VICENTE Fabien

DALL’AGNOL Tom

ROLLIN Antoine

ANDRE Julien

08/05/2016

SI4

Programmation Concurrente

**TP – Rendu n°5**

Introduction

L’objectif de ce projet est de mettre en pratique la manipulation des threads et les principes de synchronisation vus en cours, en réalisant un simulateur qui modélise le phénomène de transfert de chaleur par conduction et l'évolution de la température en tout point d’une plaque chauffée.

Plusieurs versions[[1]](#footnote-1) intégrées dans le même code seront successivement mises en œuvre :

* *[0] Simulateur avec traitement itératif de la conduction*
* *[1] Simulateur avec traitement multi-tâches synchronisé par barrière Posix*
* *[2] Simulateur avec traitement multi-tâches synchronisé par barrière Posix et variables conditions*
* *[3] Simulateur avec traitement multi-tâches synchronisé par barrière Posix et sémaphores*
* [4] Simulateur exploitant les capacités du GPU

Nous aborderons donc, dans ce document, tous les éléments nécessaires à la configuration et la mise en place des différentes versions, ainsi que leurs résultats en terme de performance.

## Rappel - Version Itérative (Étape 0)

### Principe

Le principe de la version itérative est de mettre en place le processus de conduction de la plaque (matrice) sans parallélisme. Le principe de conduction se décompose en une suite consécutive de traitements verticaux et horizontaux. Lors de chaque traitement, la nouvelle valeur de chaque cellule est calculée grâce à la formule de Tayor (vue en TD).

Pour des raisons de précision, tous les calculs sont effectués en Float. Aussi l’ordre vertical-horizontal n’influe pas sur la cohérence des données finales (nous aurions pu choisir horizontal-vertical).

### Algorithme

INPUT : N (nombre d’itérations), M1 (la matrice principale déjà chauffée), M2 (la matrice secondaire)

Main(N,M1,M2) :

Pour i allant de 1 à N faire

Traiter\_Verticalement(M1, M2)

Traiter\_Horizontalement(M2, M1)

Chauffer(M1)

Fin pour

INPUT : M1 (la matrice source), M2 (la matrice destination)

Traiter\_Verticalement(M1,M2) :

Pour i allant de 0 à M1.longueur-1 faire

Pour j allant de 0 à M1.largeur-1 faire

cellule\_cible = M1[i,j]

voisin\_haut = TEMP\_FROID

voisin\_bas = TEMP\_FROID

Si j >= 1 alors

voisin\_haut = M1[i,j-1]

Fin si

Si j <= M1.largeur alors

voisin\_bas = M1[i,j+1]

Fin si

M2[i,j] =

(voisin\_haut + (H-2)\*cellule\_cible + voisin\_bas)/H

Fin pour

Fin pour

## Rappel - Version avec Barrière POSIX (Étape 1)

### Principe

En dépit du fait que le traitement vertical et horizontal reste le même, la matrice est maintenant découpée en N régions (où N correspond au nombre de thread). Une région est un carré qui se définit par une coordonnée (x,y) et une longueur.

Chaque région est traitée par un thread. Les synchronisations des threads sont faites *via* une ou plusieurs barrières Posix pour assurer l’ordre des traitements et donc la cohérence des données.

Nous choisirons d’implémenter 2 barrières plutôt qu’une seule pour éviter une synchronisation supplémentaire (entre le processus principal et les threads après le traitement vertical).

L’utilisation de thread est destinée à exploiter toutes les ressources de notre processeur multi-cœurs, et donc à améliorer le temps d’exécution du programme. La section critique est représentée ici par la matrice.

### Découpage de la matrice en régions

Le principe de la découpe consiste à créer une première région qui correspond à la matrice entière, puis à la découper récursivement en 4 jusqu’à atteindre le nombre de région souhaité.

INPUT : N (le nombre régions souhaité), S (la taille de la matrice)

OUTPUT : liste (la liste des régions)

Générer\_Régions(N,S) :

liste = Créer\_liste\_regions\_vide()

liste[0] = Créer\_region(0, 0, S)

nb\_region = 1

Tant que (nb\_region < N) faire

liste = Couper\_Régions\_En\_4(liste)

nb\_region \*= 4

Fin tant que

### Processus principal (main)

Le processus principal consiste à :

* Initialiser les barrières
* Lancer les threads pour chaque région
* Attendre la fin des threads
* Détruire les barrières

INPUT : N (le nombre de thread), M1 (la matrice principale déjà chauffée), M2 (la matrice secondaire), IT (le nombre d’itérations)

Main (N,M1,M2) :

regions = Générer\_Régions(N, M1.size)

barrière\_a = Init\_Barrière(N)

barrière\_b = Init\_Barrière(N)

Pour i allant de 0 à N-1 faire

Lancer\_Thread(regions[i], IT, M1, M2)

Fin pour

Attendre\_Threads()

Détruire\_Barrière(barrière\_a)

Détruire\_Barrière(barrière\_b)

### Processus thread

Le processus thread consiste consécutivement à :

* Traiter verticalement la matrice principale
* Attendre la fin du traitement vertical de tous les threads
* Traiter horizontalement la nouvelle matrice
* Chauffer la matrice en fonction de la région
* Attendre la fin du traitement horizontal de tous les threads

INPUT : R (la région du thread), IT (le nombre d’itérations), M1 (la matrice principale), M2 (la matrice secondaire)

Code Thread (R,IT,M1,M2) :

Pour i allant de 1 à IT faire

Traiter\_Verticalement(M1, M2)

Barrière\_Attendre(barrière\_a)

Traiter\_Horizontalement(M2, M1)

Chauffer\_Matrice()

Barrière\_Attendre(barrière\_b)

Fin pour

## Version exploitant les capacités du GPU (Étape 4)

### Principe

L’objectif de cette version est d’exploiter les cœurs du processeur graphique, à l’inverse des autres versions qui s’exécutent sur le processeur « classique ». Elle a pour but d’améliorer les temps d’exécution du programme puisque plus de thread seront exécutés simultanément.

Pour ce faire, nous utiliserons l’API OpenCL permettant de programmer des systèmes parallèles à l’aide des composants de la machine, tels que le GPU.

### Initialisation

L’initialisation du programme consiste en :

* La récupération des informations concernant le GPU que met OpenCL à disposition
* La création d’un contexte pour pouvoir manipuler les matrices à l’aide du processeur graphique
* La création et la compilation du programme appliquant la formule de Taylor sur la matrice, à exécuter sur le GPU
* La création et la compilation du programme chauffant le centre de la matrice, à exécuter sur le GPU

Pour cela, nous aurons besoin de deux fichiers externes contenant la fonction de la formule de Taylor et la fonction chauffant me centre de la matrice, qui sera exécutée par le GPU.

Voici le prototype de la fonction Taylor du premier fichier que nous appellerons « taylor.cl » :

\_\_kernel void taylor(

\_\_global float\* A, \_\_global float\* B,

\_\_global int matrix\_size,

\_\_global int region\_x,

\_\_global int region\_y,

\_\_global int region\_size,

\_\_global int axys

) {…}

Ici, A représente la matrice source et B la matrice destination. Le paramètre matrix\_size permet de détecter les débordements de matrice et de remplacer les valeurs « débordantes » par TEMP\_FROID. Aussi, cette fonction ne va traiter qu’une partie (région) de la matrice puisqu’elle ne sera exécuté que par un unique thread, c’est pourquoi nous passons en paramètre les informations relatives à zone de la matrice à traiter. Enfin, le dernier paramètre indique l’orientation vertical ou horizontal du traitement.

Voici le prototype de la fonction de chauffe du centre de la matrice du second fichier que nous appellerons « heat\_center.cl » :

\_\_kernel void heat\_center(

\_\_global float\* A,

\_\_global int region\_x,

\_\_global int region\_y,

\_\_global int region\_size,

\_\_global int value

) {…}

A représente la matrice à chauffer, les attributs régions déterminent la zone de la matrice à chauffer puisque chaque thread s’occupe d’une partie de la matrice, et value correspond à la valeur des cellules chauffées, soit TEMP\_CHAUD.

Voici maintenant l’algorithme de l’initialisation :

OUTPUT : PT (le programme Taylor à exécuter sur le GPU), PC (le programme de chauffe à exécuter sur le GPU)

Initialiser\_Programme :

taylor = Récupérer\_Fonction(« tayor.cl »)

chauffe = Récupérer\_Fonction(« heat\_center.cl »)

plateformes = Récupérer\_Plateformes\_Disponibles()

gpu = Récupérer\_GPU(plateformes)

contexte = Créer\_Contexte(gpu)

PT = Créer\_Programme(C, taylor)

PC = Créer\_Programme(C, chauffe)

Compiler(PT, gpu)

Compiler(PC, gpu)

Retourner (PT, PC)

### Processus principal et thread

Le processus principal reste le même si ce n’est qu’il doit initialiser le programme à exécuter avec la fonction ci-dessus. Également, le lancement des threads diffère légèrement puisqu’il doit transmettre les deux programmes (taylor et chauffe) à exécuter.

Quant au processus des threads, la différence apparait dans les traitements verticaux et horizontaux des matrices, ainsi que la chauffe de du centre de la matrice, puisqu’ils vont être effectué par le programme GPU.

INPUT : A (la matrice source), B (la matrice destination), R (la région à traiter), P (le programme Taylor à exécuter), O (l’orientation du traitement)

Traiter :

A’ = Bufferiser(A) ; B’ = Bufferiser(B)

liste = Créer\_Liste()

Ajouter(A’, liste) ; Ajouter(B’, liste)

K = Créer\_Kernel(P)

Ajouter\_Argument(K, A’) ; Ajouter\_Argument(K, B’)

Ajouter\_Argument(K, A.size)  ; Ajouter\_Argument(K, R.x)

Ajouter\_Argument(K, R.y)  ; Ajouter\_Argument(K, R.size)

Ajouter\_Argument(K, O)

Exécuter(liste, K)

Récupérer(liste, A)

Récupérer(liste, B)

Néttoyer(liste)

INPUT : A (la matrice source), R (la région à traiter), P (le programme chauffe à exécuter)

Chauffer :

A’ = Bufferiser(A)

liste = Créer\_Liste()

Ajouter(A’, liste)

K = Créer\_Kernel(P)

Ajouter\_Argument(K, A’)

Ajouter\_Argument(K, R.x)

Ajouter\_Argument(K, R.y)

Ajouter\_Argument(K, R.size)

Ajouter\_Argument(K, TEMP\_CHAUD)

Exécuter(liste, K)

Récupérer(liste, A)

Néttoyer(liste)

## Résultats & Observations

## xxx

## Conclusion

Au cours de ce projet, nous avons implémenté successivement différentes approches pour la modélisation de la plaque chauffante.

Une version séquentielle et une version exploitant le parallélisme à l’aide de la barrière Posix, ont été réalisées afin d’étudier les performances de la parallélisation. Bien que la gestion de processus demande une implémentation plus complexe, les résultats ont montré que la deuxième version est plus performante en terme de temps d’exécution. Cela vient du fait qu’elle exploite les capacités de notre processeur multi-cœur tandis que la première version n’utilise qu’un seul cœur.

Une version avec la barrière reconstruite à l’aide d’une variable condition et d’un verrou, et une version avec barrière reconstruite à l’aide de sémaphore ont ensuite été réalisées. Bien que l’utilisation de sémaphores améliore les performances, ces deux versions n’ont pas égalé les performances de la version avec barrière Posix.

Enfin, une version utilisant le processeur graphique a été mise en place. Ce dernier dispose d’un très grand nombre de cœurs, idéal pour la parallélisation de tâches. Nous avons donc pu observer des temps records, écrasant les performances des autres versions.

1. Les versions en italique ne seront pas traitées dans ce document [↑](#footnote-ref-1)