, \hat{p}_e, s_e, U_e, L_e)$. Les quatre premiers auront comme dimension $t$ et le cinquième aura comme dimension $\frac{t}{2}$ (en effet à chaque itération, le nombre de nœuds est de $q$ dont $\frac{q}{2}$ nœuds nouvellement créés dont on calculera les bornes inferieures respectives. On en déduit que la dimension du tableau de la borne inferieure est la moitié de la dimension des autres tableaux). Un sixième tableau *Mark* de dimension $t$, servant au marquage des nœuds non prometteurs, est aussi alloué sur le GPU.

La dimension $t$ est obtenue en divisant la taille de la mémoire globale du GPU par $5{,}5 \times 4$ *octets* (4 octets étant la taille d'un entier).

Nous faisons de même sur CPU sauf qu'ici la taille des tableaux est plus petite car celle-ci correspond au seuil (le nombre de nœuds qui entrainera la résolution du problème sur GPU) qu'on aura fixé.

Un tableau d'articles contenant le poids *w* et le profit *p*, qui n'évoluent pas au cours de la résolution du problème KP est stocké dans la mémoire de « texture » du GPU afin de réduire les latences mémoire lors de l'accès à ces données. De plus, le nombre d'articles peut être important ce qui justifie le choix de la mémoire texture au détriment de la mémoire constante qui plus petite.

La capacité *c* ainsi que la dimension du problème *n* sont stockés dans la mémoire constante du GPU car elles n'évoluent pas au cours de la résolution du problème. Par conséquent, il ne sera pas fait référence à ces variables comme arguments dans les différents *Kernels* présentés ci-dessous, ceci afin de simplifier la notation algorithmique.

### V.4.2 Algorithme parallèle

Dans le problème du sac à dos (V.1) de taille $n$, les articles sont classés par ordre d'efficacité décroissante, cf. (V.4). La taille du problème est réduite au moyen de techniques de réduction de variables décrites dans la section II.4.3.

Le problème réduit est traité de manière séquentielle sur CPU tant que le nombre de nœuds crées n'est pas important. Le nombre « seuil » de nœuds, qui entrainera la résolution du problème sur GPU, est fixé à 192. Ce nombre représente le nombre minimal de nœuds à partir duquel le GPU utilisé (Tesla C2050) est plus efficace que le CPU en termes de temps de calcul. Le choix de ce nombre sera expliqué à la section V.4.3. Notons que si le nombre de nœuds devient par la suite inferieur à 192, la résolution se fera alors à nouveau de manière séquentielle. Une fois ce nombre atteint, la liste des nœuds est transférée sur la mémoire globale du GPU (voir Figure V.2). Le temps de communication entre le CPU et le GPU est négligeable par rapport au temps global d'exécution de l'algorithme de Branch and Bound.

Une fois la communication effectuée, l'étape de séparation est lancée en parallèle via un *Kernel* faisant intervenir autant de *threads* que de nœuds à séparer ; ce qui permet de créer *q* nouveaux nœuds fils.

L'étape de calcul de borne est ensuite lancée, en parallèle via un *Kernel*, pour évaluer les bornes supérieures et inferieures des *q* nouveaux nœuds. On en tire alors la meilleure borne inférieure qui servira de critère de comparaison à l'étape d'élagage (voir sous-section V.1.2).

L'algorithme parallèle s'achève lorsque la liste de nœuds est vide, autrement, un nouvel article est considéré.

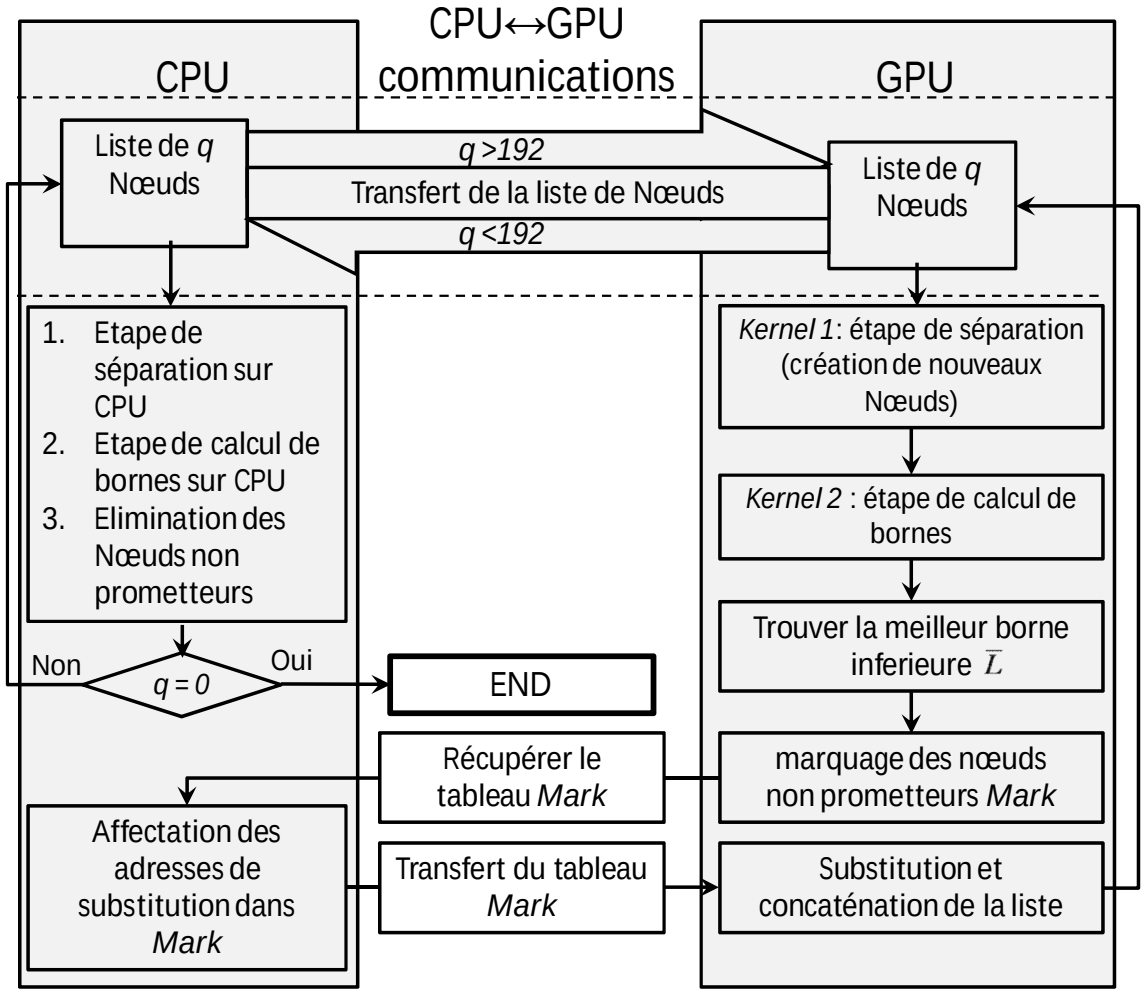



FIGURE V.2 − Algorithme de Branch and Bound sur système CPU-GPU.

Les tâches principales de notre algorithme parallèle sont présentées ci-dessous (voir aussi Figure V.2).

**Tâches effectuées sur le CPU**

- Exécuter l'algorithme de *Branch and Bound* sur le CPU lorsque la liste est de petite taille.
- Transférer la liste de nœuds sur le GPU lorsque le nombre de nœuds est supérieur au *seuil*.
- Lancer la phase de séparation sur GPU.
- Lancer la phase de calcul de bornes sur GPU.

- Récupérer le tableau *Mark* qui a servi à marquer les nœuds non prometteurs qui doivent être éliminés de la liste des nœuds. Autrement dit, chaque élément *Mark*[*i*] indique si le nœud *i* doit être éliminé ou sauvegardé, $i = 1,\ldots, 2q$.
- Repérer dans la liste, les nœuds à éliminer qui doivent être substitués par des nœuds prometteurs, dans le but de concaténer la liste des nœuds. Ceci est fait via le tableau *Mark* où on affecte à l'élément *Mark*[*i*] l'adresse du nœud prometteur qui doit remplacer le nœud *i*, si ce dernier est non prometteur. Le tableau *Mark* est ensuite transférer au GPU.

**Tâches effectuées sur le GPU**

- Effectuer la séparation (*Kernel* 1).
- Exécuter le calcul de bornes (*Kernel* 2).
- Trouver la meilleure borne inférieure $\overline{L}$ .
- Comparer les bornes supérieures à la borne $\overline{L}$ et marquer par 0 les nœuds à éliminer (via le tableau *Mark*).
- Concaténer la liste des nœuds en éliminant les nœuds non prometteurs.

Par la suite, nous détaillons les différentes tâches effectuées sur le CPU et le GPU, respectivement. Nous commençons par les tâches mises en œuvre sur le GPU.

## V.4.3 Calculs sur GPU

Quand la liste de nœuds est disponible sur le GPU, une grille de threads est alors configurée de manière à couvrir tout les nœuds de la liste. Notons par *NTh* le nombre de threads par bloc, la grille est alors composée de $\left\lceil \frac{q}{NTh} \right\rceil$ blocs.

Chaque *thread* dans la *grille* de *blocs*, que nous avons considérée effectue le calcul sur un seul nœud de la liste de *Branch and Bound*, afin d'avoir un accès contigüe (coalesçant) à la mémoire globale.

Le choix du nombre *NTh* de threads par bloc influe directement sur les performances de notre algorithme parallèle. Un nombre important de threads par blocs s'avère utile pour mieux masquer la latence des opérations mémoires et garantir une meilleure occupation du GPU. Mais d'un autre côté cela diminue le nombre de registres disponibles par threads (voir la sous-section IV.4.3.e).

Mark Harris propose dans [HAR 10] l'outil CUDA Occupancy Calculator − sous forme d'une feuille de calcul MS Excel − qui permet de choisir le nombre de threads par blocs pour un *Kernel* et un modèle de GPU donnés, de manière à avoir la meilleure occupation du GPU (une vérification a posteriori peut être effectuer grâce à l'outil CUDA PROFILER) ; ainsi, l'outil CUDA Occupancy Calculator propose, dans notre cas, une valeur de *Nth* égale à 192.

Quand le nombre de nœuds $q$ devient important (supérieur à 200 000 nœuds), le nombre de blocs devient à son tour important ce qui a pour effet de détériorer le temps d'exécution. Il arrive même que le GPU engendre des erreurs (pour $q \geq 10\,000\,000$ nœuds). Voila pourquoi il est impératif d'adapter le nombre *NTh* au nombre de nœuds dans la liste. Après plusieurs séries de testes effectuées, nous avons fait les choix suivant pour la valeur *NTh* :

- = 192 pour des valeurs de $q < 200000$ nœuds,
- = 512 pour des valeurs de $200000 \leq q < 10000000$ nœuds,
- = 1024 pour des valeurs de $q \geq 10000000$ nœuds.

Ces choix nous ont permis d'avoir les meilleurs résultats en termes de temps de calcul.

**Création de nouveaux nœuds**

Le *Kernel* 1 correspond à l'étape de séparation effectuée sur le GPU. Le $e^{\text{ème}}$ *thread* entreprend la séparation du nœud *e,* $e = 1, \ldots, q$ et crée un nouveau nœud à l'adresse $e + q$ (voir la Figure V.3) de la manière suivante :

- Dans le cas où $k < s_e$, Le $e^{\text{ème}}$ *thread* calcule les valeurs de $(\hat{w}_{e+q}, \hat{p}_{e+q}, s_{e+q})$ du fils correspondant à $x_k = 0$. Nous rappelons que la création d'un nœud fils correspondant à $x_k = 1$ revient à la conservation du nœud père dans la liste des nœuds.
- Le cas où $k = s_e$, est traité différemment : la variable de base $s_{e+q}$, du fils correspondant à $x_k = 0$, est mise à jour comme suit :

$$s_{e+q} = s_{e+q} + 1,$$

dans ce cas les valeurs de $\hat{w}_{e+q}$ et $\hat{p}_{e+q}$ sont identiques aux valeurs de $\hat{w}_e$ et $\hat{p}_e$. Par ailleurs, le nœud fils correspondant à $x_k = 1$ doit être étiqueté comme nœud non prometteur dans le CPU ; ce qui est fait en posant $U_e = 0$.

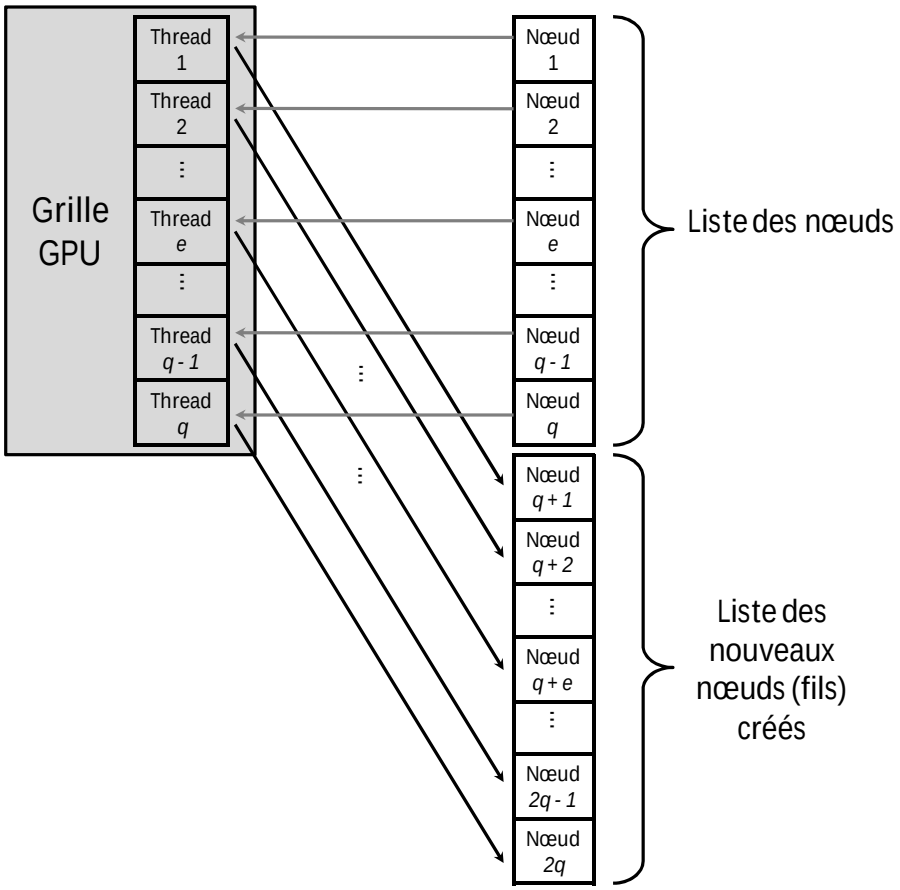



FIGURE V.3 – Création de nouveaux nœuds par la grille de threads sur GPU.

Dans le *Kernel* 1, nous notons que l'utilisation de l'instruction conditionnelle *if* peut conduire à une divergence de *threads* au sein d'un même *warp* (voir le paragraphe IV.4.2). Toutefois, dans cette mise en œuvre, seules les opérations avec les registres sont incluses dans la partie conditionnelle et les opérations d'écriture sur la mémoire globale sont exécutées simultanément par les threads à la fin du *Kernel* 1. Cela s'est avéré plus efficace que la mise en œuvre proposée dans [BOU 12].

**Kernel 1 :** Création des nouveaux nœuds.

```
global_void Kernel 1(int * ŵ , * p̂ , int *s, int *U, int k, int q)
{
int e = blockIdx.x * blockDim.x + threadIdx.x ;
int se = s[e], we = ŵ[e] , pe = p̂[e] , Ue = U[e];
                    /* Branche divergente calculée dans les registre des threads */
if (k < se)
   {
   we = we – w[k] ;
   pe = pe – p[k] ;
   }
else
{
se = se + 1 ;
Ue = 0 ;
   }
                    /* Partie accédant à la mémoire globale exécutée */
ŵ[e + q] = we ;
p̂[e + q] = pe ;
s[e + q] = se ;
U[e] = Ue ;
}
```

**Calcul de bornes**

La procédure de calcul parallèle de bornes, calcule les deux bornes des nouveaux nœuds. Le calcul de borne se fait par le *Kernel* 2 qui est une boucle de $n - h$ itérations, où $h = \max\{k, s\}$ et $s$ la première variable de base calculée lorsque le sac est vide.

**Kernel 2 :** Calcul de la borne supérieure $U_j$ et inferieure $L_j$.

```
global_void Kernel 2(int * ŵ , int * p̂ , int * s, int * U, int * L, int q, int h)
{
int j = blockIdx.x * blockDim.x + threadIdx.x + q ;
int sb = s[j], wj= ŵ[ j ] , pj = p̂[ j ] ;
int i = h, wi, pi, wu, pu = 0, wb, pb ;
While (i ≤ n)
   {
   wi = w[i] ;
   pi = p[i] ;
```

```
            /* Calcul de wu, pu, wb, pb, sb pour l'obtention de la borne supérieure U_j*/
    if (i ≥ sb)
        {
          if (wj + wi ≤ c)
            {
             wj = wj + wi ;
             pj = pj + pi ;
             }
       else if (pu == 0)
             {
             ws = wi ;
             pb = pi ;
             wu = wj ;
             pu = pj ;
             sb = i ;
             }
            }
       if ( __all(pu) )  break ;
       __syncthreads() ;
      i = i + 1 ;
      }
                                        /* Calcul de la borne inférieure L_j */
for ( ; i ≤ n; i++)
        {
         wi = w[i];
         pi = p [i];
         if (wj + wi ≤ c)
           {
            wj = wj + wi;
            pj = pj + pi;
            }
         }
       /* Calcul de la borne supérieure U_i et écriture en mémoire globale du quintuplet */
                                        /* (ŵ_j, p̂_j, s_j, U_j, L_j) */
ŵ[j] = wu ;
p̂[j] = pu ;
s[j] = sb;
pu = pu + (c − wu) × pb/wb;
U[j] = pu ;
L[j] = pj ;
}
```

A chaque nouvelle itération, un nouvel article est considéré et les valeurs de $\hat{w}_j$ et $\hat{p}_j$, la borne supérieure $U_j$ et la borne inférieure $L_j$ sont mises à jour.

Lorsque le *Kernel* 2 est lancé, le $j^{ème}$ thread met à jour à l'itération *i*, le quintuplet $(\hat{w}_j, \hat{p}_j, s_j, U_j, L_j)$ si l'article *i* n'a pas encore été pris en compte par ce nœud, c'est-à-dire si $i \geq s_j$.

Nous pouvons alors distinguer, dans le *Kernel* 2, trois parties principales : la première boucle *while*, la deuxième boucle *for* et enfin la troisième partie commençant à de l'instruction ( $\hat{w}[j]$ = *wu* ;).

La première boucle est utilisée pour mettre à jour le quadruplet $(\hat{w}_j, \hat{p}_j, s_j, U_j)$ qui est stocké dans la mémoire globale du GPU. Cette boucle fait usage d'instructions conditionnelles *if* qui conduisent à une divergence de *threads* au sein d'un même *warp.* Ceci peut causer une perte d'efficacité due à un ralentissement du *Kernel* dans le cas où la partie conditionnelle comprend des calculs et des accès en mémoire globale. Pour éviter ce problème, on sépare la partie calcul de la partie liée à l'écriture dans la mémoire globale à travers l'ajout de nouvelles variables *wu*, *pu*, *sb* stockées dans des registres et utilisées pour mettre à jour le triplet $(\hat{w}_j, \hat{p}_j, s_j)$.

Pour réduire le nombre d'opérations dans les parties conditionnelles, $U_j$ est calculé simultanément entre les threads dans la troisième partie du *Kernel* 2 (par l'opération $pu = pu + (c - wu) \times \frac{pb}{wb}$). Ceci nécessite de récupérer le poids et le profit de la variable de base $s_j$ dont la valeur va différer entre les threads d'un même *warp*. L'accès à ces deux données dans la mémoire de texture se fait de manière non contiguë et implique une latence de l'accès mémoire. Ceci peut être évité en utilisant deux autres variables nommé *wb* et *pb*, stockées dans des registres et utilisées pour récupérer le poids et le profit de la variable de base $s_j$ dans la première partie du *Kernel* 2 (en recopiant *wi* et *pi*).

Les parties conditionnelles comprennent alors des calculs sur des registres, ce qui a pour effet de réduire les pertes d'efficacité.

La partie relative à l'écriture dans la mémoire globale, autrement dit la mise à jour du quintuplet $(\hat{w}_j, \hat{p}_j, s_j, U_j, L_j)$ est faite à la fin du *Kernel* 2 (la troisième partie commençant par l'instruction $\hat{w}[j] = wu$ ;). Afin d'assurer que les threads d'un même *warp* s'exécutent de

manière simultanée, la première boucle *while* est synchronisée à la fin de chaque itération et les threads d'un même *warp* sortent de la boucle au même moment après avoir déjà mis à jour leurs triplés $(\hat{w}_j, \hat{p}_j, s_j)$ correspondants. Ceci est fait en utilisant une instruction sur *warp* (*if* (__*all*(*pu*)) *break*;) sachant que la variable *pu* a été initialisée à 0. L'instruction __*all*(*pu*) évalue alors la valeur de *pu* pour tous les threads d'un *warp* et retourne une valeur non nulle si et seulement si la valeur de *pu* est non nulle pour chacun des ces threads.

La seconde boucle *for* sert à calculer la borne inférieure $L_j$. En pratique, cette boucle est plus grande que la première boucle *while* en termes de nombre d'itérations. En *déroulant* cette boucle *for* à l'aide de l'instruction (*#pragma unroll)*, on améliore de 10% le temps de calcul global de la mise en œuvre sur GPU de la méthode de Branch et Bound. Ne connaissant pas a priori, le nombre d'itérations de cette boucle *for*, nous avons fait le choix de dérouler partiellement cette dernière en utilisant trois boucles *while* où la première déroule 100 fois la boucle *for*, la seconde déroule 10 fois la boucle *for* et la dernière boucle sur les itérations résiduelles jusqu'à ce que *i* soit égal *n*. Par exemple, considérons le cas d'une boucle *for* qui doit être itérée 392 fois, il est préférable de remplacer cette dernière par les trois boucles *while* décrites précédemment ce qui revient à itérer 3 fois la première boucle *while* puis 9 fois la seconde boucle *while* puis 2 fois la dernière boucle.

Nous noterons que la borne inférieure $L_j$ est obtenue seulement à la fin de la procédure, c'est à dire à l'itération *n*.

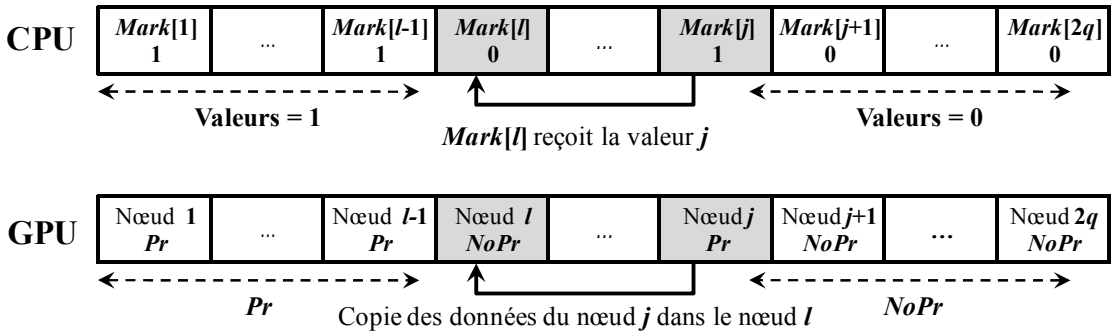
**Recherche de la meilleure borne inférieure**

Trouver la meilleure borne inférieure $\overline{L}$ revient à obtenir un maximum dans un ensemble de valeurs. Ceci se fait sur le GPU par une méthode de réduction en utilisant l'opération atomique *atomicMax* appliquée au tableau des bornes inférieures (voir [HAR 07a]).

**Marquage des nœuds à éliminer via le tableau *Mark***

À ce stade, la taille de la liste de nœuds est augmentée par le nombre *q* de nœuds récemment créés, c-à-d, *q* est mis à jour en lui assignant une valeur qui est deux fois plus grande que sa valeur précédente. Ensuite, nous utilisons un *Kernel* GPU qui permet de marquer les nœuds non prometteurs et renvoie un tableau nommée *Mark*. Un nœud *e* est considéré comme étant : non prometteur (*NoPr*) si $U_e \leq \overline{L}$ et prometteur (*Pr*) dans le cas contraire. Ainsi, si un nœud *e* est prometteur, l'élément *Mark*[*e*] prend la valeur 1, autrement, il prend la valeur 0.

À la fin de cette étape, le tableau *Mark* est transféré sur le CPU.




***Pr***: nœud Prometteur .
***NoPr***: nœud Non Prometteur .

FIGURE V.4 – Etape d'élagage : Procédure de substitution sur GPU d'un nœud non prometteur *l* après affectation sur CPU de l'adresse *j* du nœud qui va le remplacer.

**Substitution et concaténation de la liste des nœuds**

Lorsque le tableau *Mark* est récupéré, l'élément *Mark*[*l*] mis à zéro lors de l'étape précédente contient l'adresse *j* du nœud prometteur situé dans la partie droite de la liste des nœuds (la partie se trouvant sur le GPU dans la Figure V.4). Ensuite, l'étape de concaténation est effectuée via un *Kernel* GPU où un thread *l* copie les données $(\hat{w}_j, \hat{p}_j, s_j, U_j)$ du nœud prometteur *j* dans le quadruplet $(\hat{w}_l, \hat{p}_l, s_l, U_l)$. A la fin de cette étape, la taille de la liste est diminuée du nombre de nœuds non prometteurs. Par la suite, si le nombre de nœuds *q* est inferieur à 192, la liste est transférée sur le CPU, sinon un nouvel article est considéré.

Les deux étapes précédentes sont utilisées afin d'éviter le transfert de toute la liste de nœuds sur le CPU à chaque itération. Si le nombre de nœuds reste important ( $q \geq 192$ ), le GPU effectue les différentes étapes sur la liste de nœuds et seul un petit nombre de communications de données (le tableau *Mark*), entre le GPU et CPU, est nécessaire à chaque itération de l'algorithme de Branch and Bound.

On note que les trois étapes précédentes sont très simples ; aussi nous ne les détaillons pas dans ce mémoire.

### V.4.4 Calculs sur CPU

**Algorithme de Branch and Bound, partie séquentielle**

Comme nous l'avons dit précédemment, si la liste des nœuds est petite, il n'est pas utile de lancer les différents *Kernels* de Branch and Bound sur GPU puisque l'occupation de ce dernier serait très faible, et les temps de calculs sur GPU ne couvriront pas les temps de communication entre GPU et CPU.

**Affectation des adresses de substitutions dans le tableau *Mark***

Cette procédure commence après que le tableau *Mark* ait été transféré du GPU au CPU. Les éléments du tableau *Mark* qui ont été mis à zéro lors de l'étape précédente, c.-à-d. ceux qui correspondent aux nœuds non prometteurs, sont remplacés à l'aide d'une procédure itérative qui commence à partir du début du tableau *Mark*. On affecte alors à chaque élément *Mark*[*l*] = 0, l'adresse *j* du nœud prometteur qui doit remplacer le nœud non prometteur *j* dans la liste des nœuds. Les différentes étapes de cette procédure sont présentées ci-dessous (voir aussi la Figure V.4).

- Rechercher à partir de la fin du tableau *Mark*, l'élément non nul *Mark*[*j*] = 1 (correspondant au nœud prometteur *j*).
- Copier l'adresse *j* dans l'élément *Mark* [*l*] c.-à-d *Mark*[*l*] = *j*.
- Mettre à jour la taille de la liste comme suit :

$$q = j - 1.$$

## V.5 Résultats expérimentaux

Nous présentons maintenant les résultats numériques obtenus avec un CPU et un système avec un CPU/GPU. La machine que nous avons utilisée est un Dell Precision T7500 Westmere avec un processeur Quad-Core Intel Xeon E5640 2,66 GHz, 12 Go de mémoire principale et une carte GPU NVIDIA Tesla C2050.

Nous avons utilisé CUDA 3.2 pour le code parallèle et *gcc* pour le code séquentiel. Nous avons considéré des problèmes de sac à dos de type *fortement corrélés* engendrés aléatoirement. Ils présentent les caractéristiques suivantes:

- $w_i$, $i \in \{1, ..., n\}$, est pris au hasard dans [1, 100],

- $p_i = w_i + 10,\ i \in \{1, ..., n\}$,
- $c = \frac{1}{2}\sum_{i=1}^{n} w_i$

Nous avons aussi testé d'autres types de problèmes de sac à dos à savoir les problèmes *faiblement corrélés* et *non corrélés*. Ces derniers s'avèrent très faciles et leurs résolutions n'engendrent pas assez de nœuds pour faire intervenir le GPU. En effet, l'élagage est particulièrement important pour ces types de problèmes et tous les calculs sont effectués sur le CPU.

Les problèmes *fortement corrélés* engendrent, quant à eux, assez de nœuds pour faire intervenir le GPU. Certains problèmes engendrent même un nombre de nœuds qui sature la mémoire de la carte graphique, soit $q > 40000000$. Notons que les problèmes qui ont été utilisés ont été mis en ligne sur le site de notre groupe, voir la référence [LAA 12].

Pour chaque problème, les résultats affichés dans les tableaux suivants correspondent à une moyenne de dix instances.

| Taille *n* du problème | Temps (*s*) CPU | Temps (*s*) CPU-GPU | Speedup |
|---|---|---|---|
| 100 | 1,59 | 0,15 | 10,6 |
| 200 | 4,85 | 0,32 | 14,72 |
| 300 | 9,82 | 0,52 | 18,85 |
| 400 | 10,94 | 0,45 | 23,8 |
| 500 | 13,39 | 0,52 | 25,60 |
| 800 | 23,27 | 0,56 | 40,65 |
| 1000 | 39,31 | 0,76 | 51,99 |

TABLEAU V.1 – Comparaison des temps moyens de calcul de l'algorithme B&B séquentiel et parallèle.

Le Tableau V.1 présente le temps de calcul de l'algorithme de Branch and Bound sur un CPU et un système CPU-GPU. Nous voyons qu'une accélération substantielle peut être obtenue en utilisant la GPU Tesla C2050.

L'algorithme de Branch and Bound parallèle proposé permet de réduire fortement les temps de calcul. En effet, plus les cœurs de calcul de la Tesla C2050 sont disponibles pour un calcul donné, plus les threads sont exécutés en parallèle et meilleure est la performance globale.

Ainsi nous pouvons voir que les accélérations (speedup) obtenues augmentent avec la taille *n* du problème. L'accélération peut atteindre une valeur égale à 52.

L'accélération dépend fortement de la taille et de la difficulté de l'exemple considéré. En particulier, les meilleures accélérations ont été obtenues pour des instances à grand nombre de nœuds.

| Taille *n* du problème / Nombre *q* de nœuds | 100 | 1000 | 10000 |
|---|---|---|---|
| 192 | 0,43 | 1,54 | 2,89 |
| 10000 | 1,33 | 28,42 | 61,28 |
| 100000 | 25,56 | 62,31 | 102,00 |
| 200000 | 25,66 | 63,17 | 106,31 |
| 1000000 | 25,42 | 62,52 | 108,98 |
| 10000000 | 25,66 | 70,68 | 108,41 |

TABLEAU V.2 – *Speedup* de l'étape de calcul de borne.

Dans le Tableau V.2, nous présentons l'accélération obtenue pour l'étape de calcul de borne uniquement. Celle-ci varie en fonction de deux facteurs :

- le premier facteur est le nombre *q* de nœuds dans la liste ; plus on rajoute de nœuds meilleure est l'occupation de la carte graphique, ce qui permet d'avoir une bonne accélération qui se stabilise par la suite pour $q > 100000$.
- le deuxième facteur est la taille *n* du problème KP. Nous constatons que l'accélération de l'étape de calcul de bornes augmente avec la taille *n* du problème. Cela a été permis grâce aux différentes optimisations que nous avons apportées au *Kernel* 2.

La meilleure accélération (speedup de 52) présentée au Tableau V.1 correspond aux accélérations présentées dans la colonne 3 au Tableau V.2, c.-à-d. pour $n$ = 1000. Nous voulons aussi insister sur le fait que nous aurons de meilleures performances quand il sera possible de tester de plus grand problèmes. Par exemple pour des problèmes de taille n = 10000, les résultats (de l'étape de calcul de bornes) présentés dans la dernière colonne du Tableau V.2 laissent présager des accélérations supérieures à 52.

## V.6 Conclusions et perspectives

Dans ce chapitre, nous avons proposé une mise en œuvre parallèle via CUDA de la méthode de Branch and Bound pour le problème du sac à dos, sur un système CPU-GPU avec un GPU Tesla C2050. Nous avons montré comment mettre à profit le parallélisme offert par cette nouvelle architecture massivement parallèle afin d'accélérer les différentes étapes de la méthode de Branch and Bound, ceci en identifiant les tâches de cet l'algorithme qui peuvent se paralléliser de manière performante. Nous nous sommes concentrés sur la minimisation des effets des chemins divergents et la synchronisation des threads d'un même warp.

Les résultats expérimentaux ont montré que des problèmes difficiles pouvaient être résolus dans des temps de calculs réduits permettant d'avoir des accélérations de l'ordre de 52. Ce travail montre l'intérêt de l'utilisation des GPUs pour résoudre des problèmes difficiles d'optimisation combinatoire.

Nous pensons que des accélérations plus élevées peuvent être obtenues en testant des problèmes plus grands ou en mettant en œuvre des clusters de $GPU_s$.

Les perspectives à ce travail seraient d'abord d'incorporer l'étape de séparation et de calcul de bornes dans un seul *Kernel*. En effet, on comptera une seule étape de lecture et écriture en mémoire globale du quintuplet $\left(\hat{w}_j, \hat{p}_j, s_j, U_j, L_j\right)$. De plus, il serait intéressant de mettre en œuvre toute l'étape d'élagage entièrement sur GPU par l'incorporation d'algorithmes de tri tel que l'algorithme du Quick sort, ce qui permettrai d'abord de diminuer considérablement les communications entre CPU-GPU et ensuite de pouvoir tester la stratégie de parcours «meilleur d'abord» de la liste des nœuds.

A long terme, nous prévoyons de mettre en oeuvre la méthode de réduction et fixation de variables à chaque étape de séparation de l'algorithme de Branch and Bound. Ceci nécessitera de revoir toute la structure de l'algorithme que nous avons présenté dans ce chapitre.

Il sera aussi intéressant de proposer une version multi-GPU de la méthode de Branch and Bound mettant à contribution plusieurs cartes GPU disponibles dans une machine ou dans un cluster.

# Chapitre VI

# Mises en œuvre CPU-GPU et Multi-GPU de la méthode du Simplexe

## Sommaire



## VI.1 Introduction

Dans ce chapitre, nous présentons l'apport du GPU à la résolution de problèmes de programmation linéaire. Nous présentons des mises en œuvre CPU-GPU et multi-GPU de l'algorithme du Simplexe.

Nous commençons par un bref rappel mathématique sur l'algorithme du Simplexe et nous présentons les différentes étapes de la version de l'algorithme du Simplexe que nous avons mise en œuvre. Nous présenterons par la suite comment mettre en œuvre de manière parallèle cet algorithme dans un premier temps sur un système CPU-GPU et ensuite sur plusieurs GPUs. La structure des données en mémoire sera décrite ainsi que le partage de tâches et les différentes communications entre le CPU et le GPU. Finalement, Nous donnerons des résultats expérimentaux qui confirment l'intérêt de l'utilisation des GPUs afin de résoudre les problèmes de programmation linéaire.

Afin de faciliter la présentation du tableau du Simplexe, la valeur du critère est notée $x_0$ dans ce chapitre.

## VI.2 Rappels mathématique sur la méthode du Simplexe

Les problèmes de programmation linéaire, en anglais Linear Programming (LP), consistent à maximiser (ou minimiser) une fonction linéaire, la fonction objectif, sous un ensemble de contraintes linéaires. Plus formellement, un problème LP s'écrit de la manière suivante :

$$\begin{aligned} &\max x_0 = \widetilde{c}\widetilde{x}, \\ &s.c.\ \widetilde{A}\widetilde{x} \le \widetilde{b}, \\ &\widetilde{x} \ge 0, \end{aligned} \tag{VI.1}$$

avec

$$\widetilde{c} = (\widetilde{c}_1, \widetilde{c}_2, \cdots, \widetilde{c}_n) \in \Re^n,$$

$$\widetilde{A} = \begin{pmatrix} \widetilde{a}_{11} & \widetilde{a}_{12} & \cdots & \widetilde{a}_{1n} \\ \widetilde{a}_{21} & \widetilde{a}_{22} & \cdots & \widetilde{a}_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ \widetilde{a}_{m1} & \widetilde{a}_{m2} & \cdots & \widetilde{a}_{mn} \end{pmatrix} \in \Re^{m \times n},$$

et

$$\widetilde{x} = (\widetilde{x}_1, \widetilde{x}_2, \cdots, \widetilde{x}_n)^T.$$

*n* étant le nombre de variables et *m* le nombre de contraintes.

Ces $m$ contraintes de type inégalité peuvent être transformées en contraintes égalité, en introduisant $m$ nouvelles variables $x_{n+h}, h \in \{1, 2, \ldots, m\}$, appelées *variables d'écart*, de sorte que:

$$\tilde{a}_{h1}\tilde{x}_1 + \tilde{a}_{h2}\tilde{x}_2 + \cdots + \tilde{a}_{hn}\tilde{x}_n + x_{n+h} = b_h, h \in \{1, 2, \ldots, m\},$$

avec $x_{n+h} \geq 0$ et $c_{n+h} = 0$.

Ainsi, la forme standard du problème de programmation linéaire peut être écrite de la manière suivante:

$$\begin{aligned} &\max\ x_0 = cx, \\ &s.c.\ Ax = b, \\ &x \geq 0, \end{aligned} \qquad \text{(VI.2)}$$

où

$$c = (\tilde{c}_1, \cdots, \tilde{c}_n, 0, \cdots, 0) \in \Re^{(n+m)}, A = \left(\tilde{A}, I_m\right) \in \Re^{m\times(n+m)},$$

$I_m$ étant la matrice identité de dimension $m \times m$ et $x = (\tilde{x}_1, \cdots, \tilde{x}_n, x_{n+1}, x_{n+2}, \cdots, x_{n+m})^T$.

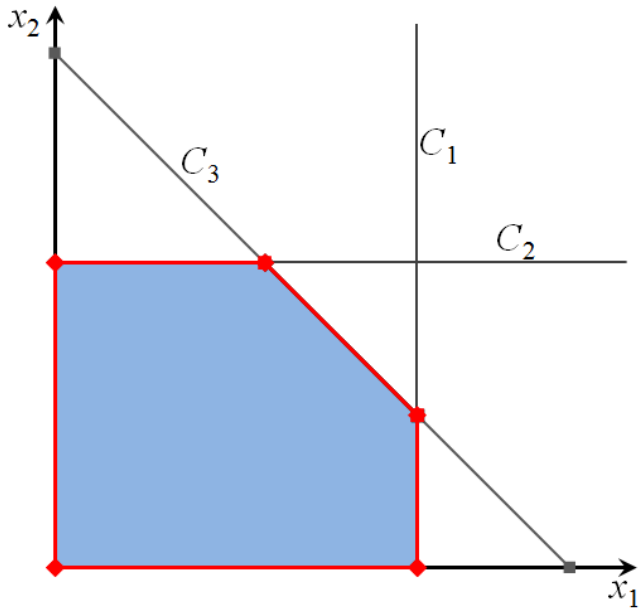



FIGURE VI.2 – Représentation géométrique du polytope pour un exemple avec $n = 2$ et $m = 3$.

En 1947, George Dantzig a proposé l'algorithme du Simplexe pour résoudre les problèmes de programmation linéaire (voir [DAN 57]). L'ensemble des solutions admissibles d'un problème LP est située sur les sommets d'un polytope, ensemble convexe dont la frontière est constituée par des bouts d'hyperplans.

L'algorithme du Simplexe procède de la manière suivante : en considérant que la solution optimale est forcement sur un sommet du polytope et en partant initialement d'un sommet (qui est une solution admissible du problème LP), l'algorithme du Simplexe se déplace sur le polytope (voir la Figure VI.3) le long d'une arrête vers un autre sommet du polytope (donc une autre solution admissible) de manière à augmenter la fonction objectif. Ce déplacement, qui correspond analytiquement à une opération matricielle appelée *pivotage*, est réitéré jusqu'à atteindre la meilleure solution admissible qui est la solution optimale.

Différentes versions de la méthode du Simplexe ont été proposées. Dans ce travail, nous considérons la méthode proposée par Garfinkel et Nemhauser [GAR 72] qui améliore l'algorithme de Dantzig en réduisant à la fois le nombre d'opérations et l'occupation mémoire.

Les colonnes de $A$ sont permutées de sorte que $A = (B, N)$, où $B$ est une matrice inversible de dimension $m \times m$. Cette dernière est appelée la *matrice de base* du problème LP.

Nous notons $x_B$, le sous-vecteur de $x$ de dimension $m$ correspondant aux *variables de base* associées à la matrice de base $B$. De la même manière, $x_N$ dénote le sous-vecteur de $x$ de dimension $n$ correspondant aux *variables hors base* associées à la matrice $N$.

En posant : $c = (c_B, c_N)$, le problème (VI.2) peut alors être écrit comme suit :

$$\begin{bmatrix} x_0 \\ x_B \end{bmatrix} = \begin{bmatrix} c_B B^{-1} b \\ B^{-1} b \end{bmatrix} - \begin{bmatrix} c_B B^{-1} N - c_N \\ B^{-1} N \end{bmatrix} x_N . \qquad \text{(VI.3)}$$

En prenant $x_N = 0, x_B = B^{-1}b$ et $x_0 = c_B B^{-1} b$, on obtient une solution initiale admissible du problème LP.

**Tableau du Simplexe**

Afin de construire le *tableau du Simplexe*, encore appelé Simplex Tableau en anglais, nous introduisons les notations suivantes:

- $\begin{bmatrix} s_{0,0} \\ s_{1,0} \\ \vdots \\ s_{m,0} \end{bmatrix} \equiv \begin{bmatrix} c_B B^{-1} b \\ B^{-1} b \end{bmatrix}$

- $\begin{bmatrix} s_{0,1} & s_{0,2} & \cdots & s_{0,n} \\ s_{1,1} & s_{1,2} & \cdots & s_{1,n} \\ \vdots & \vdots & \ddots & \vdots \\ s_{m,1} & s_{m,2} & \cdots & s_{m,n} \end{bmatrix} \equiv \begin{bmatrix} c_B B^{-1} N - c_N \\ B^{-1} N \end{bmatrix}$

(VI.3) peut ainsi être réécrit de la manière suivante

$$\begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_m \end{bmatrix} = \begin{bmatrix} s_{0,0} \\ s_{1,0} \\ \vdots \\ s_{m,0} \end{bmatrix} - \begin{bmatrix} s_{0,1} & s_{0,2} & \cdots & s_{0,n} \\ s_{1,1} & s_{1,2} & \cdots & s_{1,n} \\ \vdots & \vdots & \ddots & \vdots \\ s_{m,1} & s_{m,2} & \cdots & s_{m,n} \end{bmatrix} x_N . \qquad \text{(VI.4)}$$

Ainsi, à partir de l'équation VI.4, nous construisons le *tableau du Simplexe*, comme illustré ci-dessous.

| | | | Variables hors base | | | | | |
|---|---|---|---|---|---|---|---|---|
| | | | $-x_{N_1}$ | $-x_{N_2}$ | $\cdots$ | $-x_{N_k}$ | $\cdots$ | $-x_{N_n}$ |
| Variables de base | $x_0$ | $s_{0,0}$ | $s_{0,1}$ | $s_{0,2}$ | $\cdots$ | $s_{0,k}$ | $\cdots$ | $s_{0,n}$ |
| | $x_{B_1}$ | $s_{1,0}$ | $s_{1,1}$ | $s_{1,2}$ | $\cdots$ | $s_{1,k}$ | $\cdots$ | $s_{1,n}$ |
| | $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ | $\ddots$ | $\vdots$ | $\ddots$ | $\vdots$ |
| | $x_{B_r}$ | $s_{r,0}$ | $s_{r,1}$ | $s_{r,2}$ | $\cdots$ | $s_{r,k}$ | $\cdots$ | $s_{r,n}$ |
| | $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ | $\ddots$ | $\vdots$ | $\ddots$ | $\vdots$ |
| | $x_{B_m}$ | $s_{m,0}$ | $s_{m,1}$ | $s_{m,2}$ | $\cdots$ | $s_{m,k}$ | $\cdots$ | $s_{m,n}$ |

TABLE VI.1 – Tableau du Simplexe.

En considérant des problèmes de programmation linéaire à contrainte inégalité, voir (VI.1) et en ajoutant les variables d'écart, voir (VI.2), la décomposition de la matrice $A = (B, N)$ apparait de manière simple.

En effet, par identification $N = \widetilde{A}$ et $B = I_m$, $I_m$ étant une matrice inversible ce qui permet d'avoir $B^{-1} = I_m$, une solution initiale admissible peut être écrite comme suit:

$x_N = \widetilde{x}(0,0,\cdots,0) \in \Re^n$ et $x_B = B^{-1}b = b$. De plus, si $x_B \geq 0$, $x$ est appelée solution de base du problème de programmation linéaire.

En commençant par cette solution, à chaque itération, l'algorithme du Simplexe tente de remplacer une variable de base, appelée *variable sortante*, par une variable hors base, appelée *variable entrante*, de sorte que la fonction objectif augmente. Ainsi, une meilleure solution admissible est obtenue par la mise à jour du tableau du Simplexe. Plus explicitement, l'algorithme du Simplexe applique à chaque itération, les étapes suivantes:

- **Etape 1:** Calculer l'indice $k$ de la plus petite valeur négative de la première ligne du tableau du Simplexe :

$$k = \arg \min_{j=1,2,\cdots,n} \left\{ s_{0,j} \mid s_{0,j} < 0 \right\}.$$

  La variable $x_{N_k}$ est la *variable entrante*. Si aucun indice n'est trouvé, autrement dit si $s_{0,j} \geq 0$, alors la solution actuelle est optimale. Dans le cas contraire, l'algorithme passe à l'étape suivante.

- **Etape 2 :** Calculer les ratios $\theta_{i,k} = \dfrac{s_{i,0}}{s_{i,k}}$, $i = 1, 2, \ldots, m$ ensuite calculer l'indice $r$ comme suit :

$$r = \arg \min_{i=1,2,\cdots,m} \left\{ \theta_{i,k} \mid s_{i,k} > 0 \right\}.$$

  La variable $x_{B_r}$ est la *variable sortante*. Si l'indice $r$ est trouvé, l'algorithme passe à la dernière étape, sinon, l'algorithme s'arrête et le problème est *non borné*.

- **Etape 3:** Calculer la nouvelle solution admissible en mettant à jour la base. La variable $x_{B_r}$ sort de la base et la variable $x_{N_k}$ entre dans la base. Dans le cas pratique, la $k^{ème}$

colonne est sauvegardée et renommée « *ancienne* $k^{ème}$ colonne ». Par la suite le tableau du Simplexe est actualisé comme suit:

1) Diviser les éléments de la $r^{ème}$ ligne par l'élément pivot $s_{r,k}$ :

$$s_{r,j} := \frac{s_{r,j}}{s_{r,k}}, \quad j = 0, 1, 2, ..., n.$$

2) Multiplier les éléments de la nouvelle $r^{ème}$ ligne par l'élément $s_{i,k}$ et la soustraire à la $i^{ème}$ ligne,

   $i = 0,\ 1,\ ...,\ m,\ i \neq r$ :

$$s_{i,j} := s_{i,j} - s_{r,j} s_{i,k}, \quad j = 0, 1, 2, ..., n.$$

3) Remplacer dans le tableau du Simplexe, les éléments de la $k^{ème}$ colonne par ceux de *l'ancienne* $k^{ème}$ colonne divisés par l'élément $s_{r,k}$ à l'exception de l'élément pivot $s_{r,k}$ qui sera remplacé par la valeur $\frac{1}{s_{r,k}}$ :

$$s_{i,k} := -\frac{s_{i,k}}{s_{r,k}}, \qquad i = 0, 1, 2, ..., m, \qquad i \neq r,$$

   et

$$s_{r,k} \text{ devient } \frac{1}{s_{r,k}}.$$

   Cette étape de l'algorithme du Simplexe est la plus coûteuse en termes de temps de calcul.

   A la fin de cette étape, l'algorithme revient à l'étape 1.

L'algorithme du Simplexe s'arrête dans deux cas précis :

- Lorsque la solution optimale est atteinte (dans ce cas, le problème LP est résolu).
- Lorsque le problème LP est non borné (dans ce cas, aucune solution ne peut être trouvée).

La version de l'algorithme du Simplexe proposée par Garfinkel et Nemhauser est particulièrement intéressante dans le cas des problèmes denses puisque la taille de la matrice

manipulée (tableau du Simplexe) est $(m+1)\times(n+1)$ au lieu de $(m+1)\times(n+m+1)$ pour la méthode standard de Dantzig. Ceci a comme conséquence la diminution du taux d'occupation mémoire ainsi que la réduction du temps de calcul permettant ainsi de tester des problèmes de plus grande taille.

## VI.3 Résolution parallèle, état de l'art

La mise en œuvre parallèle de la méthode du Simplexe, dans le but de diminuer les temps de résolution des problèmes de programmation linéaire, a retenu l'attention de nombreux chercheurs [BIX 00], [PFE 76], [LEN 95], [BAD 06], [LEA 08] et [HAL 10].

Les approches d'exploitation du parallélisme dans la méthode du Simplexe peuvent être classées selon la manière dont est exploitée la structure (matrice creuse ou dense) du problème LP à résoudre. Nous présentons ci-après un bref état de l'art.

- *Simplexe parallèle utilisant l'algèbre linéaire dense*

La méthode du Simplexe standard *dense* et la méthode du Simplexe révisé *dense* ont été mises en œuvre en parallèle à plusieurs reprises. On note ainsi les travaux de Zenios dans [ZEN 89] et Luo et al. dans [LUO 91] qui ont proposé des mises en œuvre sur machines à mémoire distribuée. Cvetanovic et al. ont proposé dans [CVE 91] une mise en œuvre de la méthode du Simplexe standard sur machine à mémoire partagée. Eckstein et al. [DEM 98] ont proposé une mise en œuvre parallèle de la méthode du Simplexe standard dense et de la méthode du Simplexe révisé dense, sur des machines massivement parallèles Connexion CM-2 et CM-5. Dans un travail plus récent Badr et al. [BAD 06] ont proposé une mise en œuvre de la méthode du Simplexe standard dense sur une machine à huit processeurs utilisant la librairie de communication MPI (*Message Passing Interface*). Une accélération de 5 est obtenue lors de la résolution de petits problèmes LP denses engendrés aléatoirement.

Toutes ces mises en œuvre parallèles permettent d'obtenir des accélérations dans le cas où les problèmes LP résolus sont denses, ce qui n'est pas le cas lorsqu'il s'agit de résoudre des grands problèmes creux. En effet, le seul moyen pour que ces mises en œuvre

rivalisent avec les mises en œuvre séquentielles de la méthode du Simplexe révisé, est d'utiliser un très grand nombre de processeurs en parallèles.

- *Simplexe parallèle utilisant l'algèbre linéaire creuse*

La mise en œuvre de la méthode du Simplexe standard ou révisé utilisant des techniques d'algèbre linéaire creuse pour résoudre des problèmes généraux de programmation linéaire, représente un défi dans le domaine du calcul parallèle.

De nombreuses mises en œuvre parallèles ont été proposées. Ainsi Pfefferkorn et Tomlin discutent dans [PFE 76] du choix des tâches de calcul de l'algorithme du Simplexe révisé avec factorisation de l'inverse (la méthode utilisée dans les solveurs séquentiels), qui pourraient être efficacement parallélisées. Lentini et al. proposent dans [LEN 95] une mise en œuvre parallèle de la méthode du Simplexe standard creuse. Cette dernière s'est avérée moins efficace pour résoudre de grands problèmes LP creux que la méthode du Simplexe révisé séquentielle. Shu a proposé dans [SHU 95] une mise en œuvre parallèle de la méthode du Simplexe révisé sur une machine Touchstone Delta à 64 processeurs obtenant un speedup de 8 pour de grands problèmes LP. On note enfin, le travail de Bixby et Martin, qui ont proposé dans [BIX 00] une mise en œuvre parallèle de l'algorithme dual du Simplexe. Le code qui en a découlé est utilisé dans le solveur CPLEX.

Pour plus de détails sur les différentes mises en œuvre parallèles de l'algorithme du Simplexe, nous renvoyons à la référence [HAL 10].

Nous nous intéressons ici à la mise en œuvre de la méthode du Simplexe sur GPU. Certains travaux connexes ont été présentés sur la mise en œuvre parallèle d'algorithmes sur GPU pour des problèmes de programmation linéaire. O'Leary et Jung ont proposé dans [LEA 08] une mise en œuvre CPU-GPU de la méthode des points intérieurs pour les problèmes LP. Les tests effectués sur des problèmes LP d'au plus 516 variables et 758 contraintes montrent qu'une accélération substantielle peut être obtenue en utilisant le GPU pour résoudre des problèmes LP suffisamment grands et denses. Spampinato et Elster ont proposé dans [SPA 09], une mise en œuvre parallèle de la méthode du Simplexe révisé pour les problèmes LP sur GPU à l'aide des deux bibliothèques NVIDIA CUBLAS [CUBLAS] et LAPACK [LAPAC]. Les tests ont été effectués sur les problèmes engendrés aléatoirement d'au plus 2000 variables et 2000 contraintes. Cette mise en œuvre a montré une accélération très faible égale à 2.5 sur une carte

GPU NVIDIA GTX 280 par rapport à la mise en œuvre séquentielle sur un processeur Intel Core2 Quad 2,83 GHz.

Bieling, Peschlow et Martini ont proposé dans [BIE 10] une mise en œuvre de la méthode du Simplexe révisé sur GPU. Cette mise en œuvre permet d'avoir une accélération en *simple précision* relativement faible (elle est égale 18) sur une carte graphique NVIDIA GeForce 9600 GT GPU par rapport au solveur GLPK fonctionnant sur un processeur Intel Core 2 Duo 3 GHz.

La méthode du Simplexe révisé est généralement plus efficace que la méthode standard pour résoudre de grands problèmes de programmation linéaire (cf. [DAN 63] et [DAN 03]). Pour des problèmes programmation linéaire dense, les deux approches sont cependant équivalentes (cf. [DAN 03] et [MOR 97]).

Nous nous concentrons ici sur la mise en œuvre parallèle de l'algorithme du Simplexe standard sur systèmes CPU-GPU et Multi-GPU pour des problèmes de programmation linéaire dense. Ces derniers interviennent comme sous problèmes dans de nombreux domaines importants, par exemple, certaines décompositions telles que Benders ou Dantzig-Wolfe qui engendrent des problèmes LP denses. Nous faisons aussi référence à [YAR 06] et [ECK 10] pour les applications conduisant à des problèmes LP denses.

Nous développons actuellement dans l'équipe CDA du LAAS, une série de codes parallèles sur GPU destinés à être combinés afin de produire des méthodes hybrides parallèles efficaces pour la résolution de problèmes de la famille de sac à dos comme par exemple le problème du sac à dos multidimensionnel. Dans le cas de la résolution de ce dernier, nous sommes amenés à évaluer des bornes supérieures en résolvant des problèmes de programmation linéaire.

Dans la suite, nous présentons la mise en œuvre de manière parallèle de cet algorithme sur un système CPU-GPU (cf. [LAL 11a]) et un système multi-GPU (cf. [LAL 11b]).

## VI.4 Simplexe sur un système CPU-GPU

Dans la méthode du Simplexe, la majeure partie du temps de calcul est consacrée à l'étape de pivotage.

Cette étape consiste en $(m+1)\times(n+1)$ multiplications et soustractions en double précision. Ces opérations peuvent être parallélisées sur GPU. Initialement, la méthode du Simplexe a été mise en œuvre entièrement sur GPU via CUDA. Par la suite, certaines tâches, ont été finalement mises en œuvre sur CPU. Cela a permis d'obtenir de meilleures performances en termes de temps de calcul [LAL 11a].

Dans ce qui suit, nous présentons les différentes étapes de l'algorithme.

### VI.4.1 Initialisation et algorithme général

Après avoir chargé le problème de programmation linéaire de taille $m \times n$ et avoir calculé la première solution admissible de base sur CPU (si elle existe, c'est à dire si $x_B \geq 0$), le tableau du Simplexe de taille $(m+1)\times(n+1)$ est crée dans la mémoire vive du CPU. Ce tableau, nommé *TableauSimplexe*[$m$+1][$n$+1] est ensuite copié dans la mémoire globale du GPU, où les opérations de pivotage seront effectuées. Cela implique des communications importantes entre le CPU et le GPU ; celles-ci demeurent cependant négligeables par rapport au temps global d'exécution de l'algorithme du Simplexe.

Une grille de threads (Grid) est alors configurée de manière à couvrir tous les éléments du tableau *TableauSimplexe*[$m$+1][$n$+1].

La grille est composé en $h \times w$ blocs avec :

$$h = \lceil (m+1)/6 \rceil$$
$$w = \lceil (n+1)/32 \rceil$$

Chaque bloc est relatif à une sous-matrice de 6 lignes et 32 colonnes, ce qui correspond à un nombre de 192 threads par bloc (le choix de ce nombre sera expliqué par la suite). Ainsi, chaque thread de la grille créé est associé à une entrée du tableau du Simplexe (Tableau VI.1) comme illustré à la Figure VI.2.

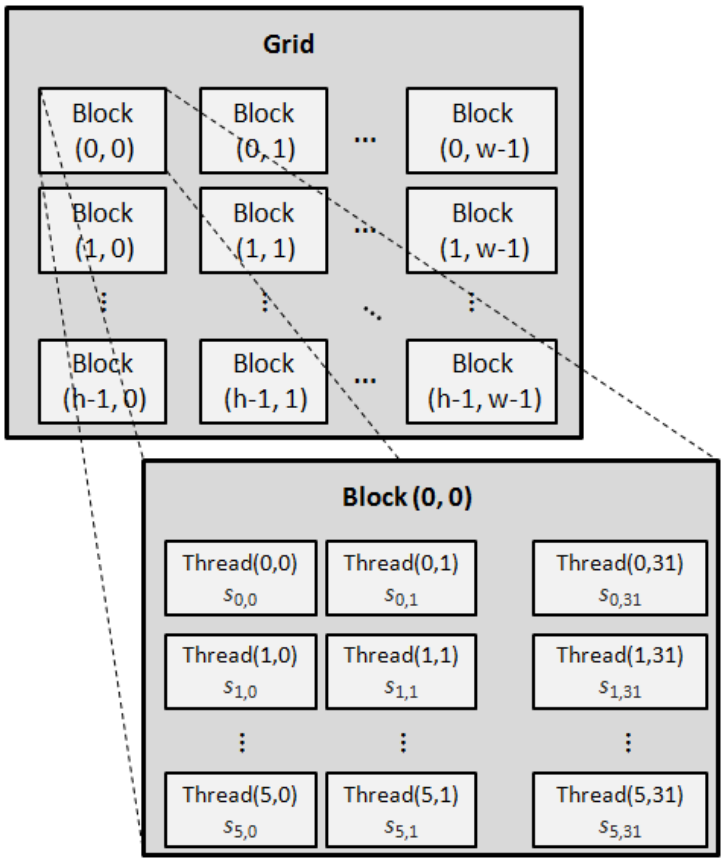



FIGURE VI.2 – Déclaration et allocation mémoire du *tableau Simplexe*.

La Figure VI.3 illustre une itération de l'algorithme du Simplexe sur système CPU-GPU.

Nous distinguons d'abord la partie initialisation qui est résumée par le transfert du tableau du Simplexe du CPU vers le GPU.

Les deux parties en gris représentent les taches exécutées sur le CPU et sur le GPU, respectivement. L'espace entre les deux parties est consacré aux différentes communications entre le CPU est le GPU.

## VI.4.2 Calcul de la variable entrante et la variable sortante

Trouver la variable entrante ou la variable sortante revient à trouver un minimum dans un ensemble de valeurs. Cela peut être fait sur le GPU via une technique de réduction inspirée des travaux de Mark Harris [HAR 07].

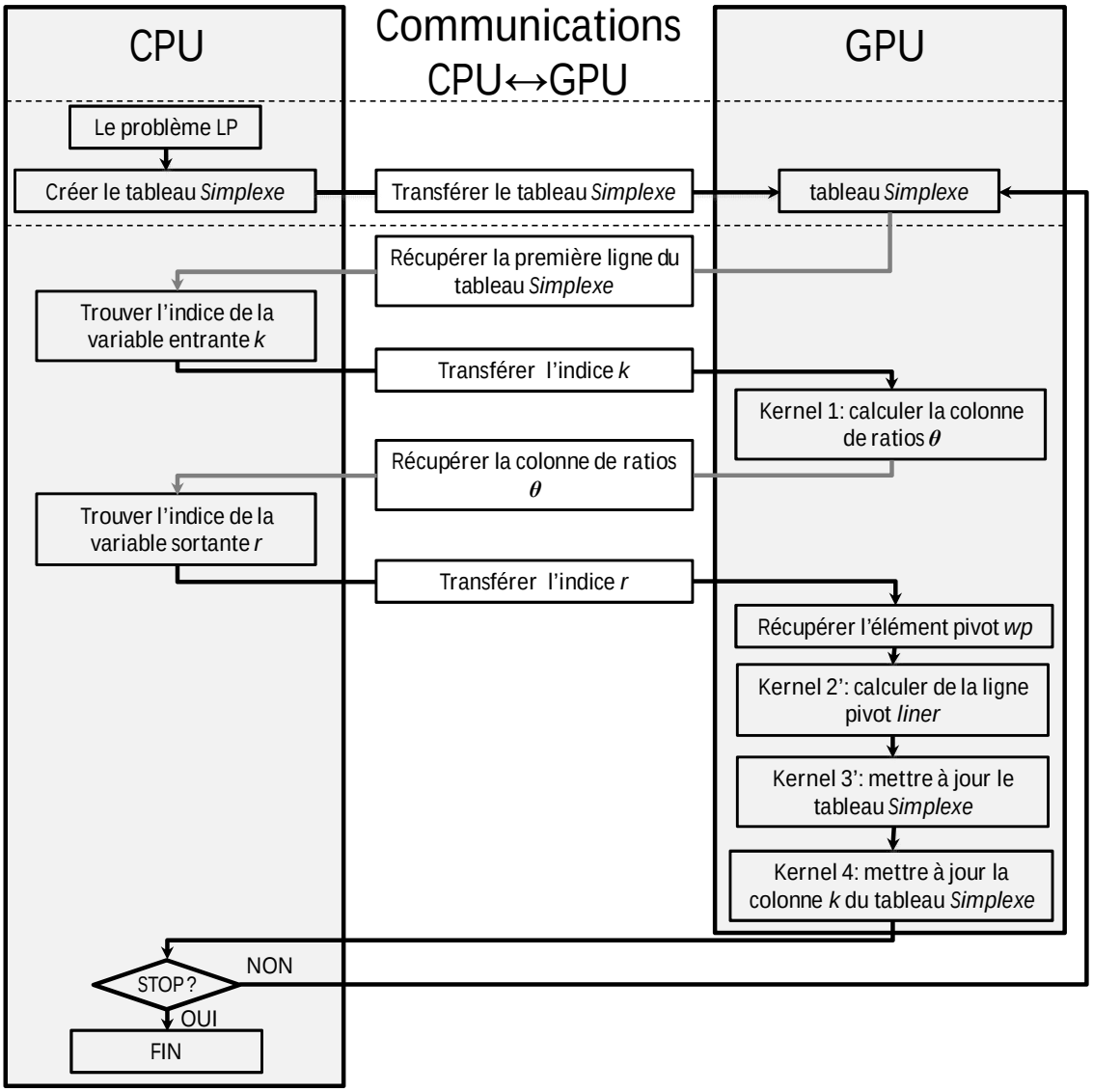



FIGURE VI.3 – Architecture globale de l'algorithme du Simplexe sur un système CPU-GPU.

Cependant, des tests expérimentaux ont montré que nous obtenons de meilleures performances en mettant en œuvre cette étape de manière séquentielle sur le CPU (la taille des problèmes testés ainsi que l'utilisation d'opérations en double précision, conduisent à de mauvaises performances sur GPU). Plus explicitement, trouver un minimum dans un tableau de 15000 valeurs, ce qui correspond à la taille maximale des vecteurs (lignes ou colonnes) traitées dans la partie expérimentale, prend en moyenne un temps égal à 0,27*ms* sur CPU, communication GPU⟶CPU comprise, et 3,17*ms* sur GPU à chaque itération de l'algorithme du Simplexe. Par ailleurs, l'utilisation des fonctions atomiques de CUDA n'est pas possible

dans ce cas car ces fonctions ne supportent pas les opérations en double précision. Cette étape de recherche de minimum est tout simplement mise en œuvre sur le CPU, et l'indice du minimum est ensuite communiqué au GPU (voir figure VI.3).

Dans l'étape 1 de l'algorithme du Simplexe, la première ligne du tableau du Simplexe est communiquée au CPU et l'indice *r* de la variable sortante est ensuite récupéré. Nous présentons maintenant le *Kernel* 1 qui calcule les éléments de la colonne des ratios $\theta$. Ensuite, la colonne des ratios $\theta$ est communiquée au CPU pour la recherche du minimum correspondant à l'indice de la variable sortante *k*.

**Kernel 1:** *Kernel* GPU pour le calcul de la colonne des ratios $\theta$ et la copie de *l'ancienne* $k^{ème}$ colonne dans la colonne *Columnk*.

```
global_void Kernel1( double *θ, double *Columnk, int k,
                     double TableauSimplexe [m+1][n+1])
{
    int idx = blockDim.x * blockIdx.x + threadIdx.x;
    double w = SimplexTableau[idx][k];
    θ[idx] = SimplexTableau[idx][1]/w;
                                /* Copie de la colonne k */
    Columnk[idx] = w;
}
```

Le *Kernel* 1 est aussi utilisé pour récupérer « *l'ancienne* $k^{ème}$ colonne » (voir la section VI.2 étape 3) nommée *Columnk* et la stocker dans la mémoire globale du GPU, ceci pour éviter les cas de conflit de mémoire lors de la mise à jour du tableau du Simplexe à l'étape 3.

A cette étape de l'algorithme, on récupère l'élément pivot, le $r^{ème}$ élément de *Columnk*, nommé *wp*, à l'aide de l'instruction *cudaMemcpy*() avec l'argument *cudaMemcpyDeviceToDevice*. L'élément *wp* est stocké dans la mémoire globale du GPU. On évite ainsi les instructions conditionnelles du type « *if* (*threadIdx.x* == 0) *wp* = *Columnk*[*r*]; » au début des tâches 1, 2 et 3 de l'étape 3 de l'algorithme du Simplexe.

### VI.4.3 Mise à jour de la base

Nous utilisons dans ce qui suit la notation standard CUDA où $x$ et $y$ désignent, respectivement, la colonne et la ligne.

L'étape 3 est entièrement réalisée sur le GPU (où chaque partie est réalisée par un *Kernel)*. Il serait possible d'utiliser uniquement un seul *Kernel* (voir *Kernel* 2) afin d'exécuter l'étape 3 qui comporte $(m + 1) \times (n + 1)$ opérations parallèles. La colonne $k$ et la ligne $r$ doivent être mises à jour de manière différente du reste du tableau du Simplexe. Cela conduit à l'utilisation, comme illustré par les parties grises du *Kernel* 2:

- d'instructions *if* suivies de synchronisations *syncthreads*(); qui forment une barrière entre les threads (voir la section V.2.1). D'ordinaire cette instruction n'est pas coûteuse, mais dans ce cas de figure, le nombre de threads parallèles à exécuter dans le *Kernel* 2 est important. La synchronisation engendre alors un coût non négligeable.

- d'instruction *if* suivie de chemins divergents, ce qui diminue le parallélisme dans le sens où dans certains *warps*, l'exécution est sérialisée (voir la section V.5.2).

Afin d'éviter les deux instructions précédentes, l'étape 3 de l'algorithme du Simplexe est réalisée au moyen de trois *Kernels* (*Kernel* 2', *Kernel* 3' et *Kernel* 4) où *Kernel* 3' est le *Kernel* de la mise à jour du tableau du Simplexe (le parallélisme est ainsi plus efficace).

Le calcul de la nouvelle ligne $r$ du tableau du Simplexe, la ligne pivot, est opéré par le *Kernel* 2' de la manière suivante :

Le thread $x$ du bloc $X$ traite l'élément *TableauSimplexe*[$r$][ $x + 32 \times X$ ] avec $x = 0, \ldots,31$ et $X = 0, \ldots, w - 1$. Le résultat est copié dans un vecteur nommé *Liner* qui est stocké dans la mémoire globale du GPU. Cet élément sera repris dans la mise en œuvre multi-GPU que nous verrons à la section VI.5.

**Kernel 2:** 1ère version du *Kernel* GPU afin d'effectuer l'étape 3 de l'algorithme du Simplexe.

```
global_void Kernel2(double wp, int r, int k,double TableauSimplexe[m+1][n+1] )
{
        int idx = blockDim.x * blockIdx.x + threadIdx.x ;
        int jdx = blockIdx.y * blockDim.y + threadIdx.y ;
        __shared__ double wk[6] ;
        __shared__ double wr[32] ;

          /*Mise à jour de la ligne r et stockage dans la mémoire partagée du bloc */

if (jdx==r)
        {
        TableauSimplexe[r][idx] =  TableauSimplexe[r][idx] / wp ;
        wr[threadIdx.x] = TableauSimplexe[r][idx] ;
        }
__syncthreads();

         /* Mise à jour de la colonne k et stokage dans la mémoire partagée du bloc */

if (idx==k)
        {
        wk[threadIdx.y] = TableauSimplexe[jdx][k] ;
        }
__syncthreads();
                                        /* Mise à jour de la base */
if (idx !=k)
        {
if (jdx ==r) return ;
        TableauSimplexe[jdx][idx] = TableauSimplexe[jdx][idx] − wk[threadIdx.y] *
        wr[threadIdx.x];
        }
        else
        {
        if (jdx ==r)
                {
                TableauSimplexe[jdx][idx] = −TableauSimplexe[jdx][idx] / wp ;
                return ;
                }
        TableauSimplexe[jdx][idx]= 1 / wp ;
        }
}
```

**Kernel 2':** *Kernel* GPU pour le calcul de la ligne pivot *r*.

```
global_void Kernel2'(double wp, int k, int r, double * Columnk, double * Liner,
                     double TableauSimplexe[m+1][n+1])
{
        int idx = blockDim.x * blockIdx.x + threadIdx.x;
        double wr = TableauSimplexe [r][idx] ;
        wr = wr / wp ;
        Liner[idx] = wr ;
        TableauSimplexe [r][idx] = wr ;
}
```

Pour éviter que le *Kernel* 3 ne modifie la ligne *r* du *TableauSimplexe*, l'élément *Columnk*[*r*] est mis à 0 à l'aide de l'instruction *CudaMemset*(). En effet, dans le *Kernel* 3 pour *jdx* = *r*, nous avons :

$$TableauSimplexe[r][idx] = TableauSimplexe[r][idx] - 0 \times wr \ ;$$

**Kernel 3:** *Kernel* GPU pour la mise à jour du tableau du Simplexe.

```
global_void Kernel3( int r, int k, double * Columnk, double * Liner,
                     double TableauSimplexe[m+1][n+1] )
{
        int idx = blockDim.x * blockIdx.x + threadIdx.x;
        int jdx = blockIdx.y * blockDim.y + threadIdx.y;
        double wr = Liner[idx];
        __shared__  wk[6] ;
   /* Chargement  d'une partie de la colonne Columnk dans la mémoire partagée du bloc */
if (threadIdx.y == 0 && threadIdx.x < 6)
        {
        wk[threadIdx.x] = Columnk[blockIdx.y * blockDim.y + threadIdx.x];
        }
__syncthreads();
                                   /* Mise à jour de la base */
TableauSimplexe[jdx][idx] = TableauSimplexe[jdx][idx] – wk[threadIdx.y]* wr;
}
```

Ensuite, le tableau du Simplexe est mis à jour par le *Kernel* 3. Pour chaque bloc de dimension 6 × 32, une colonne de 6 éléments de la *Columnk* est chargée dans la mémoire partagée du bloc. Les blocs de la grille calculent alors leur partie du tableau du Simplexe de façon indépendante (en parallèle) de telle sorte que le thread (*x*, *y*) du bloc (*X*, *Y*) met à jour l'élément *TableauSimplexe*[*y* + 6 × *Y*][*x* + 32 × *X*] avec *x* = 0, …, 31, *y* = 0, …, 5 et *X* = 0, …, *w* – 1, *Y* = 0, …, *h* – 1 (voir la figure VI.4).

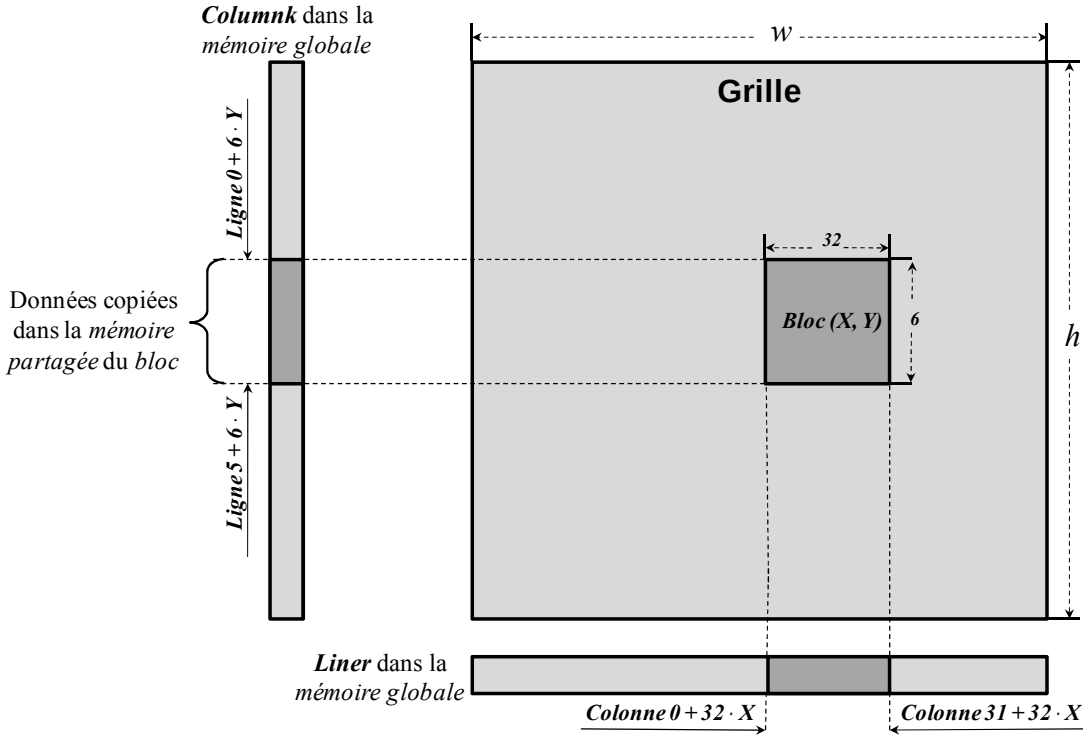



FIGURE VI.4 – Operations matricielles, gestion de mémoire dans le *Kernel* 3'.

Le choix du nombre threads par bloc, qui a une influence directe sur le temps d'exécution, est déterminé à l'aide de l'outil CUDA Occupancy Calculator de Mark Harris [HAR 10], comme cela a été fait au chapitre précédant, sous-section V.4.3. Ce nombre doit, garantir la meilleure occupation du GPU, masquer les latences de lecture/écriture en mémoire et garantir un nombre suffisant de registres par thread.

Ainsi, l'outil CUDA Occupancy Calculator propose, dans notre cas, une valeur égale à 192. On verra à la sou-section que les expérimentations ont confirmée cette valeur.

Il est encore possible d'optimiser le *Kernel* 3 en augmentant le rapport $\frac{q_{op}}{q_{le}} \times \frac{T_{op}}{T_{le}}$ où $T_{op}$ est le temps d'exécution d'une soustraction et d'une multiplication en double précision, $T_{le}$ le temps de lecture et d'écriture en mémoire globale du GPU, $q_{op}$ le nombre d'éléments mis à jour par un thread et $q_{le}$ le nombre de lectures et d'écritures en mémoire globale.

Dans le cas du *Kernel* 3, ce rapport est égal à $\frac{1}{4} \times \frac{T_{op}}{T_{le}}$.

En augmentant le nombre $q_{op}$ la taille de la grille de threads diminue (divisée par $q_{op}$) ce qui réduit le parallélisme. L'idée est de trouver la meilleure valeur de $q_{op}$ qui augmente la performance du *Kernel* 3. De manière expérimentale, $q_{op}$ est fixé à 4 ce qui correspond à une grille de dimension $\frac{h}{2} \times \frac{w}{2}$ blocs, le rapport précédent passe alors à $\frac{4}{12} \times \frac{To}{Tlem}$.

Ainsi le thread ($x$, $y$) du bloc ($X$, $Y$) met à jour les éléments :

- $TableauSimplexe[y + 6 \times Y][x + 32 \times X]$,
- $TableauSimplexe[y + 6 \times Y][x + 32 \times (X + \frac{w}{2})]$,
- $TableauSimplexe[y + 6 \times (Y + \frac{h}{2})][x + 32 \times X]$,
- $TableauSimplexe[y + 6 \times (Y + \frac{h}{2})][x + 32 \times (X + \frac{w}{2})]$,

avec $x = 0, \ldots, 31$, $y = 0, \ldots, 5$ et $X = 0, \ldots, \frac{w}{2} - 1$, $Y = 0, \ldots, \frac{h}{2} - 1$ (voir le *Kernel* 3').

**Kernel 3':** Version améliorée du *Kernel* GPU pour la mise à jour du tableau du Simplexe .

```
global_void Kernel3'( int r, int k, double * Columnk, double * Liner,
                      double TableauSimplexe[m+1][n+1] )
{
        int idx = blockDim.x * blockIdx.x + threadIdx.x;
        int jdx = blockIdx.y * blockDim.y + threadIdx.y;
        double wr1 = Liner[idx];
        double wr2 = Liner[idx + 32 × w/2 ];
        __shared__  wk1[6] ;
        __shared__  wk2[6] ;

   /* Chargement  d'une partie de la colonne Columnk dans la mémoire partagée du bloc */

if (threadIdx.y == 0 && threadIdx.x < 6)
        {
        wk1[threadIdx.x] = Columnk[blockIdx.y ∗ blockDim.y + threadIdx.x];
        wk2[threadIdx.x] = Columnk[blockIdx.y ∗ blockDim.y + threadIdx.x + 6 × h/2 ];
        }
__syncthreads();

                                /* Mise à jour de la base */

TableauSimplexe[jdx][idx] – = wk1[threadIdx.y]* wr1 ;
TableauSimplexe[jdx][idx + 32 × w/2 ] – = wk1[threadIdx.y]* wr2 ;
TableauSimplexe[jdx+ 6 × h/2 ][idx] – = wk2[threadIdx.y]* wr1 ;
TableauSimplexe[jdx+ 6 × h/2 ][idx + 32 × w/2 ] – = wk2[threadIdx.y]* wr2 ;
}
```

Finalement, la mise à jour de la $k^{ième}$ colonne du tableau du Simplexe est effectuée par le *Kernel* 4. L'élément *Colmnk*[*r*] est mis cette fois-ci à –1 à l'aide de l'instruction *CudaMemset*() pour éviter l'ajout de l'instruction « *if* (*idx* == *k*) » dans le *Kernel* 4.

**Kernel 4:** *Kernel* GPU pour la mise à jour de la colonne de la variable entrante *k*.

```
global_void Kernel4'(double * Columnk, double wp, double SimplexTableau[m+1][n+1])
{
int jdx = blockDim.x * blockIdx.x + threadIdx.x;
TableauSimplexe[jdx][k] = −Columnk[jdx]/wp;
}
```

## VI.5 Simplexe sur un système multi-GPU

Nous cherchons maintenant à mettre à contribution plusieurs cartes GPU disponibles dans un seul système. Nous présenterons ici une mise en œuvre multi-GPU de l'algorithme [LAL 11b].

Notons $I$ le nombre de cartes GPU (du même type) disponibles dans un système. Le tableau du Simplexe (*TableauSimplexe*) est décomposé en $I$ parties (voir Figure VI.5) de sorte que chaque partie est mise à jour par un GPU et ceci à chaque itération de l'algorithme du Simplexe.

La mise en œuvre proposée est alors basée sur l'exécution simultanée de $I$ *threads CPU* identiques de sorte que chaque GPU $i$ est géré par son propre *thread CPU*, $i = 1, ..., I$. Le $i^{ème}$ *thread CPU* est composé des 4 *Kernel*s requis par l'algorithme du Simplexe. Ces 4 *Kernel*s sont exécutés sur la $i^{ème}$ partie du tableau du Simplexe par le $i^{ème}$ GPU. Cette approche présente l'avantage de maintenir le contexte des *threads CPU* tout au long de l'application, c'est à dire que les *threads CPU* ne sont pas tués à la fin de chaque itération l'algorithme du Simplexe. Par conséquent, les communications sont réduites au minimum.

Nous avons choisi de diviser horizontalement le *TableauSimplexe* car cette décomposition permet le calcul de la colonne des ratios $\theta$ et la mise à jour de la colonne $k$, en parallèle entre tous les GPUs. La ligne pivot est, quand à elle, mise à jour par un seul GPU à chaque itération.

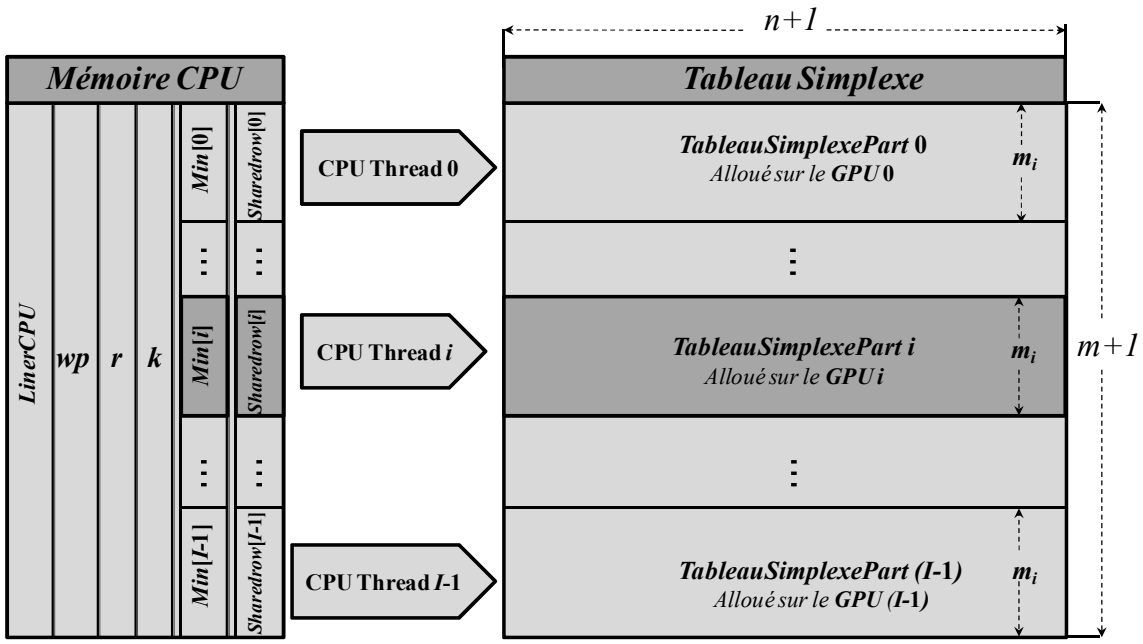



FIGURE VI.5 – Décomposition du *TableauSimplexe* et accès mémoire des threads CPU.

## VI.5.1 Initialisation

Le tableau du Simplexe de taille $(m + 1) \times (n + 1)$ qui est disponible sur le processeur, est divisé en $I$ parties, appelées *TableauSimplexePart*. Chaque *TableauSimplexePart* de taille $m_i \times (n + 1)$ où $m_i = \lceil (m+1)/I \rceil$, est chargée dans la mémoire globale d'un GPU. Cela nécessite des communications entre le CPU et les GPUs. Les opérations de pivotage seront par la suite réalisées par les GPUs. Par analogie avec la mise en œuvre CPU-GPU présentée à la sous-section IV.4, une grille (grid) de *threads GPU* est crée dans chaque GPU. La seule différence est que la dimension la grille est plus petite.

## VI.5.2 Les threads CPU

La procédure exécutée par un *thread CPU* lors d'une itération est décrite dans l'algorithme *Simplexe Thread CPU* présenté ci-dessous.

Algorithme **Simplexe Thread CPU** (*$i^{ème}$ thread CPU*):

```
                    /* Données partagées entre les Threads CPU */
Min[I], k, r, wp, LinerCPU[n + 1], Sharedrow[I][m_i] ,
                    /* Données locales d'un Thread CPU */
TableauSimplexePart[m_i][n + 1], Columnk[m_i], Liner[n + 1] ,
begin Procedure
                    /* Calcul de l'indice k de la variable entrante */
if i = 0 do
        GPU_to_CPU com(TableauSimplexePart [0], Sharedrow),
end if
Synchronize(),
Min[i] := Find_min(Sharedrow[i]),
Synchronize(),
k :=Find_min(Min),
                    /* Calcul de l'indice r de la variable sortante */
Kernel1(),
GPU_to_CPU_com(θ, Sharedrow[i]),
Min[i] := Find_min(Sharedrow[i]),
Synchronize(),
r :=Find_min(Min),
                    /* mise à jour de la base */
id := Find_GPUid_pivot(),
if i = id
        wp := Get_pivot_GPU_to_CPU(),
        Kernel2'(),
        GPU_to_CPU_com(Liner, LinerCPU),
        Set_pivot_line_to_0(),
end if
CPU_to_GPU_com(LinerCPU ,Liner),
Kernel3'(),
Kernel4(),
end Procedure.
```

On peut distinguer dans cette procédure deux types de données: les données partagées et les données locales.

- Les données partagées entre les GPUs:

  $k$, $r$ et $wp$ sont respectivement l'indice de la variable entrante, l'indice de la variable sortante et l'élément pivot. Le vecteur *Sharedrow*[$I$][$m_i$] est un vecteur de taille $I \times m_i$ qui est utilisé pour recevoir la première ligne de tableau du Simplexe et la colonne des ratios $\theta$ afin de calculer, respectivement, $k$ et $r$.

  *LinerCPU*[$n$ + 1] est utilisé pour recevoir la ligne $r$ de du tableau du Simplexe envoyé par le GPU qui contient cette ligne dans sa mémoire.

  Ces données sont stockées dans la mémoire vive du CPU (voir Figure VI.5).

- Les données locales:

  Stockées dans la mémoire globale de chaque GPU, ces données sont utilisées par les *Kernel*s GPU de l'algorithme *Simplexe Thread CPU*.

La fonction *Synchronise*() effectue une synchronisation globale de tous les threads CPU afin d'assurer la cohérence des données.

Les échanges de données entre GPUs se font via les deux fonctions suivantes:

- *GPU_to_CPU_com*(*source*, *destination*): chaque GPU écrit dans le CPU (*destination*) les données contenues dans *source*.

- *CPU_to_GPU_com*(*source*, *destination*): chaque GPU lit dans le CPU les données contenues dans *source* et les copie dans *destination* qui se trouve dans la mémoire globale du GPU.

### VI.5.3 Calcul de la variable entrante et la variable sortante

Comme indiqué à la section VI.4.2, la recherche de l'indice de la variable entrante ou sortante se fait sur un CPU. La fonction *Find_min*() est utilisée par le $i^{ème}$ *thread CPU* pour trouver l'indice du minimum local *Min*[$i$] dans *Sharedrow*[$i$]. Par la suite, l'indice du minimum global est calculé dans le vecteur *Min* qui contient les $I$ minimums locaux, puis est communiqué aux GPUs.

A l'étape 1 de l'algorithme du Simplexe, la première ligne du tableau du Simplexe est simplement communiquée au CPU, (vecteur *Sharedrow*), par le GPU d'indice 0. L'étape 2 est effectuée de manière analogue à la mise en œuvre CPU-GPU; chaque GPU calcule une partie de la colonne des ratios $\theta$. Cette dernière est communiquée au CPU, dans le vecteur *Sharedrow*, pour la recherche du minimum correspondant à l'indice de la variable sortante $k$. De plus, chaque GPU stocke dans sa mémoire globale sa partie de la colonne *Columnk*.

L'étape 3 de l'algorithme du Simplexe est réalisée de la manière suivante :

La fonction *Find_GPUid_pivot*() est utilisée pour trouver l'indice du GPU qui contient la ligne $r$ du tableau du Simplexe. Ce GPU exécute alors seul le *Kernel* 2' et transmet ensuite la ligne *Liner* et l'élément pivot $wp$ au CPU. Ce dernier transmet alors ces données à tous les GPUs.

Chaque GPU exécute de manière indépendante les *Kernel*s 3' et 4 sur leur partie respective du tableau du Simplexe.

Il est à noter que les *Kernel*s 1, 2', 3' et 4 ont cette fois comme entrée *TableauSimplexePart* [$i$] au lieu de *TableauSimplexe*.

## VI.6 Résultats expérimentaux

Nous présentons maintenant les résultats expérimentaux obtenus dans différents contextes : un CPU, un système CPU-GPU et un système avec un CPU et deux GPU. Pour ces différentes expérimentations, nous avons utilisé un processeur Intel Xeon E5640 2,66 GHz et deux GPUs NVIDIA Tesla C2050.

Nous avons utilisé CUDA 3.2 pour le code parallèle et *GCC* pour le code séquentiel. Les problèmes de programmation linéaire que nous avons considérés ont été engendrés aléatoirement ; $a_{ij}$, $b_i$, $c_j$, $i \in \{1, ..., m\}$ et $j \in \{1, ..., n\}$, sont des variables entières réparties uniformément sur l'intervalle [1, 1000]. Notons que la matrice $A$ est une matrice de type dense. Ces problèmes ont été mis en ligne sur le site de notre équipe de recherche, voir [LAA 12].

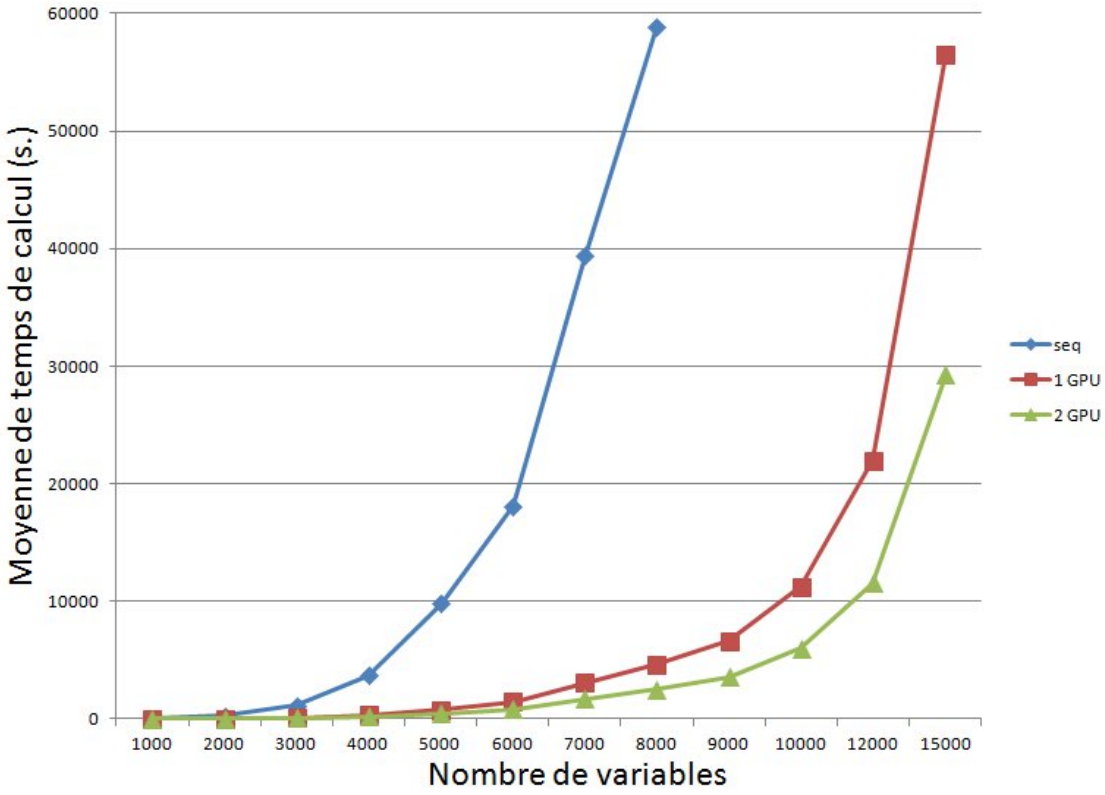



FIGURE VI.6 – Temps d'exécution de l'algorithme du Simplexe sur un CPU et différents systèmes avec un GPU et deux GPUs (Nvidia Tesla C2050).

Les calculs ont été effectués en double précision afin d'assurer une bonne précision de la solution.

Les temps de calcul sont donnés pour une moyenne de 10 instances et les accélérations (speedup) qui en résultent ont été calculées comme suit:

$$\text{Speedup 1} = \frac{\text{temps de calcul sur le processeur (s.)}}{\text{temps de calcul sur le système CPU avec un GPU (s.)}}$$

$$\text{Speedup 2} = \frac{\text{temps de calcul sur le système CPU avec un GPU (s.)}}{\text{temps de calcul sur le système CPU avec deux GPUs (s.)}}$$

Nous notons que les temps de calcul des méthodes séquentielles et parallèles correspondent à des versions identiques de l'algorithme du Simplexe.

Le *speedup*1 résulte de l'accélération obtenue en passant d'un CPU à un système CPU-GPU avec un seul GPU.

Le *speedup*2 est le rapport entre les temps obtenus avec un système ayant un CPU et un seul GPU et ceux obtenus sur un système avec deux GPUs.

La Figure VI.6 présente les temps de calculs obtenus pour différentes tailles de problèmes LP avec l'algorithme séquentiel et les algorithmes parallèles. Nous pouvons voir que les algorithmes parallèles (avec un et deux GPUs) sont plus rapides que l'algorithme séquentiel. Dans le cas séquentiel et pour des problèmes de taille de 8000 × 8000, nous notons que le temps de calcul dépasse la limite de temps de 16 heures que nous avons imposées.

Les résultats présentés au Tableau IV.2, montrent que l'accélération *speedup*1 augmente avec la taille des problèmes. Pour des problèmes de petite taille par exemple 1000 × 1000, le *speedup*1 est relativement faible (accélération moyenne = 2,93). Ceci est du au fait que la puissance réelle du GPU est très faiblement exploitée car le nombre de blocs créés dans ce cas n'est pas assez important. En augmentant la taille des problèmes considérés, une accélération moyenne stable de 12,7 est atteinte avec un seul GPU.

| *N* | *Speedup1* | *Speedup2* |
|---|---|---|
| 1000 × 1000 | 2,93 | 0,43 |
| 2000 × 2000 | 10,62 | 1,02 |
| 3000 × 3000 | 12,24 | 1,38 |
| 4000 × 4000 | 12,73 | 1,59 |
| 5000 × 5000 | 12,71 | 1,71 |
| 6000 × 6000 | 12,74 | 1,76 |
| 7000 × 7000 | 12,72 | 1,82 |
| 8000 × 8000 | 12,74 | 1,85 |
| 9000 × 9000 | / | 1,86 |
| 10000 × 10000 | / | 1,88 |
| 12000 × 12000 | / | 1,91 |
| 15000 × 15000 | / | 1,93 |

TABLE VI.2 – Accélérations moyennes : *speedup*1 (CPU-GPU/ CPU), *speedup*2 (CPU-2 GPUs/ CPU-GPU).

Nous pouvons voir aussi que la valeur de *speedup*2 augmente avec la taille des problèmes et atteint une accélération de 1,93 autrement dit une efficacité de 96.5% (calculée ainsi : $\frac{1.93}{2} \times 100\,\%$ ). Cela montre bien que l'utilisation de deux GPU conduit à une très petite perte d'efficacité pour des problèmes de grandes tailles. En conséquence, une accélération de 24,5

peut être atteinte en passant d'une mise en œuvre séquentielle à la mise en œuvre parallèle avec 2 GPUs.

Nous notons également que des instances de taille respectable, à savoir 19000 × 19000 sur le système avec un GPU et 27000 × 27000 sur le système avec deux GPU, ont pu être testées sans dépasser la capacité de la mémoire des cartes GPUs. Cela confirme l'intérêt du calcul sur GPU pour ce type de problèmes car l'utilisation de plusieurs GPUs permet de résoudre efficacement des problèmes de taille conséquente avec des temps de calcul raisonnable.

## VI.7 Conclusions et perspectives

Dans ce chapitre nous avons proposé des mises en œuvre parallèles en double précision de la méthode du Simplexe pour résoudre les problèmes de programmation linéaire sur divers systèmes comportant une ou plusieurs cartes GPUs.

Nous nous sommes d'abord intéressés au cas où un seul GPU est disponible. Ainsi, nous avons montré comment mettre à profit le parallélisme offert par cette nouvelle architecture massivement parallèle afin d'accélérer les différentes étapes de l'algorithme du Simplexe, ceci en identifiant les tâches de cet l'algorithme qui peuvent être parallélisées de manière performante.

Par la suite, nous avons proposé une mise en œuvre multi-GPU de l'algorithme du Simplexe. Nous avons montré comment partager le tableau du Simplexe entre les différent GPUs et comment diminuer ainsi les échanges entre les GPUs et le CPU.

Les résultats expérimentaux montrent que notre mise en œuvre CPU-GPU est efficace puisque nous avons obtenu des accélérations stables autour de 12,7 pour des problèmes de programmation linéaire de taille conséquente. Plus encore, le passage d'un seul GPU à deux GPUs permet de doubler l'accélération pour atteindre une valeur de 24.5. Notre approche a permis aussi de résoudre des problèmes de taille 15000 × 15000, sans dépasser la capacité de la mémoire du GPU. Ces résultats confirment l'intérêt du calcul GPU pour la programmation linéaire.

Les perspectives de ce travail concernent d'abord la mise en œuvre efficace de méthodes de recherche de la première solution de base. En effet cela permettra de diminuer le nombre d'itérations de l'algorithme du Simplexe. Il serait aussi intéressant d'optimiser le *Kernel* 3'. En effet, ce dernier est lancé à chaque itération pour exécuter une opération assez simple

(soustraction et multiplication). L'idée est de mettre à jour la base, c.-à-d. de lancer le *Kernel* 3', au bout d'un certain nombre d'itérations (par exemple 2 ou 3 itérations). Cela permettra d'avoir encore plus de taches parallèles à exécuter, tout en diminuant le nombre d'accès à la mémoire globale du tableau *TableauSimplexe*.

Dans nos futurs travaux, nous prévoyons de proposer une mise en œuvre de l'algorithme du Simplexe révisé.

# Chapitre VII

## Conclusions et Perspectives

Les travaux présentés dans ce mémoire appartiennent à deux domaines porteurs : l'optimisation combinatoire et le parallélisme. Nous nous sommes intéressés plus particulièrement à des problèmes de type sac à dos.

Nous avons proposé l'heuristique RCH pour le problème du sac à dos multiple qui consiste à résoudre récursivement chaque noyau des différents sacs à dos. Pour les $m - 1$ premiers noyaux, un problème de *Subset sum* (pour lequel le profit est égal au poids) est résolu par l'approche basée sur la programmation dynamique proposée par Elkihel [ELK 84] tandis que le dernier noyau (pour lequel les profits sont quelconques) est résolu en utilisant la programmation dynamique. Les résultats des expérimentations montrent que l'heuristique *RCH* aboutit généralement à de meilleurs *gaps* comparée à l'heuristique *MTHM* de Martello et Toth ainsi qu'au solveur CPLEX, et ceci dans de meilleurs temps de calcul.

Nous avons aussi proposé une mise en œuvre parallèle via CUDA de l'algorithme de Branch and Bound pour les problèmes de *sac à dos*, sur un système CPU-GPU. Ainsi nous avons montré comment mettre à profit ce type de carte afin d'accélérer les différentes étapes de l'algorithme de Branch and Bound. Nous nous sommes particulièrement concentrés sur la minimisation des effets des chemins divergents et la synchronisation des threads d'un même warp. Les résultats expérimentaux ont montré que des problèmes difficiles pouvaient être résolus dans des temps de calculs réduits. Nous avons obtenu des accélérations de l'ordre de 52 sur carte NVDIA C2050. Ce travail montre l'intérêt de l'utilisation des systèmes CPU/GPU pour résoudre des problèmes difficiles d'optimisation combinatoire. Nous pensons que des accélérations plus élevées peuvent être obtenues en testant des problèmes plus grands ou en mettant en œuvre nos algorithmes parallèles sur des clusters de $GPU_s$.

Les perspectives à ce travail consistent d'abord à incorporer l'étape de séparation et de calcul de bornes dans un seul kernel. Il en résultera une seule étape de lecture et écriture en mémoire globale. De plus, il serait intéressant de mettre en œuvre toute l'étape d'élagage entièrement sur GPU par l'incorporation d'algorithmes de tri tels que l'algorithme du Quicksort, ce qui permettrait d'abord de diminuer considérablement les communications entre CPU et GPU et ensuite de pouvoir tester la stratégie de parcours «meilleur d'abord» de la liste des nœuds. Nous prévoyons aussi de mettre en œuvre la méthode de réduction et fixation de variables à chaque étape de séparation de l'algorithme de Branch and Bound. Cela nécessitera de revoir toute la structure de l'algorithme. Nous comptons aussi proposer une version multi-GPU de l'algorithme de Branch and Bound mettant à contribution plusieurs cartes GPU disponibles sur une machine ou sur un cluster.

Nous avons aussi proposé une mise en œuvre parallèle en double précision de la méthode du Simplexe pour résoudre les problèmes de programmation linéaire sur un système mettant à contribution le CPU et des cartes GPU. Nous nous sommes d'abord intéressés au cas où un seul GPU était disponible. Nous avons montré comment mettre à profit le parallélisme offert par ce nouveau type d'architecture afin d'accélérer les différentes étapes de l'algorithme du Simplexe ceci en identifiant les tâches de cet algorithme qui peuvent être parallélisées de manière performante.

Par la suite, nous avons proposé une mise en œuvre multi-GPU de l'algorithme du Simplexe. Nous avons montré comment partager le tableau du Simplexe entres les différents GPUs et comment diminuer ainsi les échanges entre les GPUs et le CPU. Les résultats expérimentaux montrent que notre mise en œuvre parallèle est efficace puisque pour des problèmes de programmation linéaire de taille conséquente, nous avons obtenu des accélérations stables autour de 12,7. Plus encore, le passage d'un seul GPU à deux GPUs a permis de doubler l'accélération pour atteindre une valeur de 24,5. Notre approche a permis aussi de résoudre des problèmes de taille 15000 × 15000, sans dépasser la capacité de la mémoire du GPU.

Les perspectives de ce travail concernent d'abord la mise en œuvre d'heuristiques efficaces de recherche de la première solution de base. En effet cela permettrait de diminuer le nombre d'opérations de pivotage de l'algorithme du Simplexe. Dans un travail futur, nous prévoyons de proposer une mise en œuvre sur GPU de l'algorithme du Simplexe révisé qui est très utilisé en logistique.

Ces travaux ainsi que les travaux sur la mise en œuvre parallèle sur GPU de la méthode de programmation dynamique effectués au sein de l'équipe CDA du LAAS-CNRS permettent d'envisager la résolution efficace de problèmes difficiles en optimisation combinatoire au moyen de méthodes parallèles coopératives combinant des codes de Branch and Bound et de programmation dynamique sur GPU.

# Annexe A

## Carte Graphique Nvidia Tesla C2050

| | |
|---|---|
| **Architecture** : | Fermi |
| **Compute capability**: | 2.0 |
| **Nombre de Multiprocesseurs CUDA**: | 14 |
| **Nombre de cœurs CUDA par Multiprocesseurs**: | 32 |
| **Nombre de cœurs CUDA :** | 448 |
| **Fréquence des cœurs CUDA :** | 1.15GHz |
| **Performances de la virgule flottante en double précision :** | 515 Gigaflops |
| **Performances de la virgule flottante en simple précision :** | 1.03 TFLOPS |
| **Mémoire globale totale :** | 3 Go GDDR5 |
| **Vitesse de la mémoire :** | 1.5 GHz |
| **Bande passante mémoire** | 144 Go/sec |
| **Mémoire constante totale :** | 64 Ko |
| **Mémoire partagé par Multiprocesseur :** | 49152 octets |
| **Nombre de registres par Multiprocesseur :** | 32768 |
| **Nombre de threads par warp :** | 32 |
| **Nombre Maximum de threads par bloc :** | 1024 |
| **Dimension Maximale d'un bloc :** | (1024, 1024, 64) |
| **Dimension Maximale d'une grille :** | (65535, 65535, 1) |
| **Version de CUDA :** | 3.2 |

# Liste des publications

## Publications dans des revues scientifiques internationales


[LAL 12a] M. E. Lalami, M. Elkihel, D. El Baz, V. Boyer, *A Procedure-Based Heuristic for 0-1 Multiple Knapsack Problems*, *International Journal of Mathematics in Operational Research*, Vol.4, N°3, p. 214-224, Mars 2012.


## Publications dans des conférences scientifiques internationales


[LAL 11a] M. Lalami, V. Boyer, D. El Baz, "*Efficient Implementation of the Simplex Method on a CPU-GPU System*", in Proceedings of the Symposium IEEE IPDPSW 2011, Anchorage, USA, p.1999-2006, May 2011.

[LAL 11b] M. Lalami, D. El Baz, and V. Boyer, "*Multi GPU implementation of the Simplex algorithm*," 13th IEEE International Conference on High Performance Computing and Communications (HPCC 2011), Banff, Canada, p. 1994-2001, September 2011.

[BOU 12] A. Boukedjar, M. Lalami and D. El-Baz, Parallel Branch and Bound on a CPU-GPU System, *in Proceedings of 20th Euromicro International Conference on Parallel and Distributed Computing*, Munich, Germany, p. 392-398, February 2012.

[LAL 12b] M. Lalami and D. El Baz, "*GPU implementation of the branch and bound method for knapsack problems*". in Proceedings of the Symposium IEEE IPDPSW 2012, Shanghai, China, p. 1763-1771, 21-25 May 2012.


## Publications dans des conférences scientifiques nationales


[LALR 10] M. Lalami, D. El Baz, M. Elkihel, V. Boyer, Une heuristique pour le problème du sac à dos multiple en variables 0-1, *Congrès de la Société Française de Recherche Opérationnelle et d'Aide à la Décision*, ROADEF'2010, Toulouse, 24-26 Février 2010.

[LALR 11] M. Lalami, M. Elkihel, D. El Baz, V. Boyer, Heuristics for 0-1 multiple knapsack problems: comparison of methods, *Congrès de la Société Française de Recherche Opérationnelle et d'Aide à la Décision*, ROADEF'2011, Saint-Etienne, 2-4 Mars 2011.

# Références bibliographiques


[AHR 75] J.H. Ahrens, G. Finke. "*Merging and sorting applied to the zero-one knapsack problem"*. Operations Research, Vol. 23, pp.1099–1109. 1975.

[AMD 10] AMD, "*Compute Abstraction Layer CAL: programming guide 2 .0*",http://developer.amd.com/sdks/amdappsdk/assets/AMD_CAL_Programming_Guide_v2.0.pdf. 2010

[ARO 10] R. Arora, R. Tulshyan, and K. Deb. "*Parallelization of binary and real-coded genetic algorithms on GPU using CUDA"*. In IEEE Congress on Evolutionary Computation [DBL10], pages 1–8, 2010.

[BAD 06] E.S. Badr, M. Moussa, K. Papparrizos, N. Samaras, and A. Sifaleras. "*Some computational results on MPI parallel implementations of dense Simplex method"*. Transactions on Engineering, Computing and Technology, 17:228–231, December 2006.

[BAL 80] E. Balas, E. Zemel. "*An algorithm for large zero-one knapsack problems"*. Operations Research, Vol. 28, pp.1130–1154. 1980.

[BAT 68] K.E. Batcher, "*Sorting networks and their applications*", in Proc. AFIPS Spring Joint Comput. Conf., vol. 32,pp. 307–314. Apr. 1968.

[BEK 10] A. Bekrar, I. Kacem, C. Chu, C. Sadfi, "*An improved heuristic and an exact algorithm for the 2D strip and bin packing problem*", International Journal of Product Development 10 ,pp. 217-240, 2010.

[BEL 54] R. Bellman, "*Some applications of the theory of dynamic programming, a review*", Operations Research, vol. 2, pp. 275-288, 1954.

[BEL 57] R. Bellman, "*Dynamic Programming*", Princeton University Press, 1957.

[BEL 08a] T. Belgacem, M. Hifi. "*Sensitivity analysis of the binary knapsack problem: perturbation of a subset of items*". Discrete Optimization, 5(4):755-761, 2008.

[BEL 08b] T. Belgacem, M. Hifi." *Sensitivity analysis of the knapsack sharing problem: perturbation of the profit of an item*". International Transactions in Operational Research, 15(1):35-49, 2008

[BIE 10] J. Bieling, P. Peschlow, P. Martini, *"An efficient GPU implementation of the revised Simplex method,"* in Proc. of the 24th IEEE International Parallel and Distributed Processing Symposium and Workshops, (IPDPSW 2010), Atlanta, USA, April 2010.

[BIX 00] R.E. Bixby and A. Martin. *"Parallelizing the dual simplex method"*. INFORMS Journal on Computing, 12:45–56, 2000.

[BOL 03] J. Bolz, I. Farmer, E. Grinspun, and P. Schröder, "*Sparse matrix solvers on the GPU: Conjugate gradients and multigrid"*. ACM Trans. Graph., vol. 22, no. 3, pp. 917–924, Jul. 2003.

[BOU 12] A. Boukedjar, M. Lalami, and D. El Baz, "*Parallel branch and bound on a cpu-gpu system*", Proceedings of the 20th International Conference on Parallel and Distributed and network-based Processing (PDP 2012), pp. 392–398, Garching, Germany. 2012.

[BOY 09] V. Boyer, M. Elkihel, D. El Baz. "*Heuristics for the 0-1 multidimensional knapsack problem*". European Journal of Operational Research, Vol. 199, pp.658-664. 2009

[BOY 10] V. Boyer, D. El Baz, M. Elkihel, "*Solution of multidimensional knapsack problems via cooperation of dynamic programming and branch and bound*," European J. Industrial Engineering, Vol. 4, n° 4 : 434–449, 2010.

[BOY 11] V. Boyer, D. El Baz, M. Elkihel, "*Dense dynamic programming on multi GPU*," in Proc. of the 19th International Conference on Parallel Distributed and networked-based Processing, PDP 2011, Ayia Napa, Cyprus, 545–551, February 2011.

[BOY 12] V. Boyer, D. El Baz, and M. Elkihel, "*Solving knapsack problems on GPU*" Computers and Operations Research, vol. 39, pp. 42–47, 2012.

[BROOK] http://graphics.stanford.edu/projects/brookgpu/index.html.

[BUC 04] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, P. Hanrahan."*Brook for GPUs: Stream computing on graphics hardware*". ACM Transactions on Graphics 23, 777–786, 2004.

[BUS 06] B. Bustos, O. Deussen, S. Hiller, and D. Keim, "*A graphics hardware accelerated algorithm for nearest neighbor search"*, in Proc. 6th Int. Conf. Comput. Sci., vol. 3994, pp. 196–199, Lecture Notes in Computer Science. May 2006.

[CED 09] D. Cederman and P. Tsigas, "*A practical Quicksort algorithm for graphics processors"*, in Journal of Experimental Algorithmics (JEA), Vol 14 , July 2009

[CHA 11] I. Chakroun, A. Bendjoudi et N. Melab, "*Reducing Thread Divergence in GPU-based B&B Applied to the Flow-shop problem"*, 9th International Conference on Parallel Processing and Applied Mathematics (PPAM 2011) , Poland, September 11-14, 2011

[CHR 79] N. Christofides, A. Mingozzi, and P. Toth. "*Loading problems"*. In P. Toth and N. Christofides, editors, Combinatorial Optimization, 339--369, Wiley, 1979.

[CUDA] CUDA Zone, http://www.nvidia.com/content/cuda/cuda-toolkit.html.

[CUV 11] T. Cuvelier. "*Une introduction à CUDA*", http://tcuvelier.developpez.com/tutoriels/gpgpu/cuda/introduction/, 2011.

[CUBLAS] CUDA - CUBLAS Library 2.0, NVIDIA Corporation,

[CVE 91] Z. Cvetanovic, E. G. Freedman, and C. Nofsinger. "*Efficient decomposition and performance of parallel PDE, FFT, Monte-Carlo simulations, simplex, and sparse solvers"*. Journal of Supercomputing, 5:19–38, 1991.

[DAN 57] G.B. Dantzig, "*Discrete variable extremum problems"* Operations Research, vol. 5, pp. 266–277, 1957.

[DAN 63] G.B. Dantzig, "*Linear Programming and Extensions"*. Princeton University Press and the RAND Corporation, 1963.

[DAN 03] G.B. Dantzig, M. N. Thapa, "*Linear Programming 2: Theory and Extensions"*, Springer-Verlag, 2003.

[DEM 98] M.A.H. Dempster and R. T. Thompson. "*Parallelization and aggregation of nested Benders decomposition*". Annals of Operations Research, 81:163–187, 1998.

[DEN 96] Y. Denneulin , B. Le Cun, T. Mautor et J.F. Méhaut. "*Distributed branch and bound algorithms for large quadratic assignment problems*". 5th Computer Science Technical Section on Computer Science and Operations Research. 1996.

[DIF 76] W. Diffie and M. Hellman. "*New Directions In Cryptography*". IEEE Transactions on Information Theory, IT-22(6) : 644-654, November, 1976.

[DUD 87] K. Dudzinski and S. Walukiewicz. "*Exact methods for the knapsack problem and its generalizations"*. European Journal of Operational Research, 28:3-21, 1987

[ECK 10] J. Eckstein, I. Bodurglu, L. Polymenakos, and D. Goldfarb, "*Data-Parallel Implementations of Dense Simplex Methods on the Connection Machine CM-2*," ORSA Journal on Computing,vol. 7, 4 : 434–449, 2010.

[EGL 10] D. Egloff. High Performance Finite Difference PDE Solvers on GPUs. Technical Report, QuantAlea Gmbh 2010.

[EIL 71] S. Eilon, N. Christofides. *"The loading problem"*. Management Science, Vol. 17, pp.259–268. 1971

[ELB 05] D. El Baz, M. Elkihel. "*Load balancing methods and parallel dynamic programming algorithm using dominance technique applied to the 0-1 knapsack problem*". Journal of Parallel and Distributed Computing, Vol. 65, pp.74–84. 2005.

[ELK 84] M. Elkihel, "*Programmation dynamique et rotations de contraintes pour les problèmes d'optimisation entière*", Thèse de Doctorat, Université des Sciences et Techniques de Lille (France), 1984.

[ELK 02] M. Elkihel, G. Authié and F. Viader, "*An efficient hybrid dynamic algorithm to solve 0-1 knapsack problems*", International Symposium on Combinatorial Optimization (CO'2002), 2002, Paris. France.

[FAT 04] K. Fatahalian, J. Sugerman, and P. Hanrahan, "*Understanding the efficiency of GPU algorithms for matrix-matrix multiplication*", in Proc. Graph. Hardware 2004, pp. 133–138. Aug. 2004.

[FAY 75] D. Fayard et G. Plateau, "*Resolution of the 0-1 knapsack problem : Comparison of methods*", Mathematical Programming, vol. 8, pp. 272-307, 1975.

[FAY 82] D. Fayard and G. Plateau. "*An algorithm for the solution of the 0-1 knapsack problem*". Computing, 28:269, 1982

[FRE 94] A. Freville and G. Plateau, "*An efficient preprocessing procedure for the multidimensional 0-1 knapsack problem*", Discrete Applied Mathematics, vol. 49, pages 189-212, 1994.

[FRE 04] A. Freville, "*The multidimensional 0-1 knapsack problem : an overview"*, European Journal of Operational Research, vol. 155, pages 1-21. 2004.

[FOK 07] K.L. Fok, T.T. Wong, and M.L. Wong. "*Evolutionary computing on consumer graphics hardware*". IEEE Intelligent Systems, 22:69–78, 2007.

[GAL 05] N. Galoppo, N.K. Govindaraju, M. Henson, and D. Manocha, "*LU-GPU: Efficient algorithms for solving dense linear systems on graphics hardware*", in Proc. ACM/IEEE Conf. Supercomput. , p. 3, Nov. 2005.

[GAR 72] R.S. Garfinkel, D.L. Nemhauser, "*Integer Programming*", Wiley-Interscience, 1972.

[GEN 94] B. Gendron et T.G. Crainic. "*Parallel branch-and-bound algorithms: Survey and synthesis"*. Operation Research, 42 : 1042–1066, 1994.

[GIL 65] P. C. Gilmore and R. E. Gomory, "*Multistage Cutting Stock Problems of Two and More Dimensions*", Operations Research, vol. 13, pages 94-120, 1965.

[GOU 02] J.P. Goux, K.M. Anstreicher, N.W. Brixius, and J. Linderoth. "*Solving large quadratic assignment problems on computational grids"*. Math. Prog, 91(3): 563–588. 2002.

[GOV 04] N.K. Govindaraju, B. Lloyd, W. Wang, M. Lin, and D. Manocha, "*Fast computation of database operations using graphics processors*" in Proc. 2004 ACM SIGMOD Int. Conf. Manage. Data, pp. 215–226. Jun. 2004.

[GOV 05a] N. K. Govindaraju, M. Henson, M.C. Lin, and D. Manocha, "*Interactive visibility ordering of geometric primitives in complex environments*", in Proc. 2005 Symp. Interact. 3D Graph. Games, pp. 49–56, Apr. 2005.

[GOV 05b] N.K. Govindaraju and D. Manocha, "*Efficient relational database management using graphics processors*", in Proc. ACM SIGMOD Workshop Data Manage. New Hardware, pp. 29–34, Jun. 2005.

[GOV 08] N. K. Govindaraju, B. Lloyd, Y. Dotsenko, B. Smith, and J. Manferdelli. *High Performance Discrete Fourier Transforms on Graphics Processors*. In the 2008 ACM/IEEE Conference on Supercomputing, 2008.

[GUE 99] C. Guéret et C. Prins. "*A new lower bound for the open-shop problem*". Annals of Operations Research, 92 : 165-183, 1999.

[HAL 10] J. Hall. "*Towards a practical parallelisation of the simplex method*". Computational Management Science, vol. 7, issue 2, pages 139-170, 2010.

[HAN 96] S. Hanafi, A. Freville and A. El Abdellaoui, "*Meta-Heuristics : Theory and Application*", Chapter Comparison of heuristics for the 0-1 multidimensional knapsack problem, pages 446-465, Kluwer Academic, 1996.

[HAR 07] M. Harris, "*Optimizing parallel reduction in CUDA*," NVIDIA Developer Technology report, 2007.

[HAR 10] M. Harris, http://developer.download.nvidia.com/compute/cuda/CUDA_Occupancy calculator.xls, 2010

[HIF 05] M. Hifi, H. M'Halla, S. Sadfi: "*An exact algorithm for the knapsack sharing problem*". Computers & OR 32: 1311-1324 .2005.

[HIF 07] M. Hifi, M. Michrafy."*Reduction strategies and exact algorithms for the disjunctively knapsack problem*". Computers and Operations Research, 34 (0) : 2657-2673, 2007.

[HIF 08] M. Hifi, H Mhallah, S. Sadfi. "*Adaptive algorithms for the knapsack problem*". European Journal of Industrial Engineering, 2 (2) : 134-152, 2008.

[HIF 09] M. Hifi. "*Approximate algorithms for the container loading problem*", Operations Research, Vol. 9 (6), pp.747–774. 2009.

[HIF 10] M. Hifi, I. Kacem, S. Nègre, L. Wu. "*A Linear Programming Approach for the Three-Dimensional Bin-Packing Problems*", Electronic Notes in Discrete Mathematics 2010, 36 : 1, 993-1000, 2010.

[HOA 62] C.A.R. Hoare, "Quicksort". Computer Journal, 5:10-15, 1962.

[HOR 74] E. Horowitz et S. Sahni, "*Computing partitions with applications to the knapsack problem*", Journal of ACM, vol. 21, pp. 277-292, 1974.

[HOR 05] D. Horn, "*Stream reduction operations for GPGPU applications*", in GPU Gems 2,M. Pharr, Ed. Reading, MA: Addison-Wesley, pp. 573–589, Mar. 2005.

[HUN 78] M.S. Hung, J.C. Fisk. "*An algorithm for the 0-1 multiple knapsack problem*", Naval Research Logistics Quarterly, Vol 24, pp.571–579. 1978.

[IBM 95] IBM. "*Optimization Subroutine Library - Guide and Reference"*, Release 2.1. – IBM Corporation, 1995.

[ING 73] G. P. Ingargiola et J. F. Korsh, "*Reduction algorithm for zero-one single knapsack problems*", Management Science, vol. 20, pp. 460-463, 1973.

[JAN 08] A. Janiak, Wladyslaw and Maciej Lichtenstein. "*Tabu search on GPU*". J. UCS, 14 (14) : 2416–2426, 2008.

[JAN 88] V.K. Janakiram,Agrawal (D.P.) et Mehrotra (R.). "*A randomized parallel branch-and-bound algorithm*". In International Conference on Parallel Processing, pp. 69–75. 1988.

[KAC 08] I. Kacem, C. Chu. "*Efficient branch-and-bound algorithm for minimizing the weighted sum of completion times on a single machine with one availability* constraint". International Journal of Production Economics, pp. 138-150, 2008.

[KEL 04] H. Kellerer, D. Pisinger, U. Pferschy. "*Knapsack problems*". Springer. 2004.

[KER 08] A. Kerr, D. Campbell and M. Richards, "*GPU VSIPL: High-performance VSIPL implementation for GPUs*," in HPEC'08: High Performance Embedded Computing Workshop, Lexington, MA, USA, 2008.

[KOL 67] P. J. Kolesar, "*A branch and bound algorithm for the knapsack problem*", Management Science, vol. 13, pages 723-735, 1967.

[KRU 03] J. Krüger and R. Westermann, "*Linear algebra operators for GPU implementation of numerical algorithms*", ACM Trans. Graph., vol. 22, no. 3, pp. 908–916, Jul. 2003.

[KRU 05] J. Krüger, P. Kipfer, P. Kondratieva, and R. Westermann, "*A particle system for interactive visualization of 3D flows*," IEEE Trans. Vis. Comput. Graphics, vol. 11, pp. 744–756, Nov.–Dec. 2005.

[KUM 84] V. Kumar et Kanal (L.N.). "*Parallel branch-and-bound formulations for and/or tree search*". IEEE Trans.Pattern Anal. and Mach. Intel., pp. 768–778. 1984.

[KUM 88] V. Kumar, K. Ramesh et V. Nageshwara Rao. "*Parallel best-first search of state-space graphs: a summary of results*". in 7th National Conference on Artificial Intelligence, pp. 122–127, 1988.

[LAA 09] "*Multiple knapsack problem benchmarks"*. 'http://www.laas.fr/laas09/CDA-EN/45-31328-Multiple-knapsack-problem.php.'

[LAA 12] "*Knapsack problems benchmark"* http://www.laas.fr/laas09/CDA/23-31300-Knapsack-problems.php.

[LAB 03] M. Labbé, G. Laporte, and S. Martello. "*Upper bounds and algorithms for the maximum cardinality bin packing problem*". European Journal of Operational Research, Vol. 149, pp.490-498. 2003.

[LAL 11a] M. Lalami, V. Boyer, D. El Baz, "*Efficient Implementation of the Simplex Method on a CPU-GPU System*", in Proceedings of the Symposium IEEE IPDPSW 2011, Anchorage USA, p.1999-2006, May 2011.

[LAL 11b] M. Lalami, D. El Baz, and V. Boyer, "*Multi GPU implementation of the Simplex algorithm*," 13th IEEE International Conference on High Performance Computing and Communications (HPCC 2011), Banff, Canada, p. 1994-2001, September 2011.

[LAL 12a] M. E. Lalami, M. Elkihel, D. El Baz, V. Boyer, " *A Procedure-based heuristic for 0-1 multiple knapsack problems"*, International Journal of Mathematics in Operational Research, Vol.4, N°3, p. 214-224, Mars 2012.

[LAL 12b] M. Lalami and D. El Baz, "*GPU implementation of the branch and bound method for knapsack problems*". in Proceedings of the Symposium IEEE IPDPSW 2012, Shanghai China, p. 1763-1771 , 21-25 May 2012.

[LAN 60] A. H. Land et A. G. Doig, "*An automatic method for solving discrete programming problems*", Econometrica, vol. 28, pp. 497-520, 1960.

[LAPAC] LAPACK Library, *http://www.culatools.com/*

[LEA 08] D.P. O'Leary, J.H. Jung, "*Implementing an interior point method for linear programs on a CPU-GPU system*," Electronic Transactions on Numerical Analysis, 28 : 879–899, May 2008.

[LEE 07] J. Lee, N. Kim, H. Lee et al. "*Efficient liver segmentation using a level-set method with optimal detection of the initial liver boundary from level-set speed images*". Comput Methods Programs Biomed 88, pp : 26–38, 2007.

[LEF 03] A.E. Lefohn, "*A streaming narrow-band algorithm: Interactive computation and visualization of level-set surfaces*", Master's thesis, Univ. of Utah, Dec. 2003.

[LEN 95] M. Lentini, A. Reinoza, A. Teruel, and A. Guillen. SIMPAR: "*A parallel sparse simplex*". Computational and Applied Mathematics, 14 (1) : 49–58, 1995.

[LUO 06] Z. Luo and Hongzhi Liu. "*Cellular genetic algorithms and local search for 3-sat problem on graphic hardware*". In Evolutionary Computation, 2006. CEC 2006. IEEE Congress, pages 2988 –2992, 2006.

[LUO 10] T.V. Luong, N. Melab, and E.G. Talbi, "*Large neighborhood local search optimization on graphics processing units*," Proceedings of the Symposium IEEE IPDPSW 2010, Atlanta, USA, 2010.

[LUO 11] T.V. Luong, N. Melab, and E.G. Talbi. "*GPU-based Approaches for Multiobjective Local Search Algorithms. A Case Study: the Flowshop Scheduling Problem*". 11th European Conference on Evolutionary Computation in Combinatorial Optimization, EVOCOP 2011, pages 155–166, volume 6622 of Lecture Notes in Computer Science, Springer, 2011.

[LUO 91] J. Luo, G.L. Reijns, F. Bruggeman, and G.R. Lindfield. "*A survey of parallel algorithms for linear programming*". In E. F. Deprettere and A. J. van der Veen, editors, Algorithms and Parallel VLSI Architectures, volume B, pages 485–490. Elsevier, 1991.

[MAR 77] S. Martello and P. Toth. "*An upper bound for the zero-one knapsack problem and a branch and bound algorithm*". European Journal of Operational Research, 1:169-175, 1977

[MAR 80] S. Martello, P. Toth. "*Solution of the zero-one multiple knapsack problem*". European Journal of Operational Research, Vol. 4, pp. 276–283. 1980.

[MAR 81] S. Martello, P. Toth. "*A Bound and Bound algorithm for the zero-one multiple knapsack problem*". Discrete Applied Mathematics, 3 : 275--288, 1981.

[MAR 82] S. Martello et P. Toth, "*A mixed algorithm for the Subset-Sum Problem*", EURO V. TIMS, Lausanne , France, 1982 .

[MAR 84] S. Martello, P. Toth, "*A mixture of dynamic programming and branch and bound for the subset-sum problem*", Management Science 30, 765--771, 1984.

[MAR 88] S. Martello, P. Toth. "*A new algorithm for the 0-1 knapsack problems*". Management Science, Vol. 34, pp. 633–644. 1988.

[MAR 90] S. Martello, P. Toth. "*Knapsack problems: algorithms and computer implementations*". J Wiley. 1990.

[MAR 97] S. Martello, P. Toth. "*Upper bounds and algorithms for hard 0-1 knapsack problems*". Operations Research, Vol. 45, pp. 768–778, 1997.

[MEL 05] N. Melab. *"Contributions à la résolution de problèmes d'optimisation combinatoire sur grilles de calcul"*. Habilitation à Diriger des Recherches de l'Université de Lille1, Novembre, 2005.

[MIL 93] D.L. Miller. et J.F. Pekny."*The role of performance metrics for parallel mathematical programming algorithms*". INFORMS Journal on Computing, vol. 5, pp. 26–28, 1993.

[MOR 97] S.S. Morgan, "*A comparison of Simplex method algorithms*", Master's thesis, University of Florida, January 1997.

[MUL 79] H. Müller-Merbach. "*Improved upper bound for the zero-one knapsack problem*". European Journal of Operational Research, 2 : pp. 121-213, 1979.

[NEE 77] A. Neebe and D. G. Dannenbring. "*Algorithm for a specialized segregated storage problem*", Technical report 77-5, University of north Carolina, 1977.

[NETLIB] http://www.netlib.org/

[NUK 12] A. Nukada, Y. Maruyama and S. Matsuoka ,"*High performance 3-D FFT using multiple CUDA GPUs*". in Proceedings of the 5th Annual Workshop on General Purpose Processing with Graphics Processing Units, pp. 57-63 , 2012.

[NVI 12] NVIDIA, Cuda 4.1 "*programming guide"*, http://developer.download.nvidia.com/compute/cuda/2 0/docs/NVIDIA CUDA Programming Guide 4.1.pdf 2012.

[OPE 12a] OpenCL for AMD, "*AMD Accelerated Parallel Processing OpenCL*",http://developer.amd.com/sdks/amdappsdk/assets/amd_accelerated_parallel_processing_opencl_programming_guide.pdf, 2012.

[OPE 12b] OpenCL for AMD, "*OpenCL Programming Guide for the CUDA Architecture*", http://developer.download.nvidia.com/compute/DevZone/docs/html/OpenCL/doc/OpenCL_Programming_Guide.pdf, 2012.

[OWE 07] J.D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A. E. Lefohn, and T. Purcell, "*A survey of general-purpose computation on graphics hardware",* Comput. Graph. Forum, vol. 26, no. 1, pp. 80–113, 2007.

[PFE 76] C.E. Pfefferkorn and J. A. Tomlin. "*Design of a linear programming system for the ILLIAC IV*". Technical Report SOL 76-8, Systems Optimization Laboratory, Stanford University, 1976.

[PIS 95] D. Pisinger. "*An expanding-core algorithm for the exact 0-1 knapsack problems*". European Journal of Operational Research, Vol. 87, pp.175–187. 1995.

[PIS 97] D. Pisinger. "*A minimal algorithm for the 0-1 knapsack problem"*. Operations Research,45:758-767, 1997.

[PIS 99] D. Pisinger. "*An exact algorithm for large multiple knapsack problems*". European Journal of Operational Research 114 (3) : 528-541. 1999.

[PLA 85] G. Plateau, M. Elkihel. "*A hybrid method for the 0-1 knapsack problem*". Methods of Operations Research, Vol. 49, pp. 277–293. 1985.

[QIU 09] D. Qiu, S. May, and A. Nüchter. "*GPU-accelerated Nearest Neighbor Search for 3D registration*". in *ICVS 2009*: Proceedings of the 7th International Conference on Computer Vision Systems, October, 2009.

[QIZ 05] Y. Qizhi, C. Chen, and Z. Pan. "*Parallel genetic algorithms on programmable graphics hardware*". in Lecture Notes in Computer Science 3612, page 1051. Springer, 2005.

[REB 10] J. Rebacz, E. Oruklu, J. Saniie, "*Exploring scalability of FIR filter realizations on Graphics Processing Units*", IEEE International Conference on Electro/Information Technology (EIT), pp. 1-5, 20-22 May, 2010.

[ROU 87] C. Roucairol. "*Du Séquentiel au parallèle: la recherche arborescente et son application à la programmation quadratique en variables 0-1*". Thèse d'Etat, Université Paris VI, France, 1987.

[SAT 09] N. Satish, M. Harris and M. Garland. "*Designing efficient sorting algorithms for manycore GPUs*", in IEEE Parallel & Distributed Processing IPDPSW 2009, Roma, Italy, pp. 1–10, May 2009.

[SHU 95] W. Shu. "*Parallel implementation of a sparse simplex algorithm on MIMD distributed memory computers*". Journal of Parallel and Distributed Computing, 31 (1) : 25–40, 1995.

[SMI 05] A. Smirnov and T. Chiueh. "*An Implementation of a FIR Filter on a GPU*". Tech. rep., Experimental Computer Systems Lab, Stony Brook University, 2005.

[SPA 09] D.G. Spampinato, A.C. Elster, "*Linear optimization on modern GPUs*"; in Proc. of the 23rd IEEE International Parallel and Distributed Processing Symposium, (IPDPSW 2009), Roma, Italy, May 2009.

[SUM 05] T. SUMANAWEERA, D. LIU."*Medical image reconstruction with the FFT*". In GPU Gems 2, Pharr M., Addison Wesley, ch. 48, pp. 765–784, March 2005.

[SUX 08] Y. Su, Z. Xu, X. Jiang and J. Pickering. "*Discrete wavelet transform on consumer-level graphics processing unit*". In Proceedings of Computing and Engineering Annual Researchers' Conference 2008: CEARC'08. Huddersfield, pp. 40-47, 2008.

[TOT 80] P. Toth, "*Dynamic programming algorithm for the zero-one knapsack problem*", Computing, vol. 25, pp. 29-45, 1980.

[TSU 09] S. Tsutsui and N. Fujimoto. "*Solving quadratic assignment problems by genetic algorithms with GPU computation: a case study*". In Proceedings of the 11th Annual Conference Companion on Genetic and Evolutionary Computation Conference: Late Breaking Papers, GECCO '09, pages 2523–2530, New York, NY, USA, ACM. 2009.

[VIA 98] F. Viader, "*Méthodes de programmation dynamique et de recherche arborescente pour l'optimisation combinatoire : utilisation conjointe des deux approches et parallélisation d'algorithmes*", Thèse de Doctorat, LAAS-CNRS Toulouse (France), 1998.

[WIR 98] G. Wirsching. "*Personal communication*". 1998.

[WOL 98] L.A. Wolsey. "*Integer Programming*". J. Wiley, 1998

[WON 05] M. L. Wong, T. T. Wong, and K. L. Fok. "*Parallel evolutionary algorithms on graphics processing unit*", in Congress on Evolutionary Computation, pages 2286–2293. IEEE, 2005.

[WON 06] M. L. Wong and T. T. Wong. "*Parallel hybrid genetic algorithms on consumer-level graphics hardware*", in Evolutionary Computation. CEC 2006. IEEE Congress on, pages 2973–2980, 2006.

[WON 07] T. T. Wong, C. S. Leung, P. A. Heng and J. Q. Wang. "*Discrete Wavelet Transform on Consumer-Level Graphics Hardware*", IEEE Transaction on Multimedia, Vol. 9, No. 3, pp. 668-673, 2007.

[YAM 01] T. Yamada, S. Kataoka. "*Heuristic and exact algorithms for the disjunctively constrained knapsack problem*". EURO 2001, Rotterdam, Netherlands; July 9-11, 2001.

[YAR 06] G. Yarmish, "*The Simplex method applied to wavelet decomposition*" in Proc. of the International Conference on Applied Mathematics, Dallas, USA, 226–228, November 2006.

[ZAH 07] Y. Zaho, "*Lattice Boltzmann based PDE solver on the GPU*", Visual Comput., vol 24, pp. 323–333, 2007.

[ZEN 89] S.A. Zenios. "*Parallel numerical optimization: current status and annotated bibliography*". ORSA Journal on Computing, 1 (1) : 20–43, Winter 1989.

[ZHU 09] W. Zhu. "*A study of parallel evolution strategy: pattern search on a GPU computing platform*", in Proceedings of the first ACM/SIGEVO Summit on Genetic and Evolutionary Computation, GEC '09, pages 765–772, New York, NY, USA, 2009.

## Résumé

Les problèmes d'optimisation combinatoire sont souvent des problèmes très difficiles dont la résolution par des méthodes exactes peut s'avérer très longue ou peu réaliste. L'utilisation de méthodes heuristiques permet d'obtenir des solutions de bonne qualité en un temps de résolution raisonnable. Les heuristiques sont aussi très utiles pour le développement de méthodes exactes fondées sur des techniques d'évaluation et de séparation.
Nous nous sommes intéressés dans un premier temps à proposer une méthode heuristique pour le problème du sac à dos multiple MKP. L'approche proposée est comparée à l'heuristique MTHM et au solveur CPLEX.
Dans un deuxième temps nous présentons la mise en œuvre parallèle d'une méthode exacte de résolution de problèmes d'optimisation combinatoire de type sac à dos sur architecture GPU.
La mise en œuvre CPU-GPU de la méthode de Branch and Bound pour la résolution de problèmes de sac à dos a montré une accélération de 51 sur une carte graphique Nvidia Tesla C2050.
Nous présentons aussi une mise en œuvre CPU-GPU de la méthode du Simplexe pour la résolution de problèmes de programmation linéaire. Cette dernière offre une accélération de 12.7 sur une carte graphique Nvidia Tesla C2050.
Enfin, nous proposons une mise en œuvre multi-GPU de l'algorithme du Simplexe, mettant à contribution plusieurs cartes graphiques présentes dans une même machine (2 cartes Nvidia Tesla C2050 dans notre cas). Outre l'accélération obtenue par rapport à la mise en œuvre séquentielle de la méthode du Simplexe, une efficacité de 96.5 % est obtenue, en passant d'une carte à deux cartes graphiques.

## Abstract

Combinatorial optimization problems are difficult problems whose solution by exact methods can be time consuming or not realistic. The use of heuristics permits one to obtain good quality solutions in a reasonable time. Heuristics are also very useful for the development of exact methods based on branch and bound techniques.
The first part of this thesis concerns the Multiple Knapsack Problem (MKP). We propose here a heuristic called RCH which yields a good solution for the MKP problem. This approach is compared to the MTHM heuristic and CPLEX solver.
The second part of this thesis concerns parallel implementation of an exact method for solving combinatorial optimization problems like knapsack problems on GPU architecture. The parallel implementation of the Branch and Bound method via CUDA for knapsack problems is proposed. Experimental results show a speedup of 51 for difficult problems using a Nvidia Tesla C2050 (448 cores).
A CPU-GPU implementation of the simplex method for solving linear programming problems is also proposed. This implementation offers a speedup around 12.7 on a Tesla C2050 board.
Finally, we propose a multi-GPU implementation of the simplex algorithm via CUDA. An efficiency of 96.5% is obtained when passing from one GPU to two GPUs.