Programmation Multicoeur et GPU
Tas de Sable

Kevin Baptista Bérénice Faltrept

1er mai 2016

Table des matières

1 Introduction

1.1 Présentation du problème

Dans ce rapport, nous allons traiter d'un programme de répartition de tas de sable abélien. C'est à dire que le programme va simuler dans un espace 2D la répartition du nombre de grains de sable sur une table. Il est imposé qu'un grain de sable appartienne exactement à une seule case de cet espace 2D, et que toute case interne ayant plus de 4 grains de sables cède 1 grain à chaque case voisine. De plus, les cases au bord du tableau représentant cet espace peuvent contenir un nombre illimité de grains, ceci simulant le bord de la table.Dès lors, on peut voir deux algorithmes possibles.

Le premier, que nous avons appelée **expander**, effectue naïvement une soustraction lorsqu'une case contient plus de 4 grains de sables, et ajoute 1 aux cases voisines. Cet algorithme est simple, mais pose un problème lorsqu'une case a une valeur élevée, obligeant ainsi un grand nombre de calcul. Pour éviter ce problème, on préfère ajouter le résultat de la division entière par 4 de la case visitée aux cases voisines, puis on utilise la division euclidienne par 4 pour qu'il ne reste plus que le reste dans la case visitée. On peut se demander si l'accès en écriture aux cases l'entourant ne sera pas trop long, mais surtout si ce n'est pas gênant lors de la parallélisation. Nous ne nous en sommes donc pas servie dans nos algorithmes parallèles et lui avons préféré la méthode suivante.

Notre seconde méthode, appelée **gatherer**, consiste a lire le contenu de la case à analyser ainsi que le contenu des cases qui lui sont adjacentes pour établir la nouvelle valeur de la case. On a donc uniquement besoin d'accès en lecture aux autres cases et l'écriture est contrainte à une seule case.

Nous avons développés un programme unique pour utiliser tout nos algorithmes, excepté pour l'utilisation d'OpenCL. Ce programme possède un man que vous pouvez afficher avec la commande **./sand**

1.2 Méthode de test de l'amélioration des performances

Pour vérifier nos algorithmes, nous afficherons la matrice obtenue une fois le calcul terminé, et nous redirigerons ce résultat ainsi que l'affichage du temps de calcul dans un fichier .txt afin de pouvoir utiliser la commande "diff". Nous utiliserons comme résultat de référence l'algorithme donné par le sujet.

Pour ce qui est du temps de calcul, nous exécuterons plusieurs fois les algorithmes afin de faire des moyennes, plus représentatives qu'une valeur isolée. Les calculs ont été effectué sur une machine de la salle 203 du CREMI.

2 Programmes Séquentiels

Nous avons produit une version séquentielle naïve (expander) qui correspond à l'algorithme proposé dans l'énoncé. Elle nous sert de référence pour comparer les algorithmes qui suivent.

Nous avons par la suite fait plusieurs tentatives pour produire d'autres algorithmes séquentiels qui ne se sont pas avérés efficaces. Le seul algorithme séquentiel de comparaison nous avons gardé est appelé **multipleline**. Il s'appuie sur le principe du **déroulement de boucle**. Cette technique nous permet de réduire le nombre de contrôles effectués sur la boucle, et ainsi éviter une perte de temps ou une éventuelle mauvaise prédiction de branchement. Concrètement, on effectue n opérations au lieu d'une seule à chaque tour de la boucle, et on incrémente le pas de déplacement de n. Nous nous sommes contentés d'un déroulement de 4 opérations, de manière arbitraire.

2.1 Résultats

Ces valeurs ont été établies sur un ordinateur de la salle 203 du CREMI avec 10 itérations entre deux rafraîchissements.

Dimension	configuration	Algorithme	temps en millisecondes
128	homogène	naif expander	266.900
128	centré	naif expander	555.272
512	homogène	naif expander	63942.729
512	centré	naif expander	4252.023
128	homogène	naif gatherer	595.162
128	centré	naif gatherer	1160.500
512	homogène	naif gatherer	148501.473
512	centré	naif gatherer	32898.397
128	homogène	multipleline expander	240.200
128	centré	multipleline expander	581.835
512	homogène	multipleline expander	58656.005
512	centré	multipleline expander	4757.715
128	homogène	multipleline gatherer	1494.036
128	centré	multipleline gatherer	2306.964
512	homogène	multipleline gatherer	306095.504
512	centré	multipleline gatherer	67872.163

On voit très clairement que la méthode expander est plus efficace que notre gatherer. Ce dernier à d'abord été développé pour les algorithmes parallèles puis porté en séquentiel, nous avons tentés de l'améliorer sans parvenir à le rendre efficace en séquentiel. Le ralentissement vient peut-être du fait que l'on doit stocker deux tableaux dans le cache au lieu d'un, et qu'on écrive donc à des endroits très éloignés de là où on lit. Ainsi, peut-être qu'en stockant les cases des deux tables avec un même x et un même y cote à cote (concrètement, nos actuels ocean[x*DIM+y] et ocean[DIM*DIM+x*DIM+y] deviendrait ocean[x*DIM+y] et

ocean[x*DIM+y+1]), cela accélérerait le programme. De plus, notre "deuxième" tableau, nécessaire pour ne pas faire de calcul avec de fausses valeurs, est alloué dynamiquement. Peut être qu'en utilisant un tableau alloué sur la pile durant l'appel à compute_seq_gatherer on optimiserait également les accès en mémoire.

Le fait de dérouler les boucles peut s'avérer efficace. On remarque par exemple un gain de l'ordre de 10% avec l'expander sur les cas homogènes. En revanche on perd environ 5% de performances sur les cas centrés.
Quant à l'algorithme gatherer, il voit ses temps de calculs exploser avec cette technique.

Nous nous servirons de la méthode expander naive (compute_seq_expander) pour établir nos temps de référence pour le calcul de l'accélération.
Nous n'avons pas su trouver une réel amélioration à cet algorithme, notre version gatherer étant moins efficace, et notre version multipleline, ou unwrapped, n'étant plus efficace que dans certains cas.

3 Programmes Parallèles

Nous avons produit plusieurs versions parallèles utilisant différentes fonctionnalités d'OpenMP.

3.1 Synchronisation à chaque itération

3.1.1 Boucles for réparties

Notre première version de parallélisation utilise des threads qui se répartissent le travail par ligne. Pour ne pas devoir gérer de conflits entre deux threads, on utilise l'algorithme gatherer. De cette manière, on a jamais deux threads qui désirent écrire au même endroit. Nous avions développé une version expander, mais la gestion des conflits rendait notre algorithme plus lent que la version séquentielle.

3.1.2 Task

Nous avons pensés faire une méthode utilisant le système de tâches, elle s'est cependant avérée tellement peu efficace qu'elle était moins intéressante que le code séquentiel.
Le surcoût engendré par la gestion des tâches n'a pas été compensé par leurs traitements parallèles mais surtout, nous ne pouvons laisser les tâches manipuler à leur guise la table sur laquelle on écrit, on doit donc mettre en place une politique de protection sur cette table.
La première idée serait de mettre un *pragma omp critical* sur le segment de code d'écriture mais le fait de bloquer toutes les tâches qui arrivent à ce segment de code à chaque écriture de l'une d'entre elles est beaucoup trop contraignant, nous avons donc essayés d'établir des dépendances entre les tâches.
Mais cela nous oblige à limiter le contenu de chaque tâche :
Notre première idée était de faire traiter une ligne par tâche mais il est alors impossible d'établir la dépendance sur les lignes au dessus et en dessous (elles se bloqueront entre elle).

Nous avons alors été contraint de limiter le contenu de chaque tâche : chacune ne gère que le traitement d'une case. Mais le problème vient du fait que nous avons beaucoup trop de tâches à gérer pour trop peu de code à traiter pour chacune. Le traitement devient lent et on perd tout l'intérêt d'avoir utilisés la parallélisation.

3.1.3 Résultats

Ces valeurs ont été établies sur un ordinateur de la salle 203 du cremi avec un nombre d'itérations de 10 entre deux rafraîchissements.
Notre référence séquentiel pour le speed up est le **multipleline expender** en 128 et le **naïf expender** en 512 étant donné qu'ils sont réciproquement les plus efficaces que nous ayons produits.

TABLE 1 – parallelisation des boucles For

Threads	Dimension	configuration	speed up
4	128	homogène	0.91
4	128	centré	0.75
4	512	homogène	0.73
4	512	centré	2.22
8	128	homogène	0.69
8	128	centré	0.55
8	512	homogène	0.42
8	512	centré	1.30
16	128	homogène	0.72
16	128	centré	0.59
16	512	homogène	0.54
16	512	centré	1.69
24	128	homogène	0.79
24	128	centré	0.63
24	512	homogène	0.68
24	512	centré	2.09

TABLE 2 – parallelisation avec des Tâches

Threads	Dimension	configuration	speed up
4	128	homogène	6.39
4	128	centré	5.40
4	512	homogène	6.83
4	512	centré	18.15
8	128	homogène	6.37
8	128	centré	5.40
8	512	homogène	6.82
8	512	centré	17.88
16	128	homogène	6.40
16	128	centré	5.41
16	512	homogène	6.83
16	512	centré	17.72
24	128	homogène	6.38
24	128	centré	5.39
24	512	homogène	6.83
24	512	centré	17.89

3.2 Synchronisation toutes les p itérations

Nous avons choisit de faire une répartition par lignes et non par dalles. L'algorithme répartie équitablement le nombre de lignes entre tout les threads excepté le dernier qui se charge du surplus de lignes de la division.
Comme indiqué dans le sujet, nous avons eu besoin d'une zone d'influence autour des cases traitées de la taille du nombre d'itérations qui est représentée ici en gris clair. Et nous synchronisons les threads toutes les p-itérations en écrivant les résultats dans la matrice initiale. Nous avons tout de même dû placer une barrière pour synchroniser la lecture entre les threads avant l'écriture. Notre algorithme, bien que plus efficace que le séquentiel, ne donne malheu-

reusement pas le résultat exact. Nous avons un résultat proche mais avons manqués de temps pour chercher l'origine de l'erreur qui fausse le calcul.

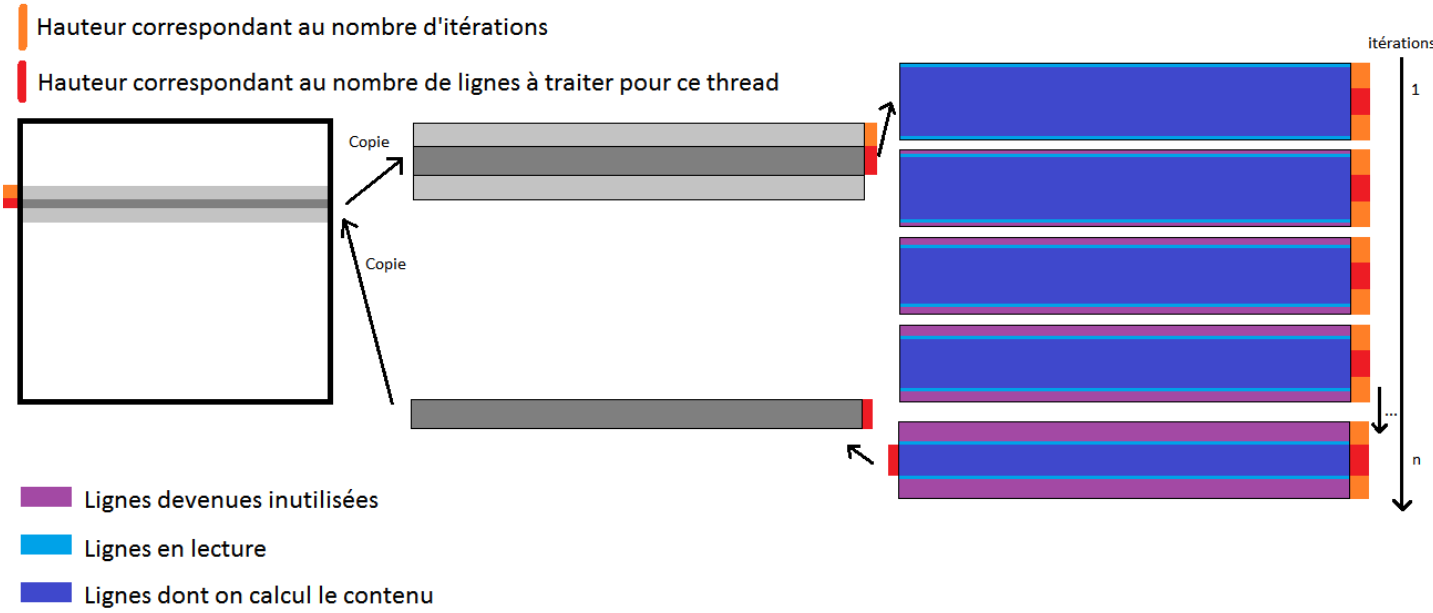

FIGURE 1 – schéma du déroulement de l'algorithme pour chaque thread.

Threads	Dimension	configuration	*speed up*
04	128	homogène	0.98
04	128	centré	0.82
04	512	homogène	0.71
04	512	centré	2.20
08	128	homogène	0.63
08	128	centré	0.54
08	512	homogène	0.40
08	512	centré	1.27
16	128	homogène	0.70
16	128	centré	0.74
16	512	homogène	0.47
16	512	centré	1.52
24	128	homogène	0.90
24	128	centré	0.71
24	512	homogène	0.39
24	512	centré	1.19

TABLE 3 – parallelisation avec synchronisations toutes les 10 itérations

On observe de bons résultats lorsque l'on limite le nombre de threads à 8 alors que l'on s'attendait à avoir les meilleurs *speed up* avec 16 threads. En effet, 24 est le nombre de threads maximal proposé par le matériel utilisé mais ce n'est pas possible de diviser parfaitement équitablement 128 ou 512 par 24 et on s'attendait à une potentielle perte d'efficacité.

4 Programme OpenCL

Nous avons initiés le développement d'un exécutable qui opère le calcul sur la carte graphique mais n'avons pas pu le faire aboutir à un résultat juste. Nous nous sommes basé sur le code du TP-opencl. Nous envoyons donc au GPU un buffer pour écrire le résultat (output_buffer) et un buffer pour lire les valeurs (ocean), ainsi qu'un entier. Celui-ci servira à additionner les valeurs absolues des différences entre la nouvelle valeur et la valeur lue. Ainsi, on saura si le GPU a modifier un élément du tas de sable, et arrêter le programme si il est arrivé à un état stable. Le coup de l'échange de cet entier pourrait cependant se révéler important.

En l'état actuel, le programme OpenCL boucle à l'infinie. En effet, nous envoyons toujours les mêmes donnés au GPU. Il faudrait les arguments. Par exemple, à chaque tour de boucle on incrémenterai un compteur, si il a une valeur pair, on envoie ocean_buffer en premier argument et output_buffer en deuxième, puis si le compteur est impair, on envoie ocean_buffer en deuxième argument et ouput_buffer en premier. On écrirait donc dans un buffer, puis dans l'autre, ainsi de suite.

5 Conclusion

Pour conclure ce projet, nous devons indiquer que nous regrettons de ne pas avoir réussis a obtenir des résultats plus performants dans nos tentatives pour paralléliser notre code. De plus, nous avons manqués de temps et n'avons fait que des tentatives de traitement par lignes alors que nous souhaitions initialement faire aussi du traitement en damier qui aurait sûrement été plus efficace dans certains cas.
Notre projet est donc loin d'être terminé. Il nous aura permit d'appréhender la programmation multi-coeur, et de réellement nous en montrer la difficulté. Nous en apercevons la logique, bien qu'elle nous échappe parfois. C'est un domaine riche et complexe, que nous avons eu parfois du mal à appréhender. Nous avons perdu beaucoup de temps (plus encore qu'en programmation "classique") à tâtonner dans le code, pour voir comment il réagit.
En conséquences, nous n'avons donc pas d'OpenCL fonctionnel, un programme parallèle synchrone toutes les p itérations légèrement faussé, et aucune version asynchrone. Cependant, notre algorithme séquentiel déroulé améliore légèrement l'algorithme d'origine, et notre version parallélisée et synchrone affiche une amélioration inférieure à 50%.