République Algérienne Démocratique et Populaire

Ministère de l'enseignement Supérieur

Et de la recherche Scientifique

Université de -Jijel- Mohamed Sadik Ben Yahia

Faculté des sciences Exactes et informatique

Département d'Informatique

Mémoire

De fin d'étude pour l'obtention du diplôme

Master de Recherche en Informatique

Option : *Système d'information et aide à la décision*

Thème :

Implémentation parallèle basée GPU d'un algorithme de colonie de fourmis appliqué au problème de coloration de graphe

Réalisé par :

Touati Narimane

Encadré par :

Mr : Kerroum Ali

Promotion : 2017

Remerciments

En préambule à ce mémoire, je remercie Dieu le tout puissant et miséricordieux, qui m'a donné la force et la patience d'accomplir ce Modeste travail.

Ensuite, je tiens à remercier mon encadreur Kerroum Ali pour la confiance qu'il m'a accordé en acceptant d'encadrer ce travail, pour ses multiples conseils et son attention permanente sur l'évolution de mon travail. Merci aussi pour toutes les relectures, suggestions et commentaires, qui m'ont permis d'améliorer la qualité de cet mémoire.

Aussi, je joins ces remerciements également aux membres du jury, pour avoir accepté de juger ce mémoire.

Enfin, je remercie tous les enseignants qui ont été à la base de mon formation.

DEDICACE

Après cette réussite que fait la joie à tous qui m'aime. Je dédie ce modeste travail :

A mon père & ma mère, vous êtes pour moi une source de vie car sans vos sacrifices, votre tendresse et votre affectation je ne pourrais arriver jusqu'au bout. Je me réjouis de cet amour filial. Que dieu vous garde afin que votre regard puisse suivre ma destinée.

A ma sœur Leila et à mon frère Hamza qui ont été toujours présent pour moi.

A mes amis et collègues surtout Rami Guellil qui m'a beaucoup aider.

A tous ceux qui sont chères, proches de mon cœur, et à tous ceux qui m'aiment et qui aurait voulu partager ma joie…

Narimane

Table de matières

Liste des figures

Liste des tableaux

Liste des acronymes

GPU (Graphic Processing Unit) Processeur Graphique.

CPU (Central processing Unit) Unité Centrale de Traitement.

GPGPU (General Purpose computing on Graphics Processing Units) Calcul Générique sur un Processeur Graphique.

OpenCL (Open Computing Language) Langage de Programmation Ouvert.

ALU (Arithmetic and Logic Unit) Unité Arithmétique et Logique.

CUDA (Compute Unified Device Architecture) .

API (Application Programming Language) Interface de Programmation Applicative.

ALS-COL (Ant Local Search for Coloring problem) recherche locale de fourmis pour le problème de coloration des sommets d'un graphe.

Introduction Générale

Les algorithmes de colonies de fourmis, communément appelées ACO, pour Ant Colony Optimisation, forment une classe de méta-heuristique introduite par Marco Dorigo en 1990 pour la résolution des problèmes d'optimisation combinatoires difficiles. Ces algorithmes s'inspirent du comportement collectif des fourmis réelles recherchant le chemin optimal entre leur nid et la source de nourriture.

Les algorithmes de colonies de fourmi été appliqués à un grand nombre de problèmes d'optimisation difficiles tels que le problème de voyageur de commerce (PVC) et le problème de coloration des sommets d'un graphe (PCSG). Cependant, le temps de calcul de ces algorithmes est sérieusement compromis lors que l'instance du problème a une dimension élevée. Afin de réduire le temps de calcul de la solution, une implémentation parallèle devient attractive.

Poussées par une industrie du jeu vidéo de plus en plus exigeante, les cartes graphiques modernes, communément appelées GPUs, pour Graphics Processing Units, sont devenues de véritable plateformes de calcul intensif. Partant de ce constat, des sociétés comme NVIDIA ou AMD, se sont mises à développer des architectures, permettant le développement et l'exécution de codes généraux sur GPU.

L'attrait des processeurs graphiques réside dans deux points importants. Tout d'abord, les GPUs nous offrent une architecture massivement parallèle. Ainsi, si nous trouvions un moyen d'exploiter efficacement les GPUs, nous pourrions accélérer significativement une large gamme de codes de calculs. Deuxièmement, l'intérêt des cartes graphiques réside dans leur faible coût, par rapport au prix d'un microprocesseur classique. C'est pourquoi, ces dernières années, un grand nombre de travaux ont émergé sur le sujet de la programmation sur GPU.

Dans le cadre de ce travail, nous allons étudier et évaluer la possibilité d'utilisé les GPU pour faire une implémentation parallèle d'un algorithme de colonie de fourmis appliqué au problème de coloration de graphe.

Dans le premier chapitre nous allons présenter le GPU ainsi qu'un de ses frameworks de programmation «OpenCl».

Le deuxième chapitre sera consacré à la présentation du problème de colorations des sommets du graphe et a l'explication de l'algorithme de fourmis «ALS-COL» qui est destiné à sa résolution.

Le troisième chapitre est dédié à la présentation de la stratégie suivi pour réaliser l'implémentation parallèle de l'algorithme «ALS-COL».

Le dernier chapitre sera consacré à l'expérimentation des deux implémentations séquentielle et parallèle et à la comparaison des performances de ces deux dernières.

CHAPITRE 1:

Programmation parallèle sur GPU

1. Introduction

Les unités de traitement graphique ou GPU (Graphic Processing Unit) sont des composants matériels dédiés aux traitements graphiques mais pouvant également être utiliser en collaboration avec les CPUs (Central Processing Unit) pour réaliser des calculs généraux [1], ce qui est connu sous le nom de GPGPU : General purpose computing On Graphics Processing Units.

Ces unités sont dotées d'une architecture massivement parallèle qui leurs permet de réaliser plusieurs traitements en même temps en appliquant la même opération sur plusieurs données [2]. De plus ces composants sont caractérisés par une grande puissance de calcul qui les a rendus très attrayants pour les programmeurs.

Plusieurs frameworks de programmation parallèle ont été proposé afin de permettre aux programmeurs d'utiliser ses composants dans l'accélération des calculs et le développement d'applications parallèles.

Dans ce chapitre nous allons citer les différences entre un CPU et un GPU, parler de ses frameworks de programmation les plus célèbres, présenter le framework de programmation OpenCl et à la fin donner un exemple d'un programme parallèle écris en OpenCl C.

2. Différence entre CPU et GPU

Malgré que les GPUs puissent réaliser les calculs généraux des CPUs, ses composants restent différents. Comme il est montré dans la figure 1.1, un CPU regroupe un petit nombre d'unités de calculs appelées cores (cœurs) : le CPU intel Xeon Phi contient 72 cores [3]. Par contre un GPU intègre plusieurs centaines de cores : la carte graphique nvidia Geforce gtx 1080 possède 3584 cores [4].

Chaque core d'un CPU est capable d'effectuer son propre flot d'instructions, sur ses propres données, indépendamment des autres. Les cores des GPUs doivent par contre effectuer chacun un flot d'instructions identiques, sur des données différentes.

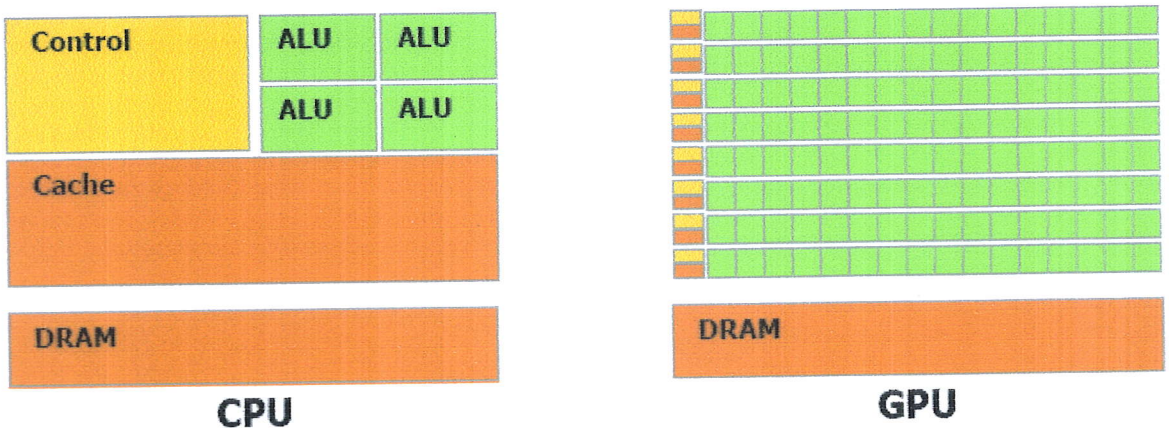

Figure 1.1 : architecture d'un CPU et architecture d'un GPU [5]

De plus les cores des CPUs et des GPUs sont aussi différents car les cores d'un CPU ne peut gérer qu'un ou deux threads par contre celui d'un GPU peut lancer de 4 à 10 threads à la fois [6].

Cette différence en nombre de cores et de threads rend les GPUs plus rapide que les CPUs dans l'exécution de leurs tâches [7], et leur puissance de calcul n'est pas restée stagnée à un stade mais elle a continué à évoluer de manière exponentielle comparativement aux CPUs. La figure 1.2 Montre l'évolution des performances des CPUs et des GPUs.

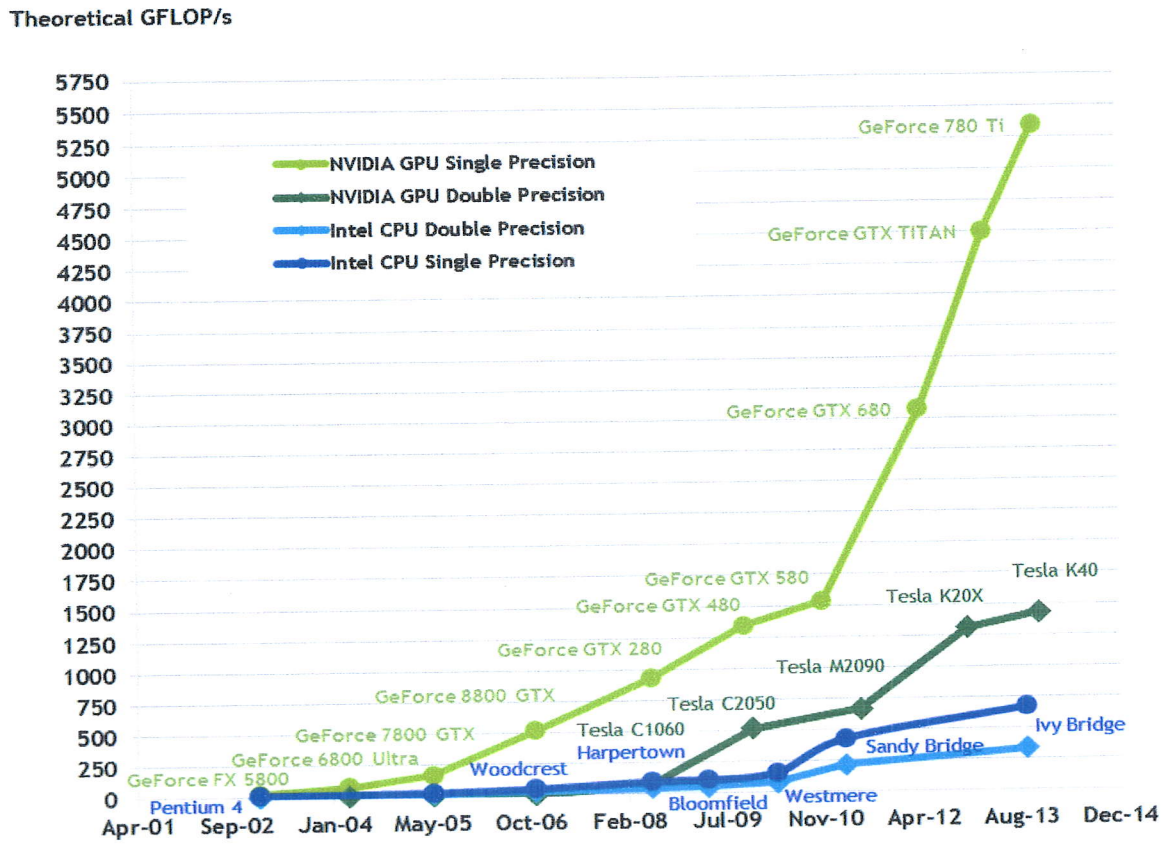

Figure 1.2 : evolution des performances en GFlops des CPU et GPU [8]

3. Frameworks de programmation parallèle sur GPU

Les GPUs modernes ne sont pas que de simples unités effectuant des tâches graphiques mais également des processeurs programmables et très parallèles réalisant les calculs généraux des CPUs [9]. Plusieurs frameworks de programmation parallèle ont été proposé, permettant l'utilisation des GPUs pour les calculs généraux et le développement d'applications parallèles. Parmi les frameworks les plus connus, on peut citer CUDA (Compute Unified Device Architecture) et OpenCl (Open Computing Language).

Le framework CUDA a été introduit par NVIDIA en 2007, Il permet aux développeurs de logiciels d'utiliser un GPU (seulement les GPUs provenant de NVIDIA) pour un traitement général [10] afin de résoudre de nombreux problèmes complexes de manière plus efficace que sur

un CPU. Avec des millions de GPU compatibles avec CUDA vendus à ce jour, les développeurs de logiciels, les scientifiques et les chercheurs trouvent de vastes utilisations pour le calcul de GPU avec CUDA [11].

OpenCl est un framework de programmation qui sert à écrire des programmes qui s'exécutent sur des systèmes parallèles et hétérogènes comprenant par exemple à la fois un CPU multi-cœurs et un GPU [12]. OpenCl a été initialement conçu par Apple et puis confié à Khronos Group pour qu'il devient après un standard ouvert, sa première version a été lancé en avril 2010 [13].

OpenCl contrairement à CUDA peut marcher sur une variété de GPUs et même dans des systèmes contenant des composants (CPU et GPU) qui proviennent de différents vendeurs [14].

4. Le framework OpenCl

4.1. Architecture d'OpenCl

L'architecture d'OpenCl est définie en quatre modèles qui aident à bien comprendre le fonctionnement d'une application OpenCl [15]. Ces quatre modèles peuvent être résumés en :

- **Modèle de platforme** : qui fournit une description du système hétérogène.
- **Modèle d'exécution** : qui montre comment l'exécution se passe dans le système hétérogène.
- **Modèle de mémoire** : qui décrit les différentes régions de mémoire d'OpenCl et les interactions entre elles.
- **Modèle de programmation** : ce modèle est utilisé par les programmeurs pour concevoir leurs algorithmes et implémenter des applications [16].

4.1.1. Modèle de plateforme

Le modèle de plateforme d'OpenCl se compose d'un host (ou hôte : représente le CPU) connecté à un ou plusieurs devices (ou périphérique : peut-être un CPU ou un GPU ou tout autre matériel accélérant et supportant OpenCl). Un device est constitué de plusieurs unités de calcul (compute unit), chacune comprend un ou plusieurs éléments de traitement (processing element).

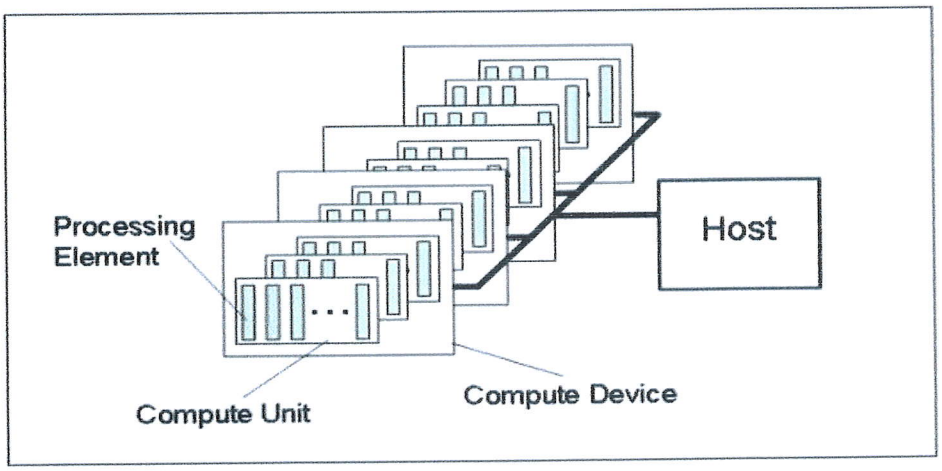

Figure 1.3 : modèle de plateforme d'OpenCl [17]

4.1.2. Modèle d'exécution

Une application écrite en OpenCL comporte deux parties :

- la première partie, appelée host code ou host program, destinée à s'exécuter sur le host. Elle sera chargée de récupérer les informations sur l'architecture matérielle disponible, de configurer, d'ordonnancer les tâches à exécuter et de récupérer les résultats produits. Cette partie est écrite en langage de haut niveau (C, C++, Java, etc.) et comporte des appels à des fonctions spécifiques OpenCL.
- la seconde partie, appelée kernel code, contenant un ensemble de fonctions appelées kernels écrites en langage OpenCL C et destinées à s'exécuter en parallèle sur un ou plusieurs devices. Ces kernels sont compilés à la volée par le host code pour générer un code binaire spécifique au device cible.

Un kernel est une fonction qui est utilisé pour réaliser un parallélisme de données ou de tâches, son exécution est lancée sur le host code en plusieurs instances appelés work-items. Lorsque l'exécution d'un kernel est lancée, un espace d'indexe à N dimension est définit appelé NDRange, N peut être 1, 2 ou 3 (comme il est montré dans la figure 1.4), chaque élément de cet espace représente un work-item distingué des autres par un ID global.

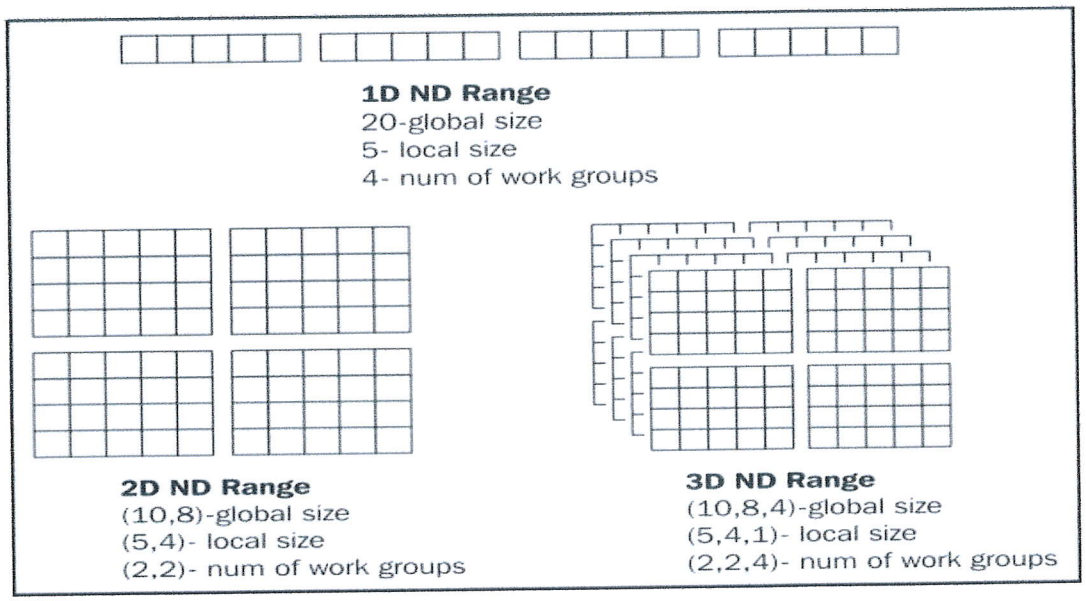

Figure 1.4 :NDRange d'OpenCl [18]

La taille d'une dimension x dans un NDRange est appellée «gloabal size(x) », x prend des valeurs de 0 à 2.

Les work-items sont organisés dans des groupes nommés "work-groups" et sont identifiés dans ses groupes par des IDs locaux.

Un work-group possède un ID du work-groupe unique et la taille d'une de ses dimensions (x) est appellée « local size(x) ».

Un work-item est déterminé de deux manières (voir la figure 1.5): la première par l'ID global et la deuxième par la combinaison de l'ID local et l'ID du work-group.

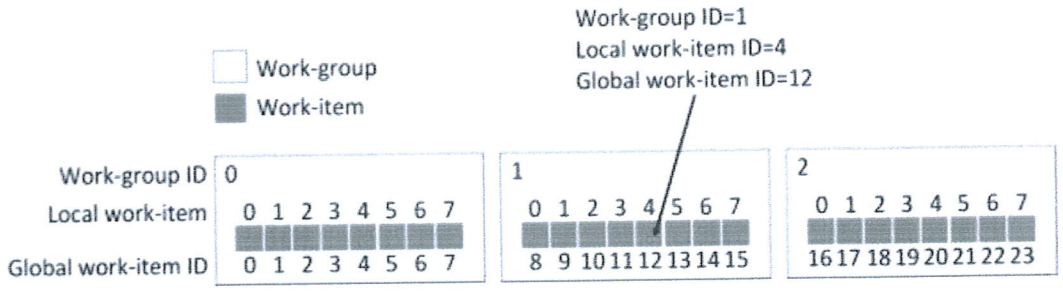

Figure 1.5: ID du work-group et du work-item [19]

Les IDs des work-items et des work-groups sont des identifiants à N valeurs (N est égale aux nombres de dimensions du NDRange).

Les valeurs des IDs globaux (locaux) sont comprises entre zéro (ou la valeur de l'offset) et le nombre de work-items dans la dimension du NDRange (dans la dimension du work-group qui correspond) moins 1.

Par contre les valeurs de l'ID du work-group sont entre zéro (ou la valeur de l'offset) et le nombre de work-groups dans la dimension du NDRange moins 1.

Dans un NDRange à 2 dimensions (voir la figure 1.6), l'identifiant global d'un work-item (gx, gy) est calculé comme suit :

(gx, gy) = (wx * Sx + sx, wy * Sy + sy).

Avec :

(wx, wy) : l'identifiant du work-group.

(sx, sy) : l'identifiant local du work-item.

(Sx, Sy) : la taille du work-group (local size).

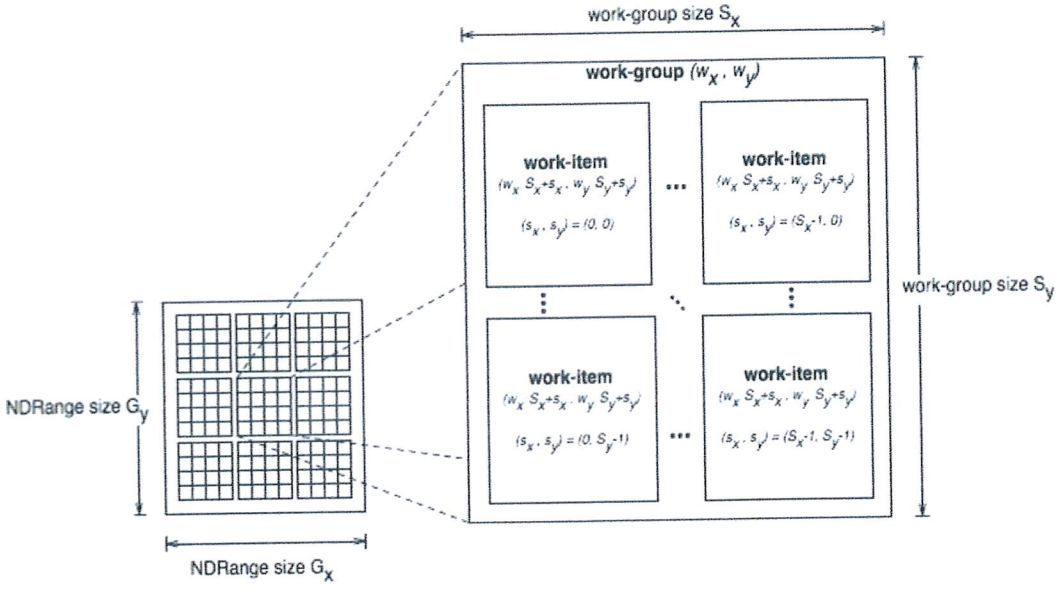

Figure1.6 : exemple d'un NDRange à deux dimensions [17]

4.1.3. Modèle de mémoire

Le modèle de mémoire d'OpenCl comprend les cinq types de mémoire suivante (voir la figure 1.6) :

- **La mémoire globale (Global Memory)**: cette mémoire est partagée par tous les work-groups : chaque work-item peut y accéder.
- **La mémoire constante (Constant Memory)**: elle constitue une partie de la mémoire globale qui reste constante tout au long de l'exécution du kernel. Les work-items n'ont qu'un accès en lecture dans cette région.
- **La mémoire locale (Local Memory)**: cette mémoire est partagée par tout un work-group : chaque work-item du même work-groupe peut y accéder.
- **La mémoire privée (Private Memory)**: chaque work-item dispose d'une mémoire privée qui est la plus rapide d'accès.
- **La mémoire hôte (Host Memory):** cette mémoire n'est visible que pour l'hôte et les objets mémoires sont transmis de l'hôte vers les devices par des appels d'API [20].

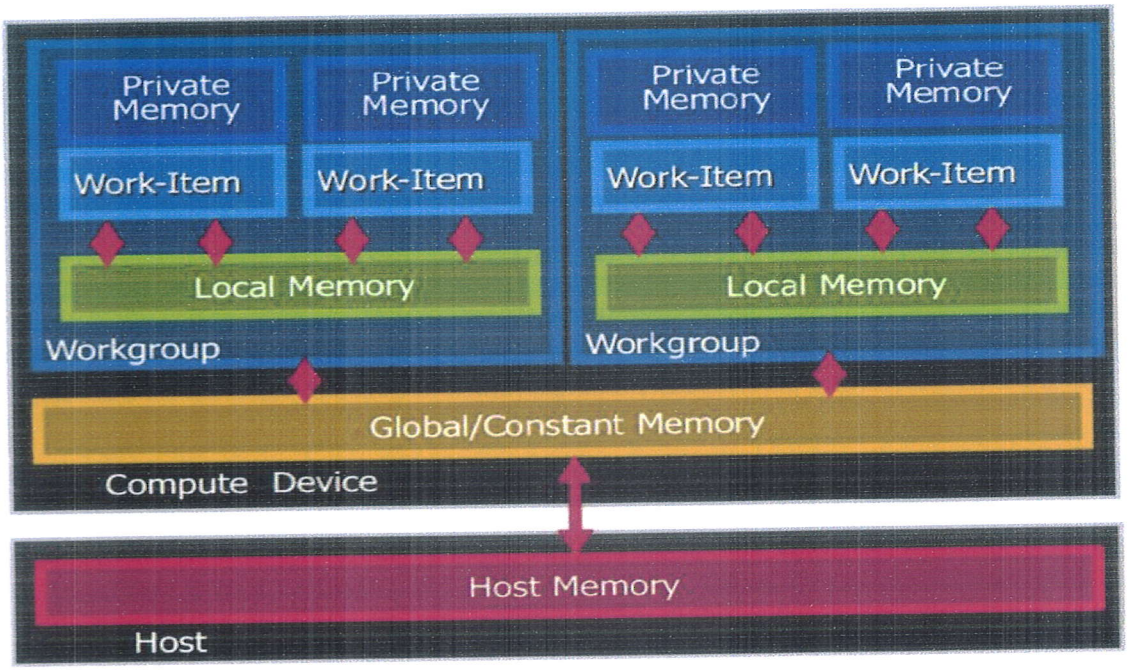

Figure 1.7 : modèle de mémoire d'OpenCl [21]

4.1.4. Modèle de programmation

OpenCl a été définit avec deux modèles de programmation parallèles différents qui sont : Le modèle de programmation parallèle de donnée et le modèle de programmation parallèle de tâche.

Dans le modèle parallèle de données plusieurs instances d'un même kernel sont exécutées simultanément et chaque instance opère sur des données différentes, par contre dans le modèle parallèle de tâches plusieurs kernels qui sont différents sont traités en parallèle [22].

4.2. Programmation avec OpenCl

4.2.1. Les objets d'OpenCl

- **Plateform object:**
  Un objet plateforme permet à une application de partager les ressources et d'exécuter les kernels sur les devices au sein de la plateforme. Il est référencé par un objet de type cl_Platform_id et il est initialisé par la fonction **cl_int clGetPlatformIDs (…).**

- **Device object:**
  Un objet device est utilisé pour exécuter un kernel. Il est référencé par un objet de type cl_device_id et il est initialisé par la fonction **cl_int clGetDeviceIDs (…).**

- **Context object:**
  Cet objet représente l'environnement dans lequel les work-items s'exécutent. Il est utilisé par l'environnement d'exécution d'OpenCL.
  Un contexte est associe à un ou plusieurs devices [23]. Il est référencé par l'objet cl_context et Il est initialisé par la fonction **cl_context clCreateContext (…).**

- **Commande queue object:**
  L'objet queue de commande est une fille utilisé par l'hôte pour enfiler les commandes à exécuter sur les devices [23]. Il est référencé par un objet de type cl_command_queue et il est initialisé par la fonction **cl_command_queue clCreateCommandQueue (…).** Plusieurs queues peuvent être crées (les commandes sont alors indépendantes et non synchronisés).

- **Buffer object:**

  Un objet buffer en OpenCl est utilisé pour réserver de la mémoire (mémoire global) sur un device afin de sauvegarder les données de l'application [22]. Il est référencé par un objet de type cl_mem et il est initialisé par la fonction **cl_mem clCreateBuffer (…).**

- **Program object:**

  Un objet programme représente le programme OpenCl [22]. Il est référencé par un objet de type cl_program et il est initialisé l'une des deux fonctions suivante :

  - **cl_program clCreateProgramWithSource (…).**
  - **cl_program clCreateProgramWithBinary (…).**

- **Kernel object:**

  Cet objet encapsule un kernel avec ses arguments dont les valeurs seront utilisées lors de son exécution [22]. Il est référencé par un objet de type cl_kernel et il est initialisé par la fonction **cl_mem clCreatekernel (…).**

4.2.2. Les étapes à suivre pour écrire un programme hôte OpenCl

L'élaboration d'un programme hôte en OpenCl nécessite aux programmeurs de suivre des étapes qui l'aide à réaliser son code, vue que le programme hôte est initié à base de plusieurs appels d'API.

Ces étapes sont résumées dans ce qui suit [24]:

1. Sélectionner une plateforme.
2. Sélectionner un device.
3. Créer un contexte.
4. Créer une file d'attente de commandes.
5. Créer des buffers.
6. Ecrire les données de l'hôte dans les buffers.
7. Créer et compiler le programme.
8. Créer un kernel.
9. Placer les arguments du kernel.

10. Configurer la structure des work-items.
11. Lancer l'exécution du kernel.
12. Lire les données reçus sur l'hôte.
13. Libérer les ressources d'OpenCl.

4.2.3. Exemple d'une application OpenCL : Multiplication de deux matrices

L'exemple en dessous représente un programme OpenCl parallèle qui calcule le produit de deux matrices A et B. Ce programme est constitué de deux parties qui sont :

1- Code du kernel :

Le code du kernel est sauvegardé dans un fichier (.cl) : "multiplication_matrice.cl".

Programme1.1 : multiplication de matrices – code du kernel

```
__kernel void multiplication_matrice(__global float * C ,__global float * A , __global float * B ,
                                     int CA, int LB){
int x = get_global_id(0);
int y = get_global_id(1);
float valeur = 0.0;
for(int i=0; i<CA; i++){
  int elementA = A[y * CA + i];
  int elementB = B[i * LB  + x];
  valeur = valeur + (elementA * elementB);
}
C[y * CA + x] = valeur;
}
```

Ce kernel (multiplication de matrice) est chargé de calculer le produit entre une ligne de la matrice A (identifié par le ID global du work-item dans la dimension 1) et une colonne de la

matrice B (identifié par le ID global du work-item dans la dimension 0) et affecte le résultat à une case de la matrice C.

2- code de l'hôte:

Dans cette partie l'exécution du kernel est invoquée et le résultat du produit des deux matrices est récupéré à la fin.

L'écriture du code de l'hôte est réalisée en suivant des étapes qui sont mentionnées dans des commentaires (voir le programme 1.2).

Programme1.2 : multiplication de matrices – code de l'hôte

```
#include <stdio.h>
#include <stdlib.h>
#include <CL/cl.h>
#include <time.h>
#include <random>
#define MAX_SOURCE_SIZE (0x100000)
int main()
{
//********* Déclarations
float * A ,B, C;
int LA , CA ;
int LB , CB;
unsigned int size_A, size_B, size_C ;
cl_mem buffer_A, buffer_B, buffer_C;
cl_platform_id ID_platform = NULL;
cl_uint num_platforms ;
cl_device_id ID_device ;
cl_uint num_devices ;
cl_context contexte ;
cl_int erreur ;
```

```
cl_command_queue command_queue;
cl_program programme ;
FILE * fichier_cl ;
cl_kernel kernel ;

//********** Lecture des dimensions des deux matrices A et B
printf("\n\n\n") ;
printf("%s\n\n","Saisissez les dimensions des matrices A et B : ") ;
printf(" %s","le nombre de ligne de la matrice A : ");
scanf("%i", &LA);
printf("\n");
printf(" %s","le nombre de colonne de la matrice A : ");
scanf("%i", &CA);
printf(" %s\n\n","le nombre de ligne de la matrice B : ") ;
printf(" %s\n\n","Remarque : le nombre de ligne de B doit être égale aux nombre de colonnes de
A");
scanf("%i", &LB) ;
printf("\n") ;
printf(" %s","le nombre de colonne de la matrice B : ") ;
scanf("%i", &CB) ;
printf("\n") ;
//********* Allouer et initialiser les matrices A , B et C
srand(time(NULL)) ;
size_A = LA * CA;
A = (float *)malloc(sizeof(float)*size_A) ;
for (int i=0 ; i <size_A ; i++){
    A[i] = rand()/(float)RAND_MAX;
}
size_B = LB * CB;
B = (float *)malloc(sizeof(float)* size_B) ;
```

```
for (int i=0 ; i <size_B ; i++){
  B[i] = rand()/(float)RAND_MAX;
}
size_C = LB * CA;
C = (float *)malloc(sizeof(float)* size_C);
//******* Etape 1 : Séléctionner une plateforme
erreur = clGetPlatformIDs (1, &ID_platform, &num_platforms);
if (erreur != CL_SUCCESS){
    printf ("%s", "Erreur : 0 platforme trouvée !");
    exit(EXIT_FAILURE);
}

//******* Etape 2 : Sélectionner un Device
erreur = clGetDeviceIDs(ID_platform,CL_DEVICE_TYPE_GPU,1, ID_DEVICE,
&num_devices) ;
if (erreur != CL_SUCCESS){
    printf ("%s", "Erreur : 0 device trouvée !");
    exit(EXIT_FAILURE);
}
//****** Etape 3 : Créer un contexte
contexte = clCreateContext(NULL, 1, &ID_device, NULL,NULL, &erreur);
if (erreur != CL_SUCCESS){
  printf ("%s", "Erreur : Echec lors de la creation du contexte   !");
  exit(EXIT_FAILURE);
}
//***** Etape 4 : Créer une Fille D'attente De Commande
command_queue = clCreateCommandQueue (contexte, ID_device, 0,&erreur);
if (erreur != CL_SUCCESS){
    printf ("%s", "Erreur : Echec lors de la creation de la fille d'attente de commandes !") ;
```

```
  exit(EXIT_FAILURE);
}
//***** Etape 5 et 6 : Créer Les Buffer et placer Les Données De l'hôte dans ces derniers
buffer_A = clCreateBuffer(contexte, CL_MEM_READ_WRITE|
CL_MEM_COPY_HOST_PTR ,size_A * sizeof(float),A,&erreur);
buffer_B = clCreateBuffer(contexte, CL_MEM_READ_WRITE |
CL_MEM_COPY_HOST_PTR, size_B * sizeof(float),B,&erreur);
buffer_C = clCreateBuffer(contexte, CL_MEM_READ_WRITE ,size_B * sizeof(float), NULL,
&erreur) ;
if(!buffer_A || ! buffer_B || ! buffer_C){
   printf("Echec de la création des objets mémoires !");
   exit(EXIT_FAILURE);
}
//***** Etape 7 : Créer Et Compiler Le Programme
fichier_cl = fopen ("multiplication_matrice.cl", "r") ;
if ( !fichier_cl ) {
    printf ("%s", "Echec dans l'ouverture du fichier !") ;
    exit(EXIT_FAILURE);
}
char * source_str = (char*)malloc(MAX_SOURCE_SIZE);
size_tsource_size = fread(source_str,1, MAX_SOURCE_SIZE, fichier_cl);
fclose (fichier_cl);
programme = clCreateProgramWithSource(contexte, 1, (const char **)&source_str,(constsize_t
*)&source_size, &erreur);
if (erreur != NULL){
   printf ("%s", "Erreur : Echec de la creation du programme !");
   exit(EXIT_FAILURE);
}
//***** Etape 8 : créer un kernel
```

```
kernel = clCreateKernel(programme,"multiplication_matrice",  &erreur);
if(erreur != CL_SUCCESS){
   printf ("Echec de la Création du kernel !") ;
   exit(EXIT_FAILURE);
}

//***** Etape 9 : placer les arguments du kernel
erreur = clSetKernelArg(kernel,0, sizeof(cl_mem),(void*) &buffer_C );
erreur = clSetKernelArg (kernel, 1,sizeof(cl_mem),(void *)&buffer_A);
erreur = clSetKernelArg (kernel, 2, sizeof(cl_mem),(void *)&buffer_B);
erreur = clSetKernelArg (kernel, 3, sizeof(int), &nbr_colonne_A);
erreur = clSetKernelArg (kernel, 4, sizeof(int), &nbr_ligne_B) ;
if(erreur != CL_SUCCESS){
  printf ("Echec de l'initialisation des arguments du kernel!");
  exit(EXIT_FAILURE);
}
erreur = clSetKernelArg (kernel, 1,sizeof(cl_mem),(void *) &buffer_A);
erreur = clSetKernelArg (kernel, 2, sizeof(cl_mem),(void *) &buffer_B);
erreur = clSetKernelArg (kernel, 3, sizeof(int), &nbr_colonne_A);
erreur = clSetKernelArg (kernel, 4, sizeof(int), &nbr_ligne_B) ;
erreur = clSetKernelArg (kernel, 1,sizeof(cl_mem),(void *) &buffer_A);
erreur = clSetKernelArg (kernel, 2, sizeof(cl_mem),(void *) &buffer_B);
erreur = clSetKernelArg (kernel, 3, sizeof(int), &nbr_colonne_A);
erreur = clSetKernelArg (kernel, 4, sizeof(int), &nbr_ligne_B) ;
if(erreur != CL_SUCCESS){
   printf ("Echec de l'initialisation des arguments du kernel !") ;
   exit(EXIT_FAILURE);
}

//***** Etape 10 et 11 : Configurer La Structure Des Work-items et du kernel
```

```
size_tglobal_size[2] = {nbr_co_A, nbr_li_B} ;
erreur = clEnqueueNDRangeKernel(command_queue, kernel, 2, NULL,
global_size, NULL, 0, NULL, NULL);
if(erreur != CL_SUCCESS){
    printf ("Echec de l'exécution du kernel !") ;
    exit(EXIT_FAILURE);
}
//**** Etape 12 : lire les données reçues du device par l'hôte
erreur = clEnqueueReadBuffer(command_queue, buffer_C, CL_TRUE, 0,
size_C * sizeof(float),Res, 0, NULL, NULL);
if(erreur != CL_SUCCESS){
  printf("Echec de la lecture des résultats à partir du device!");
  exit(EXIT_FAILURE);
}
//**** Affichage des éléments des matrice A, B et C
printf ( "\n\n" ) ;
printf ( "%s","LaMatrice A : " ) ;
for ( int i=0 ; i <size_A ; i++ ){
    printf ( "%i ", A[i] ) ;
     if ( (i+1)%nbr_co_A == 0 && i != 0){
       printf ( "\n" ) ;
     }
}
printf ("\n") ;
printf ("%s", "La Matrice B : ") ;
printf ("\n\n") ;
for (int i=0 ; i<size_B ; i++ ){
    printf ( "%i  ", B[i] ) ;
```

```
    if ( (i+1)% nbr_co_B == 0 && i != 0 ){
      printf ("\n") ;
    }
}
printf("\n\n");
printf("%s","le resultat du produit des deux matrices A et B : ");
printf("\n\n");
for(int i=0;i<size_C;i++){
    printf("%i  ",C[i]);
    if((i+1)% nbr_colonne_A == 0 && i != 0 ){
      printf("\n");
    }
}

//**** Etape 13 : liberer les ressources d'OpenCl
clReleaseKernel (kernel) ;
clReleaseProgram (programme) ;
clReleaseCommandQueue (command_queue) ;
clReleaseMemObject (buffer_A) ;
clReleaseMemObject (buffer_B) ;
clReleaseMemObject (buffer_C) ;
clReleaseContext (contexte);
return 0;
}
```

5. Conclusion

Dans ce chapitre nous avons présenté les GPUs comme étant des composants parallèles et programmables réalisant des calculs généraux en collaboration avec les CPUs et ainsi le framework OpenCl avec ses modèles et également les étapes à suivre pour écrire un programme OpenCl.

Le prochain chapitre sera dédié à la définition du problème de coloration de graphe et à sa résolution par un algorithme de fourmis.

CHAPITRE 2 :

Les algorithmes de colonies de fourmis pour le PCSG

1. Introduction

Au début de ce chapitre nous allons rappeler quelques notions de la théorie de graphe ainsi que la définition du problème de la coloration, par la suite nous allons parler sur le comportement des fourmis réelles ainsi que l'idée générale de fonctionnement des algorithmes de colonies de fourmis, puis nous allons présenter les trois principales classes d'algorithmes de colonies de fourmis proposées pour la résolution du problème de la coloration des sommets d'un graphe et à la fin nous allons décrire en détail l'algorithme de colonie de fourmi proposée par Plumettaz et al appelé ALS-COL.

2. Problème de coloration de graphe

2.1. Rappel théorique

Afin de fixer les notations et le vocabulaire utilisés, nous rappelons quelques notions de la théorie de graphe.

Définition 2.1. Un graphe non orienté G est un couple défini par:

- Un ensemble fini de sommets V, aussi appelés nœuds.
- Un ensemble fini d'arêtes $E \subseteq \{X \subseteq V : |X| = 2\}$ qui représentent les liens entre les sommets.

Une arête a est une paire de sommets {x , y} avec x , $y \in V$. x et y sont appelées les extrémités de l'arête a.

Le graphe de la figure 2.1 est défini par le couple (V, E) tel que :

- **V**={1, 2, 3, 4, 5}
- E = {{1, 2}, {1, 5}, {1, 4}, {2, 3}, {3, 4}, {4, 5}}

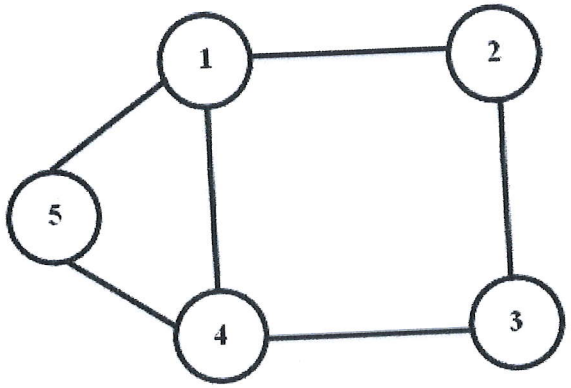

Figure 2.1: Graphe non orienté

Définition 2.2. Dans un graphe G = (V, E), un sommet x est dit adjacent à un autre sommet y si $\{x, y\} \in E$.

Dans le graphe de la figure 2.1, les deux sommets 2 et 3 sont adjacents puisqu'ils sont reliés par l'arête {2,3}.

Définition 2.3. Le voisinage d'un sommet x, noté N(x), est l'ensemble des sommets adjacents à x.

Dans le graphe de la figure 2.1, le voisinage de sommet 1 est N (x) = {2, 4, 5}.

Définition 2.4. Un stable d'un graphe G = (V, E) est un ensemble de sommets $V' \subseteq V$ tel que : $\forall x, y \in V'$, $\{x, y\} \notin E$.

Dans le graphe de la figure 2.1, l'ensemble {3, 5} forme un stable.

2.2. Définition du problème de la coloration

Le problème de coloration des sommets d'un graphe peut être posé de manière claire, à travers un ensemble de définitions [25].

Définition 2.5. Une k-coloration d'un graphe G = (V, E) peut être vue comme une fonction $C:V \rightarrow \{1,2,...,k\}$ qui assigne à chaque sommet x de G un nombre $C(x) \in \{1,2,\dots,k\}$ appelé couleur.

Une coloration définit des classes de couleur V_i ($i \in \{1,2\dots, k\}$) qui forment une partition de l'ensemble V. Chaque classe V_i contient les sommets de couleur i.

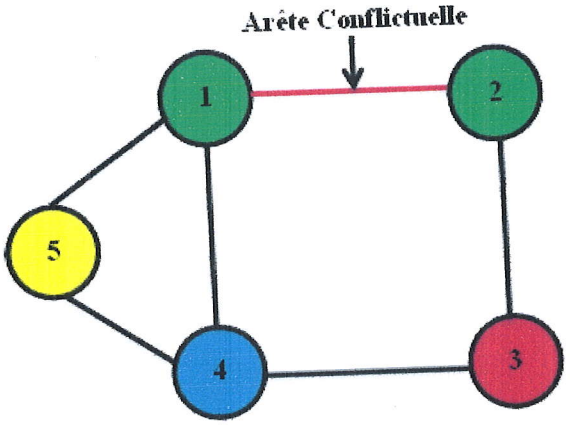

Figure 2.2: Arête conflictuelle

Définition 2.6. Une arête {x, y} est dite conflictuelle si C(x) = C(y) ou, en terme de classes de couleur, si $V_{C(x)} = V_{C(y)}$. Les sommets x, y sont appelés des sommets conflictuels. Par exemple, dans la Figure 2.2, l'arête {1, 2} est conflictuelle.

Définition 2.7. Une coloration est dite partielle (incomplète) si tous les sommets ne sont pas coloriés, autrement dit, si $\exists\, x \in V$ tel que C(x) n'est pas définie.
La Figure 2.3 donne un exemple de coloration partielle où la couleur des sommets 2 et 5 n'est pas définie.

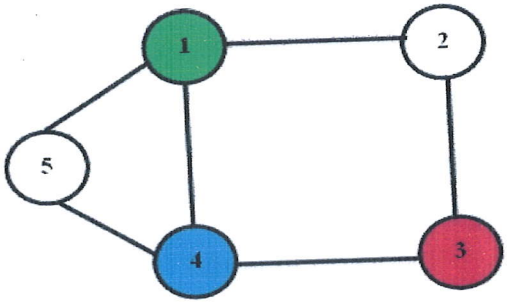

Figure 2.3 : Coloration partielle

Définition 2.8. Une k-coloration d'un graphe G = (V, E) est dite légale si et seulement si :

- cette coloration n'est pas partielle
- aucune arête de E n'est conflictuelle.

La figure 2.4 représente un graphe avec une coloration légale.

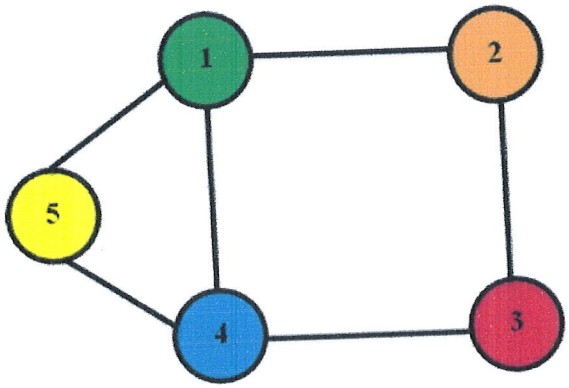

Figure 2.4 : Coloration légale

Définition 2.9. Le nombre chromatique d'un graphe est le nombre minimum de couleurs nécessaires à sa coloration, c'est à dire le plus petit nombre de couleurs permettant de colorier tous les sommets du graphe sans que deux sommets adjacents soient de la même couleur. Ce nombre est noté $\chi(G)$. Par exemple, Dans le graphe de la figure 2.5, le nombre chromatique est 3.

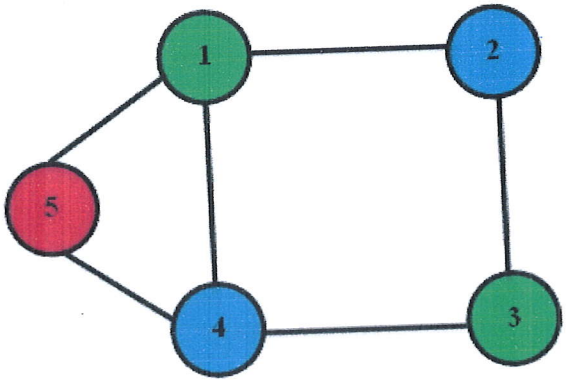

Figure 2.5 : Une coloration légale avec 3 couleurs

Définition 2.10. Le problème d'optimisation de coloration des sommets d'un graphe (PCSG) consiste à déterminer le nombre chromatique d'un graphe G donné. Ce problème est NP-difficile.

Définition 2.11. Le problème de décision de k-coloration d'un graphe (k-PCSG) consiste, pour un k et un graphe G donnés, à déterminer si G est k-colorable. Ce problème est NP-complet.

La plupart des algorithmes d'optimisation considèrent le problème général de coloration comme une série de problèmes de k-coloration de plus en plus difficiles. Cette méthode

commence par fixer une valeur initiale *k* très grande, puis décrémente itérativement k après avoir trouvé une k-coloration sans conflit. Le problème de k-coloration devient de plus en plus difficile jusqu' à ce que l'algorithme ne puisse plus trouver une k-coloration sans conflit. Le *k* le plus petit pour lequel le problème de k-coloration a été résolu constitue la meilleure solution pour le problème général de coloration [26].

3. Algorithmes de colonies de fourmis

Le principe de l'optimisation par colonies de fourmis est apparu au début des années 90 par M. Dorigo, V. Maniezzo et A. Colorni pour résoudre le problème du Voyageur de commerce. Ce sont des algorithmes itératifs à population ou tous les individus partagent un savoir commun qui leurs permet d'orienter leurs futur choix et indiquer aux autres individus des choix à suivre ou à éviter.

Le principe de cette méta-heuristique (ou ensemble d'algorithme) repose sur le comportement particulier des fourmis recherchant le chemin minimal entre leur nid et source de nourriture [27].

3.1. Comportement d'une fourmi réelle

Les fourmis ont la particularité d'employer pour communiquer des substances volatiles appelées phéromones. Elles sont attirées par ces substances, qu'elles perçoivent grâce à des récepteurs situes dans leurs antennes. Les fourmis peuvent déposer des phéromones au sol, grâce à une glande située dans leur abdomen, et former ainsi des pistes odorantes, qui pourront être suivies par leurs congénères [28].

Les fourmis utilisent les pistes de phéromone pour marquer leur trajet, par exemple entre le nid et une source de nourriture. Une colonie est ainsi capable de choisir le plus court chemin vers une source à exploiter, sans que les individus aient une vision globale du trajet.

En effet, comme l'illustre la Figure 2.6 les fourmis les plus rapidement arrivées au nid, après avoir visité la source de nourriture, sont celles qui empruntent les deux branches les plus courtes. Ainsi, la quantité de phéromone présente sur le plus court trajet est légèrement plus importante que celle présente sur le chemin le plus long. Or, une piste présentant une plus grande

concentration en phéromone est plus attirante pour les fourmis, elle a une probabilité plus grande d'être empruntée. La piste courte va alors être plus renforcée que la longue, et à terme sera choisie par la grande majorité des fourmis [29].

Figure 2.6 : Expérience de sélection des branches les plus courtes par une colonie de fourmis (a) au début de l'expérience, (b) à la fin de l'expérience.

3.2. Forme général d'un algorithme de colonie de fourmis

D'une façon générale, les algorithmes de colonies de fourmis sont considérés comme des méta-heuristiques à population, où chaque solution est représentée par une fourmi se déplaçant sur l'espace de recherche. Les fourmis marquent les meilleures solutions, et tiennent compte des des marquages précédents pour optimiser leur recherche.

Un algorithme de colonie de fourmis suit le schéma général algorithmique suivant:

Algorithme 2.1 Algorithme de colonie de fourmis

```
Début
    P := initialiser_pheromones ( ) //la structure qui contient les traces de phéromones.
    Sm := Nil // meilleur solution
    Tanque (critère d'arrêt n'est pas satisfait) Faire
        Site := {}
        Pour i := 1 à nbrfourmis faire
            S := construire_la_solution(P)
            Si (g(S) < g(Sm)) ou (Sm = Nil) Alors //g(S) une fonction d'évaluation
                Sm := S
            Finsi
            Site := Site ∪ {S}
        FinPour
        Mettre_A_Jour_Les_Pheromones (P, Site)
    FinTanque
Fin
```

4. Algorithmes de colonies de fourmis pour le PCSG

Plusieurs algorithmes de colonie de fourmis ont été proposés pour résoudre le PCSC ou le K-PCSG. Selon le rôle que les fourmis jouent dans l'algorithme, ces algorithmes peuvent être regroupés en trois classes [30].

Dans la première classe d'algorithmes, chaque fourmi est un algorithme constructif qui laisse une trace sur chaque paire de sommets non adjacents pour indiquer si ces sommets ont reçu la même couleur. Le premier algorithme de colonie de fourmis pour le PCSG appartient à cette classe et il a été proposé par Costa et Hertz [31].

Dans la deuxième classe d'algorithmes, les fourmis se promènent sur le graphe et tentent collectivement de modifier la couleur des sommets qu'elles visitent. Le but étant de diminuer le nombre d'arêtes conflictuelles dans une k-coloration non légale. Les algorithmes de cette classe

tentent de résoudre le K-PCSG. L'algorithme proposé par Comellas et Ozon [32] et l'algorithme proposé par Hertz et Zufferey [20, 30] sont deux exemples d'algorithmes appartenant à cette classe.

Dans la troisième classe d'algorithmes, les fourmis sont des algorithmes de recherche locale qui laissent des traces sur l'exploration qu'elles ont faites de l'espace de recherche. Les algorithmes de cette classe tentent de résoudre le K-PCSG. L'algorithme proposé par Plumettaz et al [33] et l'algorithme proposé par Shawe-Taylor et Zerovnik [34] sont deux exemples d'algorithmes appartenant à cette classe.

5. Algorithme de Plumettaz et al

Dans l'algorithme proposé par Plumettaz et al [33], appelé ALS-COL, (Algorithme 2.1), Chaque fourmi est un algorithme de recherche locale qui tente d'améliorer sa k-coloration partielle légale dans chaque cycle (génération). Cet algorithme est dérivé de l'algorithme de recherche tabou, appelé PartialCol, proposé par Bloechliger et Zufferey [35].

Soit $V_1,\ldots,V_k$ k ensembles disjoints stables des sommets colorés et V_{k+1} l'ensemble de sommets non colorés. L'objectif de l'algorithme est de minimiser la fonction objective qui représente la taille de l'ensemble V_{k+1} des nœuds non-colorés. Initialement tous sommets du graphe sont dans
l'ensemble V_{k+1}.

Les voisins d'une solution s = ($V_1,\ldots,V_k, V_{k+1}$), sont toutes les colorations que l'on peut obtenir en appliquant un mouvement d'un sommet v de la classe V_{k+1}. Il s'agit de déplacer le sommet v vers une classe V_i ($1 \leq i \leq k$) mais également déplacer tous les sommets adjacents à v de la classe V_i à la classe V_{k+1} pour ne pas créer de conflits. Un tel mouvement sera note m = (v, V_i). Une fois un mouvement m est appliqué, il est ajouté à une liste tabou TL et il devient tabou tout mouvement qui en résulte le déplacement du sommet v vers V_{k+1} pendant tl itérations avec tl définit comme suite :

$$tl = 0.6NC(s) + \text{Random}(0,9). \quad (2.1)$$

Avec $NC(s)$ est le nombre des sommets non coloré dans la solution s actuelle et Random (i_1, i_2) (avec i_1, i_2 des nombres entiers tels que $i_1 < i_2$) est une fonction fournissant un nombre aléatoire

uniforme dans l'ensemble $\{i_1, i_1 + 1, \ldots, i_2 - 1, i_2\}$.

A chaque itération, chaque fourmi modifie sa k-coloration partielle légale en appliquant un mouvement m = (v,V_i). Le choix du mouvement m dépend de deux facteurs :

- L'attrait : l'attrait associe au choix du mouvement m est défini comme l'inverse du nombre de sommets dans V_i qui sont adjacents à v, $A(m) = \frac{1}{|N_{V_i}(v)|}$ **(2.2)**.
- La trace : les traces de phéromones sont mémorisées dans une matrice Tr($|V|$ x $|V|$). Au début de l'algorithme, la matrice Tr est initialisée selon la formule suivante :

$$Tr[v,w] = \begin{cases} 1 & \forall\{v,w\} \notin E \\ 0 & \forall\{v,w\} \in E \end{cases} \quad \textbf{(2.3)}$$

La trace $T(m)$ associe au choix du mouvement m est définie comme suit :

$$T(m) = \sum_{w \in Vi} Tr(v,w) \quad \textbf{(2.4)}$$

En notant M l'ensemble de tous les mouvements m = (v,V_i) non tabous avec $v \in V_{k+1}$ et $i \in \{1,..,k\}$, chaque fourmi doit donc choisir sa prochaine action dans M de façon aléatoire, avec la probabilité P (m) de choisir le mouvement m défini comme suit :

$$P(m) = \frac{A(m)^{\alpha}T(mi)^{\beta}}{\sum_{m \in M} A(m)^{\alpha}T(m)^{\beta}} \quad \textbf{(2.5)}$$

Où α et β sont deux paramètres qui donnent plus ou moins d'importance à l'attrait et à la trace. Pour accélérer leur algorithme, Plumettaz et al proposent d'utiliser une autre stratégie, que nous notons Ψ, de choix plus rapide qui évite le calcul des probabilités P (m). Cette stratégie consiste à déterminer un sous-ensemble $M_1 \subseteq$ M de mouvements d'attrait A(m) maximum. Si C_1 contient plus d'un mouvement, alors on détermine un sous-ensemble $M_2 \subseteq M_1$ de mouvements ayant la plus grande trace T(m). Si M_2 contient plus d'un mouvement, alors un choix aléatoire est effectué.

A la fin de sa recherche locale ($V_{k+1} = \{\}$ et/ou la fourmi à effectuer le nombre d'itérations maximum Nit_{max}), chaque fourmi rajoute une quantité $|V_j|^2$ aux traces $\Delta Tr(v,w)$ sur les paires de sommets (v, w) de même couleur j $\in \{1,...,k\}$ selon la formule suivante :

$$\Delta Tr[v,w] = \Delta Tr[v,w] + |V_j|^2 \text{ , } \forall \{v,w\} \notin E \text{ et } \{v,w\} \subseteq V_j, j = 1, k \quad \textbf{(2.6)}$$

A la fin de chaque cycle, les valeurs de la matrice Tr sont mises à jour selon la formule suivante :

$$Tr[v, w] = \rho * Tr[v, w] + \Delta Tr[v, w], \forall \{v, w\} \notin E \qquad (2.7)$$

Algorithme 2.1 : Algorithme ALS-COL

Début

Initialiser la matrice Tr selon la formule **2.3**

$S_{best} := S$

$K_{Best} := |V|$

Pour ncycles := 1 à $ncycles_{max}$,**Faire**

$\Delta Tr := 0$

Pour i :=1 à nants **Faire**

Nit :=1

$V_{k+1} := V$

Pour i :=1 à k **Faire**

$V_k := \{\}$

Finpour

$S = (V_1,\ldots,V_k, V_{k+1})$

TL := {}

M:= {}

Tant que ($|V_{k+1}| \neq 0$) et (Nit $\leq Nit_{max}$) **Faire**

Calculer l'ensemble des mouvements non tabous, M

Pour chaque mouvement m = $(v,V_i) \in M$ **Faire**

Calculer A(m) selon la formule **2.2**

Calculer T(m) selon la formule **2.4**

Finpour

Choisir un mouvement m = $(v,V_i) \in M$ selon la stratégie Ψ

$V_{k+1} := V_{k+1} - \{v\}$

$V_i := V_i \cup \{v\}$

Pour chaque sommet w $\in N_{V_i}(v)$ **Faire**

$V_{k+1} := V_{k+1} \cup \{w\}$

$V_i := V_i - \{w\}$

Finpour

Décrémenter la valeur tl des mouvements dans TL par 1

Supprimer de TL les mouvements qui ont la valeur tl=0

Calculer la valeur tl du mouvement m selon la formule **2.1**

$TL := TL \cup \{m, tl\}$

$Nit := Nit + 1$

Fintantque

Soit $s = (V_1,\ldots,V_k, V_{k+1})$ une k-coloration réalisé par la fourmi i

Si $|V_{k+1}| < K_{Best}$ **Alors**

$S_{Best} := (V_1,\ldots,V_k, V_{k+1})$

$K_{Best} := |V_{k+1}|$

Finsi

Mis à jour de ΔTr selon la formule **2.6**

Finpour

Mis à jour de Tr conformément à la formule **2.7**

Finpour

Fin

Exemple :

Dans cet exemple nous allons illustrer une itération de la recherche taboue de l'algorithme ALS-COL. Considérons la 3-coloration partielle légale du graphe de la Figure 2.7 ci-dessous.

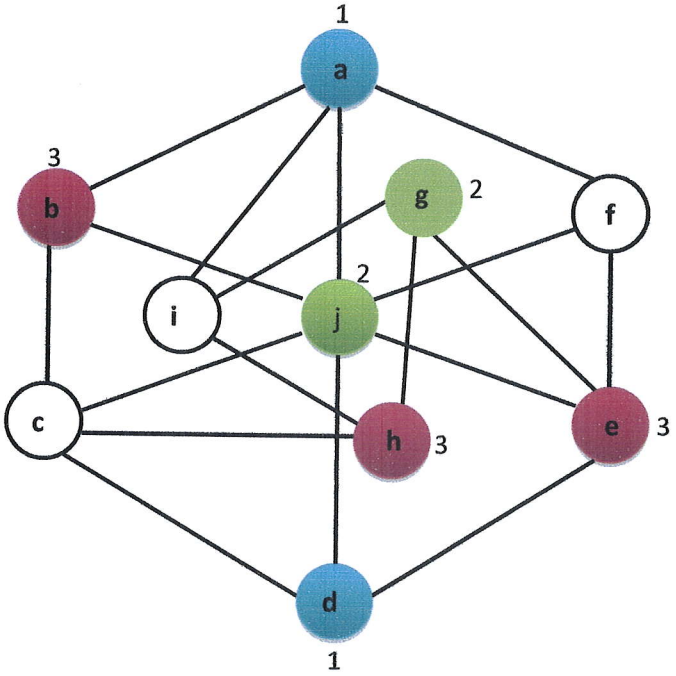

Figure 2.7 : une 3-coloration partielle légale

La solution actuelle est S = (V1, V2, V3, V4) avec :

V_1 = {a, d}, V_2 = {j, g}, V_3 = {e, h, b}, V_4 = {f, i, c}.

On suppose que la liste taboue contient trois mouvements comme il est indiqué dans le tableau 2.1.

Mouvement	Tl
(a, V_1)	1
(j, V_2)	2
(e, V_3)	3

Tableau 2.1 : Contenu de la liste taboue TL

Le tableau 2.2 donne pour chaque mouvement possible m, son attrait A(m), sa trace T(m) et indique s'il est tabou ou non.

M	f,V_1	f,V_2	f,V_3	i,V_1	i,V_2	i,V_3	c,V_1	c,V_2	c,V_3
Movement tabou	Oui	Oui	Oui	Oui	Non	Non	Non	Oui	Non
A(M)					1	1	1		0.5
T(m)					1	2	1		1

Tableau 2.2: Les valeurs de l'attrait, la trace et le type de chaque mouvement possible dans la solution S

Les mouvements m_{i2}= (i, V_2), m_{i3}= (i, V_3) et m_{c1}=(c, V_1) possèdent la plus grande valeur d'attrait, donc l'algorithme de la recherche taboue va choisir le mouvement m_{i3} car il a la plus grande valeur de trace

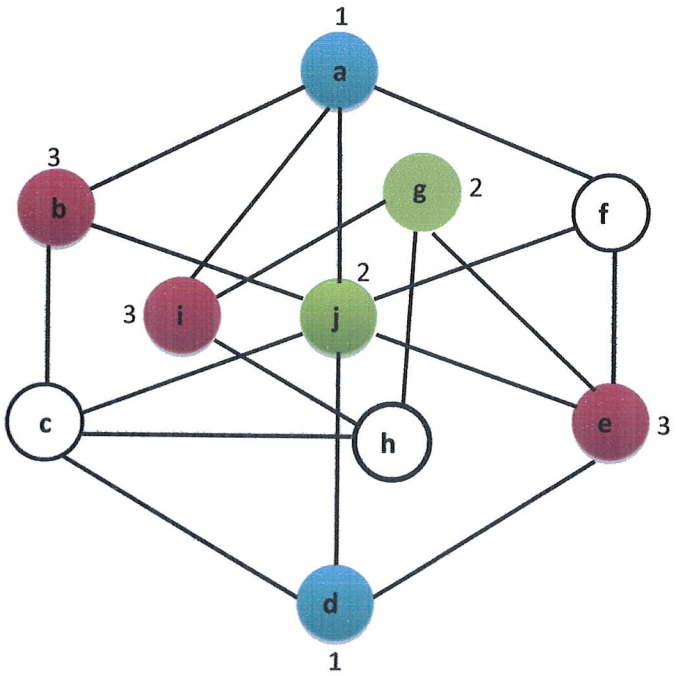

Figure 2.8 : Résultat d'une itération de la recherche taboue

L'application du mouvement m_{i3} consiste à déplacer le sommet i de V_4 vers V_3 et aussi déplacer le sommet h de de V_3vers V_4 parce qu'il est adjacent à i. Cette nouvelle 3-coloration partielle légale est représentée dans la Figure 2.7.

6. Conclusion

Dans ce chapitre nous avons présenté le problème de coloration de graphe avec un algorithme de colonie de fourmi destiné à sa résolution « ALS-COL ».

Le prochain chapitre sera consacré à la présentation de la stratégie suivi pour réaliser l'implémentation parallèle de l'algorithme de fourmis ALS-COL.

CHAPITRE 3 :

Implémentation parallèle basée GPU de l'algorithme ALS-COL

1. Introduction

Au début de ce chapitre nous allons expliquer la stratégie que nous avons suivi pour réaliser une implémentation parallèle sur GPU de l'algorithme ALS-COL en utilisant le framework OpenCL puis nous donnons le code kernel des parties de l'algorithme qui seront exécutées sur le GPU.

2. Stratégie de parallélisation de l'algorithme ALS-COL

Différentes parties de l'algorithme ALS-COL sont constitués de calculs qui sont répétés plusieurs fois (voir l'algorithme 3.1), ce qui les rend des bons candidats pour une parallélisation de données. Ces parties sont :

- L'initialisation de la matrice des phéromones Tr (P1)
- L'initialisation de la matrice ΔTr(P2)
- L'initialisation de la matrice solution S (P3)
- L'initialisation de la matrice TL(P4)
- Le calcul de la trace et de l'attrait de chaque mouvement (P5)
- L'Application du choix d'un mouvement et la mise à jour de la matrice TL (P6)
- La mise à jour de la matrice des phéromones Tr (P7)

Algorithme 3.1 : Algorithme ALS-COL parallèle

Début

Initialiser la matrice Tr selon la formule **2.3**	**P1**

S_{best} := S

K_{Best} :=$|V|$

Pour ncycles := 1 à $ncycles_{max}$ **Faire**

ΔTr :=0	**P2**

V_{k+1} := V

Pour i :=1 à k **Faire**

V_k := {}

Finpour

S = (V_1,…,V_k , V_{k+1}) — **P3**

TL :={}	**P4**

M:= {}

Pour i :=1 à nants **Faire**

Nit :=1

Tant que ($|V_{k+1}| \neq 0$) et (Nit $\leq Nit_{max}$) **Faire**

Calculer l'ensemble des mouvements non tabous, M.

Pour chaque mouvement m= (v,V_i) ∈ M **Faire**

Calculer A(m) selon la formule **2.2**

Calculer T(m) selon la formule **2.4** — **P5**

Finpour

Choisir un mouvement m = $(v,V_i) \in M$ selon la stratégie Ψ

$V_{k+1} := V_{k+1} - \{v\}$

$V_i := V_i \cup \{v\}$

Pour chaque sommet w $\in N_{V_i}(v)$**Faire**

$V_{k+1} := V_{k+1} \cup \{w\}$

$V_i := V_i - \{w\}$

Finpour

Décrémenter la valeur tl des mouvements dans TL par 1

Supprimer de TL les mouvements qui ont la valeur tl=0

Calculer la valeur tl du mouvement m selon la formule **2.1**

$TL := TL \cup \{m, tl\}$

P6

Nit := Nit + 1

Fintantque

Soit $s = (V_1,\ldots,V_k, V_{k+1})$ une k-coloration réalisé par la fourmi i

Si $|V_{k+1}| < K_{Best}$ **Alors**

$S_{Best} := (V_1,\ldots,V_k, V_{k+1})$

$K_{Best} := |V_{k+1}|$

Finsi

Mis à jour de ΔTr selon la formule **2.6**

Finpour

Mis à jour de Tr conformément à la formule **2.7**	**P7**

Finpour

Fin

3. Les kernels

Différentes parties de l'algorithme ALS-COL ont étés paralléliser et écrites sous forme de kernels, Ces kernels sont expliquées dans ce qui suit :

3.1. Initialiser la matrice des phéromones

Afin d'initialiser la matrice des phéromones, nous avons utilisé le kernel suivant :

Programme 3.1 : Kernel OpenCl pour initialiser la matrice des phéromones

```
__kernel void initialiser_matrice_pheromone(int taille,__global int * Madj,
                                            __global float * TR){
uint j = get_global_id(0);
uint i = get_global_id(1);
if((Madj[i*taille + j] == 1) || (i == j))
   TR[i*taille + j] = 0.0;
else
   TR[i*taille + j] = 1.0;
}
```

Ce kernel prend en paramètre le nombre de sommets de du graphe (taille) ainsi que deux vecteurs : Madj qui représente la matrice d'adjacence et TR qui représente la matrice des phéromones.

Pour exécuter le kernel on utilise un NDRange avec deux dimensions dont la taille des deux dimensions 0 et 1 (global-size(x) et global-size(y)) est égal au nombre de sommets du graphe.

Le travail de chaque work-item consiste à affecter la valeur 0 ou 1 à une case du vecteur TR en fonction de la valeur du vecteur Madj.

3.2. Initialiser la matrice DTR

Afin d'initialiser la matrice DTR à chaque cycle, nous avons utilisé le kernel suivant :

Programme 3.2 : kernel OpenCl pour initialiser la matrice DTR

```
__kernel void initialiser_matrice_DTR(int taille,__global float * DTR){
 uint j = get_global_id(0);
 uint i = get_global_id(1);
 DTR[i*taille + j] = 0;
}
```

Pour exécuter ce kernel on utilise un NDRange avec deux dimensions dont la taille des deux dimensions 0 et 1 (global-size(x) et global-size(y)) est égal au nombre de sommets du graphe. Chaque work-item va affecter la valeur 0 à une case de la matrice DTR.

3.3. Initialiser la matrice solution S

La construction de la solution commence toujours par une initialisation de cette dernière, pour cela nous avons utilisé le kernel suivant :

Programme 3.3 : Kernel OpenCl pour initialiser la solution

```
__kernel void initialiser_matrice_Sol(int taille,int nc,__global int * Sol){
uint j = get_global_id(0);
uint i = get_global_id(1);
if (i== nc){
  Sol[i*taille + j] = 1;
 }else{
  Sol[i*taille + j] = 0;
 }
}
```

Pour exécuter ce kernel on utilise un NDRange avec deux dimensions. La taille de la dimension 0 est égal au nombre de sommets du graphe et la taille de la dimension 1 est égal au nombre de couleur plus 1.

Les work-items dont l'ID_global de la dimension 1 est égal au nombre de couleurs (nc) vont initialiser les éléments de la ligne nc (les sommets non colorées) par la valeur 1, par contre les autres work-items vont initialiser les éléments des autres lignes (les stables de la solution) par la valeur 0.

3.4. Initialiser la matrice TL

Afin d'initialiser la matrice TL nous avons utilisé le kernel suivant :

Programme 3.4 : Initialiser la matrice TL

```
__kernel void initialiser_matrice_TL(int taille, int nc, __global int * TL){
 uint j = get_global_id(0);
 uint i = get_global_id(1);
 TL[i * nc + j] = 0;
}
```

Pour exécuter ce kernel on utilise un NDRange avec deux dimensions. La taille de la dimension 1 est égal au nombre de sommets du graphe et la taille de la dimension 0 est égal au nombre de couleur.

Chaque work-item va affecter la valeur 0 à une case de la matrice TL.

3.5. Calculer la trace et l'attrait de chaque mouvement

Le kernel suivant est chargé de calculer l'attrait et la trace associé à chaque mouvement possible.

Programme 3.5 : kernel OpenCl qui indique si un mouvement est tabou et calcul la trace et l'attrait

```
__kernel void calculer_Attrait_Trace_mvtsNT(__global int * sol, __global int * Madj, int taille ,
                                            int nc,__global float * TR,__global float  * Atmv,
                                            __global float * Trmv,__global int * TL){
uint v = get_global_id(0);
uints = get_global_id(1);
if (sol[nc * taille + s]==1){
    // tester si le mouvement est tabou ou non
  int x = 0;
  for(int j=0 ; j<taille ; j++){
    if(sol[v * taille + j] == 1){
      if(Madj[s * taille + j] == 1){
        if(TL[j * nc + v]>0){
          x=1;
          break;
        }
      }
    }
  }
  if(x==0){
    // calculer la trace du mouvement
    float som = 0.0;
    for(int j=0 ; j<taille ; j++){
      if (sol[v * taille + j] == 1){
        som += TR[s * taille + j];
      }
    }
   Trmv[s * nc + v]=som;
```

```
        // calculer l'attrait du mouvement
        int cpt = 0;
        for(int j=0 ; j<taille ; j++){
          if (sol[v * taille + j] == 1){
            if (Madj[s * taille + j] == 1){
              cpt++;
            }
          }
        }
        if(cpt == 0){
          Atmv[s * nc + v]= 2.0;
        }else{
          Atmv[s * nc + v]= 1.0/cpt;
        }
      }else {
        Atmv[s * nc + v]=-1;
        Trmv[s * nc + v]=-1;
      }
    }else{
      Atmv[s * nc + v]=-1;
      Trmv[s * nc + v]=-1;
    }
  }
```


Pour exécuter ce kernel on utilise un NDRange avec deux dimensions. La taille de la dimension 1 est égal au nombre de sommets du graphe et la taille de la dimension 0 est égal au nombre de couleur.

Le travail de chaque work-item consiste à calculer la trace et l'attrait du mouvement qui lui associé.

3.6. Appliquer le choix d'un mouvement et mettre à jour la matrice TL

Pour Appliquer le choix d'un mouvement et mettre à jour la matrice TL nous avons utilisé le kernel suivant :

Programme 3.6 : kernel OpenCl pour appliquer un mouvement

```
__kernel void AppliquerMouvement(int v,int s, int taille, int nc, int t,__global int * Dsnc,
                                 __global   int * sol ,__global int * Madj,__global int * TL){
uint vd = get_global_id(0);
uint s1 = get_global_id(1);
if((vd==v)&&(s1==s)){
  int cpt=0;
  sol[v * taille + s]=1;
  sol[nc * taille + s]=0;
  for(int j=0 ; j < taille ;j++){
     if((sol[v * taille + j]==1)&&(Madj[j * taille + s]==1)){
          sol[v * taille + j]=0;
          sol[nc * taille + j]=1;
          cpt++ ;
      }
  }
  TL[s * nc + v]=t;
  (*Dsnc)=cpt;
}
else{
  if(TL[s1 * nc + vd]> 0){
    TL[s1 * nc + vd]-- ;
  }
}
}
```

Pour exécuter ce kernel on utilise un NDRange avec deux dimensions. La taille de la dimension 1 est égal au nombre de sommets du graphe et la taille de la dimension 0 est égal au nombre de couleur.

3.7. Mise à jour de la matrice des phéromones TR

Pour réaliser la mise à jour de la matrice TR, nous avons utilisé le kernel suivant :

Programme 3.7 : Kernel OpenCl pour la mise à jour de la matrice TR

```
__kernel void mise_a_jourTR(int taille,__global float * TR,__global float * DTR,
                            float r,__global int * mat_adj){
  uint j = get_global_id(0);
  uint i = get_global_id(1);
  TR[i * taille + j] = (r * TR[i * taille + j]) + DTR[i * taille + j];
}
```

Pour exécuter ce kernel on utilise un NDRange avec deux dimensions dont la taille des deux dimensions 0 et 1 égal au nombre de sommets du graphe.

Chaque work-item calcul l'expression (r * TR [i * taille + j]) + DTR [i * taille + j] puis affecte le résultat à une case de la matrice TR identifié par i * taille + j .

4. Conclusion

Dans ce chapitre nous avons présenté la stratégie suivie pour réaliser l'implémentation parallèle sur GPU de l'algorithme de fourmis ALS-COL.

Le prochain chapitre sera consacré aux tests des deux implémentations séquentielle et parallèle de l'algorithme ALS-COL.

Chapitre 4:

Tests et comparaison des résultats

1. Introduction

Ce chapitre est dédié à la représentation des résultats obtenus après avoir effectué un ensemble de tests sur les deux implémentations séquentielle et parallèle de l'algorithme ALS-COL.

Au début nous allons donner une petite description des jeux de données utilisés puis on va présenter les différents résultats obtenus après avoir exécuter les deux implémentations avec différents jeux de données et à la fin nous allons comparer les résultats des différents tests.

2. Les jeux de données

Afin d'expérimenter nos implémentations nous avons utilisé deux types de graphe :
Les graphes de la première catégorie sont des graphes générés grâce à un programme que nous avons développé, ces graphes sont notés $G_{n,p}$ où n est le nombre de sommet, et p est la probabilité d'existence d'une arrête entre deux sommets.

Pour les graphes de la deuxième catégorie ils représentent des instances des graphes de centre DIMACS (Center of Discrete Mathematics and Theorical Computer Science) [http://mat.gsia.cmu.edu/COLOR03], ces derniers constituent le banc d'essai classique pour le problème de coloration et ils ont l'avantage d'avoir des structures très variées et de modéliser des problèmes concrets de la vie réelle.

3. Résultats d'expérimentation

Toutes les expérimentations que nous avons réalisées ont été faites en suivant le paramétrage suivant :

Le nombre de cycle et de fourmis est égal à 10 et le nombre d'itération est 1000000 et pour les graphes à 500 sommets nous avons utilisés que 5 fourmis.

Et avant que nous présentions nos résultats, il faut indiquer que ses expérimentations ont été réalisées sur une machine avec les propriétés suivantes : le CPU est un intel® pentium® 3825U avec une fréquence de 1.9 GHz, une RAM dont la taille est de 4 GBytes et enfin une unité de traitement graphique broadwell GT1 qui offre 12 unités de calcul.

Chaque expérimentation est répétée 10 fois et le temps d'exécution séquentielle ou parallèle indiquée est obtenu on choisissant le meilleur temps d'exécution.

Nous présentons dans le tableau 4.1 les résultats obtenus pour des graphes $G_{n,0.4}$ avec n = 100, 300, 500.

|v|	ALS-COL	Temps séquentiel (s)	Temps parallèle (s)
100	13	0.111079	0.212262
300	29	51.8719	7.68257
500	44	114.169	58.9422

Tableau 4.1 : Les résultats obtenus pour les graphes aléatoires $G_{n,0.4}$

Nous présentons dans le tableau 4.2 les résultats obtenus pour des graphes $G_{n,0.5}$ avec n = 100, 300, 500.

|v|	ALS-COL	Temps sequentiel (s)	Temps parallèle (s)
100	16	0.097087	0.40211
300	36	205.519	34.1672
500	55	171.559	103.642

Tableau 4.2 : Les résultats obtenus pour les graphes aléatoires $G_{n,0.5}$

Nous présentons dans le tableau 4.3 les résultats obtenus pour des graphes $G_{n,0.6}$ avec n = 100, 300, 500.

|v|	ALS-COL	Temps sequential (s)	Temps parallèle (s)
100	18	76.3493	274.826
300	45	237.622	26.3235
500	69	198.988	213.448

Tableau 4.3 : Les résultats obtenus pour les graphes aléatoires $G_{n,0.6}$

Pour les instances des graphes du centre DIMACS les résultats sont présentés dans les tableaux suivants. Pour chacun de ces graphes, nous indiquons son nombre de sommets |V|, son nombre d'arêtes |E|, son nombre chromatique $\chi(G)$ s'il est connu, le plus petit nombre $k*$ pour lequel une heuristique de coloration a produit une k-coloration légale du graphe considéré, ainsi si que le temps séquentielle et parallèle pris pour avoir une coloration.

Le tableau 4.4 représente les résultats obtenus pour quelques graphes DSJC, ces graphes sont créés par David Johnson, ce sont des graphes aléatoires de la forme Gn,p.

graphe	|V|	|E|	$\chi(G)$	K^*	ALS-COL	Temps sequential (s)	Temps parallèle (s)
DSJC125.1	125	736	?	5	5	6.67	7.27602
DSJC125.5	125	3891	?	18	18	62.7866	20.431
DSJC125.9	125	6961	?	44	44	5.8802	1.47399
DSJC250.1	250	3218	?	8	9	1.215	1.71561
DSJC250.5	250	15668	?	28	32	5.15	1.5131
DSJC250.9	250	27897	?	72	76	17.4514	21.4032
DSJC500.1	500	12458	?	12	13	42.6633	36.7386
DSJC500.5	500	62624	?	48	56	44.9299	9.77281

Tableau 4.4 : les résultats obtenus pour les graphes DSJC

Le tableau 4.5 représente les résultats obtenus pour des graphes Queen, ces graphes sont créés par Michael Trick et représente une modélisation au problème de n-reins.

Graphe	|V|	|E|	χ (G)	K*	ALS-COL	Temps sequential (s)	Temps parallèle (s)
queen5_5	25	160	5	5	5	0.002835	1.716
queen6_6	36	290	7	7	7	0.016011	0.202503
queen7_7	49	476	7	7	7	0.418316	0.0624
queen8_8	64	728	9	9	9	2.20358	0.558708
queen9_9	81	2112	10	10	10	3.57856	3.4505
queen10_10	100	2940	?	11	11	997.550320	64.7787
queen11_11	121	3960	11	11	13	0.127091	0.712758
queen12_12	144	5192	?	12	14	0.247176	1.24706
queen13_13	164	6656	13	14	15	3.20328	2.08158
queen14_14	196	8372	?	14	16	3.20328	4.9427
queen15_15	225	10360	?	15	17	25.0708	5.32115
queen16_16	256	12640	?	16	18	15.1157	57.4775

Tableau 4.5 : Les résultats obtenus pour les graphes Queen

Le tableau 4.6 représente les résultats obtenus pour les graphes Miles Graphs, ces graphes sont créés par Michael Trick et ce sont des graphes géométriques, où les sommets de graphe représentent des villes aux Etats-Unis.

Graphe	|V|	|E|	χ (G)	K*	ALS_COL	Temps séquentiel (s)	Temps parallèle (s)
miles250	125	387	8	8	8	0.09308	0.1248
miles500	125	1170	20	20	20	0.237165	0.136097
miles750	125	2113	31	31	31	0.377524	0.174922
miles1000	125	3216	42	42	42	1.40799	0.403392

miles1500	125	5198	73	73	73	1.23588	0.601034

Tableau 4.6 : Les résultats obtenus pour les graphes Miles

Le tableau 4.7 représente les résultats obtenus pour des graphes REG, ces graphes sont créés par Gary Lewandowski, ils représentent des modélisations pour des problèmes basés sur l'allocation des registres.

graphe	|V|	|E|	χ(G)	K*	ALS_COL	Temps sequential (s)	Temps parallèle (s)
mulsol.i.1	197	197	3925	49	49	2.14051	1.349661
mulsol.i.2	188	188	3885	31	31	1.16985	0.530401
mulsol.i.3	184	184	3916	31	31	1.11279	0.496258
mulsol.i.4	185	185	3946	31	31	1.12079	1.11127
mulsol.i.5	186	186	3973	31	31	1.14482	0.608456
zeroin.i.1	211	4100	49	49	49	2.68089	1.72867
zeroin.i.2	211	3541	30	30	30	1.61815	1.67742
zeroin.i.3	206	3540	30	30	30	1.56211	0.985912

Le tableau 4.7 : Les résultats obtenus pour les graphes REG.

Le tableau 4.8 représente les résultats obtenus pour les graphes LEI, ces graphes sont créés par Craig Morgenstern.

graphe	|V|	|E|	χ(G)	K*	ALS_COL	Temps sequential (s)	Temps parallèle (s)
le450_5a	450	5714	5	5	5	90.66	3.4101
le450_5b	450	5734	5	5	5	192.39	16.9356
le450_5c	450	9803	5	5	5	36.508	2.429945
le450_5d	450	9757	5	5	5	51.1249	2.56397
le450_15b	450	8169	15	15	16	310.759	25.1135
le450_15c	450	16680	15	15	19	844.57	107.863

le450_15d	450	16750	15	15	19	157.415	55.7883
le450_25a	450	8260	25	25	25	12.73	2.22249
le450_25b	450	8263	25	25	25	12.7331	1.59243

Le tableau 4.8 : Les résultats obtenus pour les graphes LEI

4. Discussion et comparaison

Pour des interprétations plus claires des résultats présentés dans les tableaux précédant, nous allons illustrer le temps d'exécution (séquentielle et parallèle) par des histogrammes, ou les abscisses représentent les instances d'un certain type de graphe et les ordres représentent le temps d'exécution en seconde.

La figure 4.1 illustre le temps d'exécution séquentielle et parallèle pour des graphes DSJC.

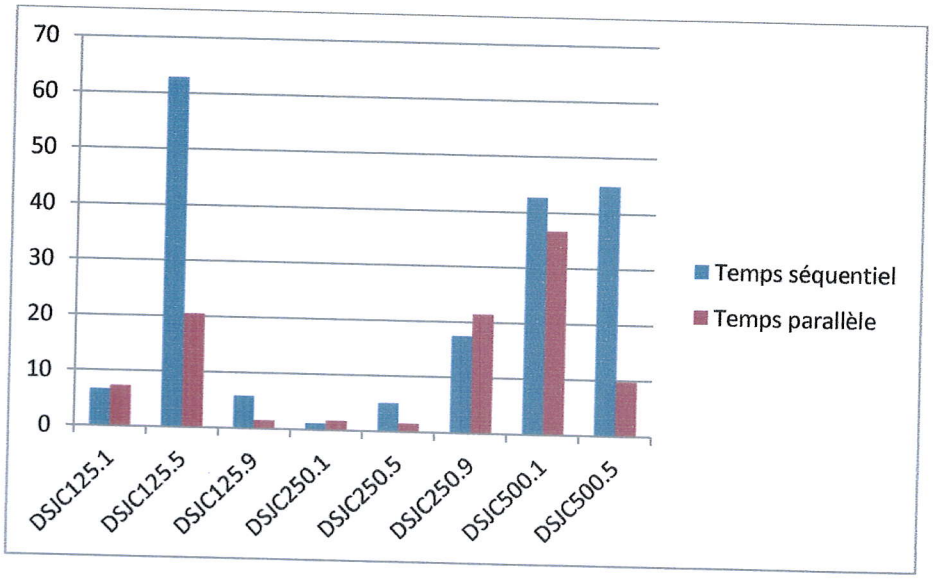

Figure 4.1 : histogramme du temps d'exécution parallèle et séquentiel pour les graphes DSJC

La figure 4.2 illustre le temps d'exécution séquentielle et parallèle pour les graphes Queen

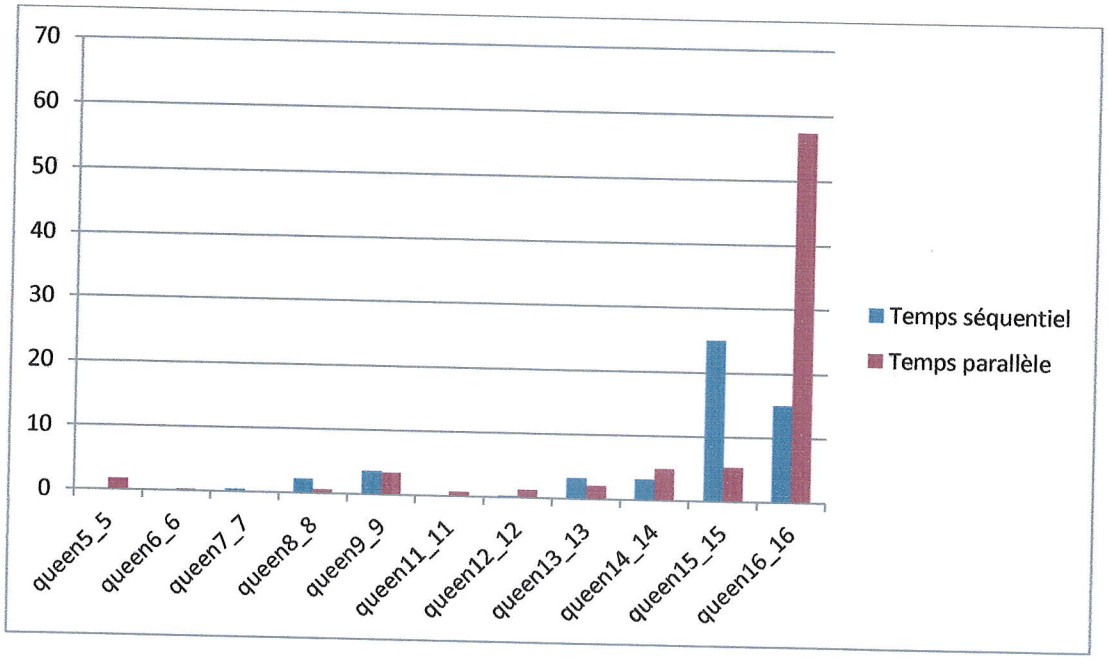

Figure 4.2 : histogramme du temps d'exécution parallèle et séquentiel pour les graphes Queen

La figure 4.3 illustre le temps d'exécution séquentielle et parallèle pour les graphes LEI.

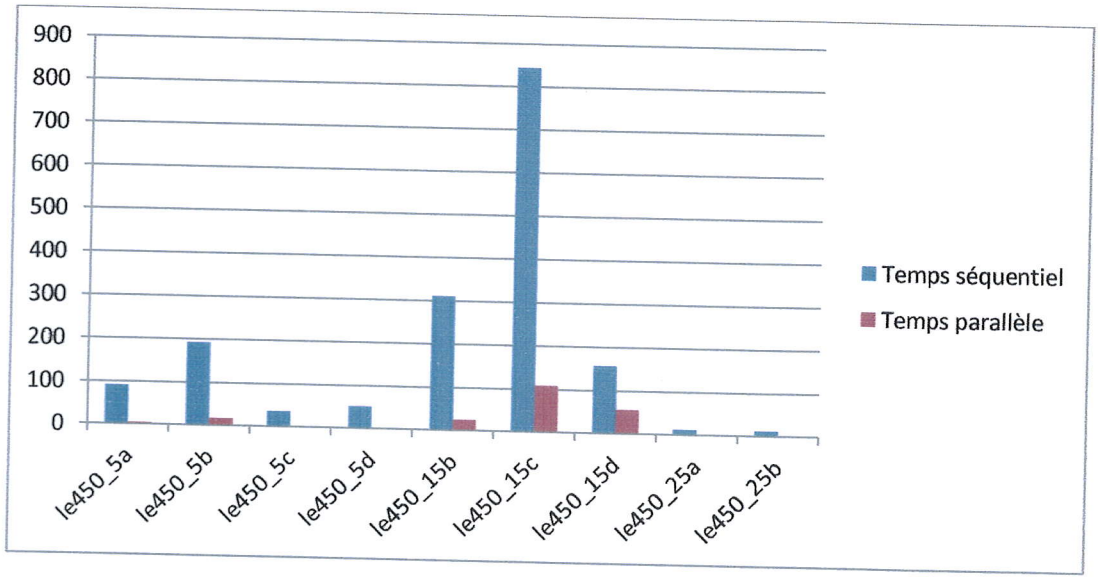

Figure 4.3 : histogramme du temps d'exécution parallèle et séquentiel pour les graphes LEI

La figure 4.4 illustre le temps d'exécution séquentielle et parallèle pour les graphes REG.

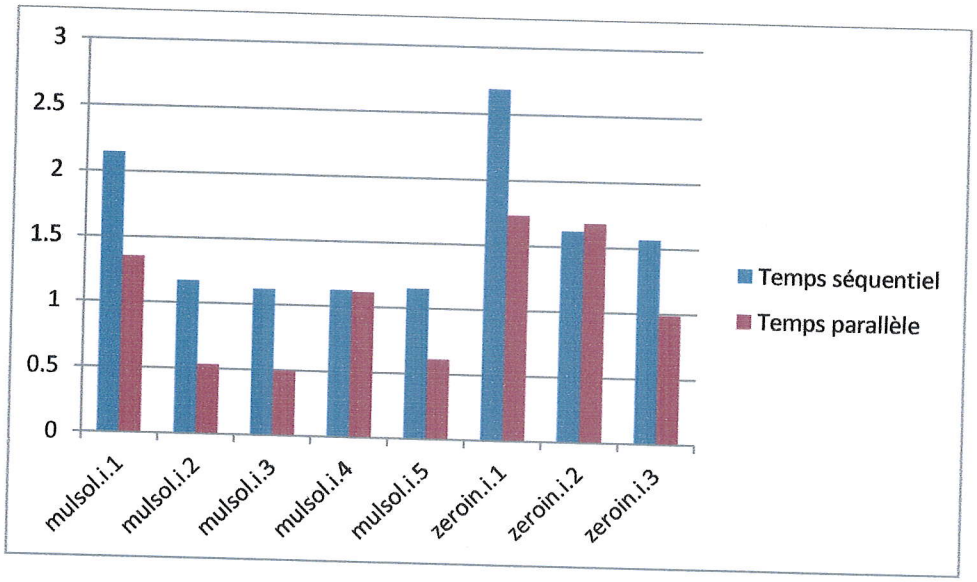

Figure 4.4 : histogramme du temps d'exécution parallèle et séquentiel pour les graphes REG

La figure 4.5 illustre le temps d'exécution séquentielle et parallèle pour les graphes miles.

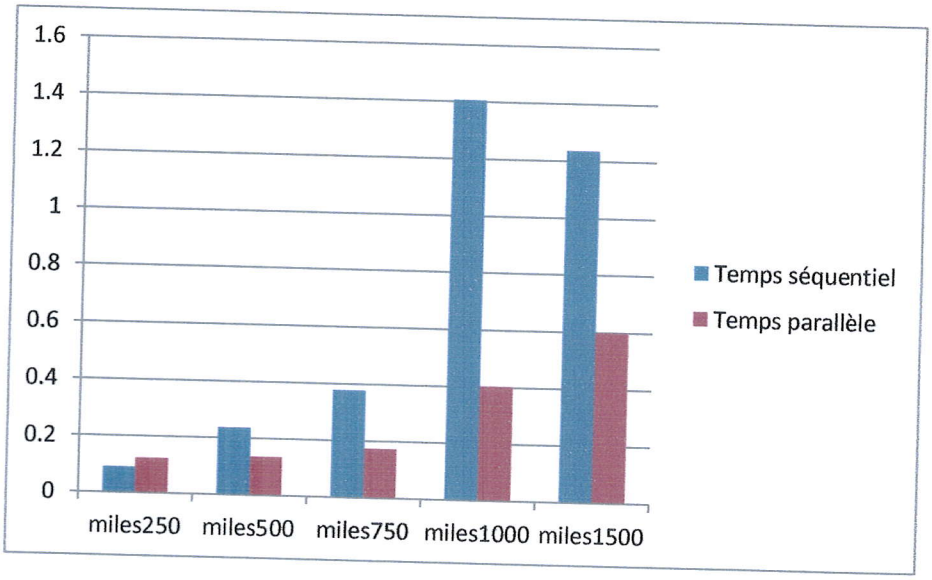

Figure 4.4 : histogramme du temps d'exécution parallèle et séquentiel pour les graphes miles

Depuis les tableaux de section 4.3 et les courbes de la section 4.4, on peut remarquer les point suivants :

- Pour la majorité des graphes il y a une amélioration dans le temps d'exécution en utilisant un GPU.
- le temps d'exécution parallèle est égal aux temps d'exécution séquentielle lorsque l'instance du graphe est petite ou le nombre de sommet est inférieur à 100 (les graphes queen).
- Le temps parallèle n'est pas forcément mieux que le temps séquentiel.
- En utilisant le même paramétrage pour les deux applications on a réussi à obtenir des solutions avec le même nombre de couleur.

Les résultats obtenus montrent que l'implémentation de l'algorithme ALS-COL à base de GPU permet une amélioration remarquable dans le temps d'exécution.

5. Conclusion

Dans ce chapitre, nous avons présenté les résultats d'une expérience que nous avons réalisée, concernant les performances de deux implémentations parallèle et séquentielle de l'algorithme de fourmis ALS-COL et nous avons réalisé une comparaison des résultats obtenus afin de voir s'il y a une amélioration en performances.

Conclusion Générale

Le problème de coloration des sommets d'un graphe (PCSG) est un problème d'optimisation combinatoire qui appartient à la classe des problèmes NP-difficiles. Il a fait l'objet de nombreux travaux de recherche, en raison de ses multiples applications pratiques et de la complexité de sa résolution.

Les algorithmes de colonies de fourmis forment une classe des méta-heuristiques proposées pour résoudre des problèmes d'optimisation difficiles tel que le PCSG. Ces algorithmes s'inspirent des comportements collectifs de dépôt et de suivi de piste observés dans les colonies de fourmis.

Cependant, le temps de calcul de ces algorithmes est sérieusement compromis lorsque l'instance du problème a une dimension élevée. Afin de réduire le temps de calcul, une implémentation parallèle devienne attractive.

L'apparition d'architectures parallèles telles les unités de traitement graphique (GPU) ont permis de nouvelles implémentations parallèles afin d'accélérer la performance de calcul. Les GPUs sont des composants matériels massivement parallèles connus par leur grande puissance de calcul qui dépasse largement celle des CPUs.

Dans ce mémoire nous avons réalisé deux implémentations pour l'algorithme de colonies de fourmis ALS-COL appliqué au PCSG. La première est une implémentation parallèle sur le GPU et la deuxième une implémentation séquentielle sur le CPU. Nous avons également effectué un ensemble de tests pour les deux implémentations sur différent types de graphes.

Les résultats des tests effectués sur les graphes du centre DIMACS et les graphes générés aléatoirement montre que l'implémentation parallèle basée GPU de l'algorithme ALS-COL permettent d'améliorer le temps de calcul si l'instance du problème est suffisamment grande.

Bibliographique

[1] Didier Donsez et Georges-Pierre Bonneau, GPGPU, disponible sur : <http://air.imag.fr/ mediawiki/index.php/GPGPU >.

[2] Christian de Looper, what is the difference between the cpu and gpu ?, disponble sur : <https://www.pcmech.com/article/cpu-vs-gpu/>.

[3] intel, intelxeon phi processor 7290, disponible sur : < https://ark.intel.com/products/9583 0/Intel-Xeon-Phi-Processor-7290-16GB-1_50-GHz-72-core>.

[4] Vedeocardz, Ge Force GTx 1080 Ti, disponible sur : <https://videocardz.com/66557/nvidi a-launches-geforce-gtx-1080-ti-with-3584-cuda-cores>.

[5] Freemake.com, CUDA & DXVA Easily Explained, disponible sur : <http://www.free make.com/blog/cuda-dxva-easily-explained/>.

[6] Vincent Hindriksen , how many threads can run on a gpu ?, disponible sur : < http://str eamcomputing.eu/blog/2017-01-24/many-threads-can-run-gpu/>.

[7] differncebetween.net, difference between cpu and gpu, disponible sur : < http://www. differencebetween.net/technology/difference-between-cpu-and-gpu/>.

[8] Jean Michel Richer, page de Jean-Michel Richer, disponible sur : <http://www.info.uni v-angers.fr/~richer/cuda_crs1.php>.

[9] Anil Rajput, Bhushan Ishwarkar et Sarita Kadhao. Parallel Processing Unit with MIMD Architecture, volume 4, 2277 128X, pp. 1055-1059, 2014.

[10] wikipedia, CUDA, disponible sur : <https://en.wikipedia.org wiki/CUDA>.

[11] NVIDIA, CUDA, Disponible sur : < http://www.nvidia.com/object/cuda_home_new.ht ml>.

[12] Wikipedia ,opencl, disponiblesur , <https://en.wikipedia.org/wiki/OpenCL>.

[13] Sidia Hmed Mahmoudi, Traitement Efficace d'Objets Multimédias sur Architectures Parallèles et Hétérogènes, Mémoire, 29 Janvier 2013.

[14] matthew scarpino, OpenCl in action : How to Accelerate Graphics and Computation, november 2011.

[15] Perhaad Mistry, Dana Schaa, introduction to opencl, Northeastern University Computer Architectur Research Lab, 2011.

[16] Aaftab Munshi, Benedict R. Gaster, Timothy G. Mattson, James Fung et Dan Ginsburg, OpenCl Programming Guide.

[17] khronosOpenCl working group, the opencl specification version : 1.0.

[18] Ravishekhar Banger, Koushik Bhattacharyya, OpenCL Programming by Example.

[19] FixStars, The OpenCl Programming Book, disponible sur : <https://www.fixstars.com/en/opencl/book/OpenCLProgrammingBook/calling-the-kernel/>.

[20] khronos OpenCl working group, the opencl specification version : 2.0.

[21] AMD, introduction to opencl programming.

[22] Apple Inc, Opencl programming guide for Mac OS X.

[23] Cliff Wooley, Introduction to OpenCl.

[24] bharath kumar, OpenCl code structure, disponblesur : <https://gpuenthusiast.Wordpress .com/2012/10/08/opencl-code-structure/>.

[25] Xavier DUBUC, Algorithme de recherche dans des espaces variables appliqué au problème de coloration de graphes. Projet de master 2010Jf.

[26] Daniel Cosmin Porumbel. Algorithmes Heuristiques et Techniques d'Apprentissage : Applications au Problème de Coloration de Graphe. Thèse d'Université d'Angers, 2009.

[27] Wikipedia, Algorithmes de colonie de fourmis, disponible sur : < https://fr.wikipedia .org/wiki/Algorithme_de_colonies_de_fourmis>.

[28] Johann Dréo, Adaptation de la méthode des colonies de fourmis pour l'optimisation en variables continues. Application engenie biomédical. Université Paris XII Val de Marne, 2003.

[29] BELFADEL ET M. DIAF, De la fourmi réelle à la fourmi artificielle. Mémoire, Faculté du Génie Electrique et de l'informatique, Université Mouloud Mammeri de Tizi-Ouzou.

[30] Alain Hertz, Nicolas Zufferey, La coloration des sommets d'un graphe par colonies de fourmis. Les Cahiers du GERAD, G–2008–29 Avril 2008.

[31] Costa D., Hertz A., Ants can colour graphs, Journal of the Operational Research Society, vol. 48, pp. 295–305, 1997.

[32] Francesc Comellas and Javier Oz¶on, Graph Coloring Algorithms for Assignment Problems in Radio Networks. Departament de Matemµatica Aplicada i Telemµatica E.T.S.E. Télécommunication, Université Politµecnica de Catalunya Campus Nord C- (1995).

[33] Plumettaz M., Schindl D, Zufferey N., Ant Local Search and its Efficient Adaptation to Graph Colouring, Submitted for publication, 2007.

[34] Shawe - Taylor J., Zerovnik J., Ants and Graph Coloring, Proceedings of ICANNGA'01 , pp. 593–597, 2002.489-505.

[35] Bl¨ochliger I., Zufferey N., A Graph Coloring Heuristic Using Partial Solutions and a Reactive Tabu Scheme, Computers & Operations Research, vol. 35, pp. 960–975, 2008.

Résumé

Les algorithmes de colonies de fourmis ont été appliqués à un grand nombre de problèmes d'optimisation difficiles tel que le problème de coloration des sommets d'un graphe (PCSG). Cependant, le temps de calcul de ces algorithmes est sérieusement compromis lorsque l'instance du problème a une dimension élevée. Afin de réduire le temps de calcul, une implémentation parallèle devienne attractive.

L'apparition d'architectures parallèles telles que les unités de traitement graphique (GPU) ont permis de nouvelles implémentations parallèles afin d'accélérer la performance de calcul.

Dans ce mémoire nous avons réalisé une implémentation séquentielle et une autre parallèle basée GPU de l'algorithme de colonies de fourmis ALS-COL applique au PCSG ainsi qu'une comparaison entre les résultats obtenus après avoir réalisé un ensemble de test sur différent types de graphes.

Mots clés: algorithmes de colonies de fourmis, architectures parallèles, GPU, ALS-COL, PCSG.

Abstract

Ant colony optimization algorithms have been applied to a large number of difficult optimization problems such as the graph coloring problem (GCP). However, the computation time of these algorithms is seriously compromised when the instance of the problem has a high dimension. In order to reduce computation time, a parallel implementation becomes attractive.

The appearance of parallel architectures such as graphics processing units (GPUs) has allowed new parallel implementations to accelerate computing performance.

In this paper, we carried out a sequential implementation and another GPU-based parallel of the ant-colony algorithm ALS-COL applied to the GCP as well as a comparison between the results obtained after carrying out a test set on different types of graphs.

Keywords: ant colony optimization algorithms, parallel architectures, GPU, ALS-COL, GCP.