Travail Pratique 4

OpenCL / Cuda

Équipe 7:
Diego Silva Pires
Hervé Simard

8 avril 2018

Introduction

L'exercice consiste à reprendre le code du travail pratique 2 (convolution d'image) et d'en paralléliser le traitement en utilisant les technologies OpenCL/CUDA et OpenAcc.

Tout comme le travail pratique 2, le programme accepte sur une ligne de commande au moins 2 paramètres, soit le le nom du fichier à convoluer et la matrice à appliquer sur le fichier. Un troisième paramètre indiquant le nom du fichier de sortie est optionnel, par défaut le fichier s'appellera "output.png" à moins d'indication contraire. Le code exécutera d'abord la version séquentielle, puis créera un second fichier "output2.png" avec le code parallèle.

Par la suite nous voulons exposer le temps que le programme séquentiel a pris pour convoluer l'image versus le temps pris par l'implantation parallèle afin d'en comparer la performance et, ainsi, établir le speedup obtenu.

Malheureusement nous avons connu beaucoup de problèmes de configurations durant la réalisation du travail pratique et n'avons pu obtenir de résultats concluants. Nous vous présentons tout de même le résultat de nos travaux.

Matériel

OpenCL

Processeur: 2.9 GHz Intel Core i5
Memoire: 8 GB 1867 MHz DDR3
Carte graphique: Intel Iris Graphics 6100 1536 MB

OpenAcc

Processeur: Intel Core I5-4670K CPU
Mémoire: 8 gigs
Carte graphique: AMD Radeon HG 5700 series
Mémoire GPU : 4850mb approx

Procédé

OpenCL

OpenCL été choisi au lieu de CUDA parce que ce dernier ne marche pas sur une carte graphique Intel et aussi parce que OpenCL viens par défaut dans Mac.

Néanmoins l'utilisation d'OpenGL s'est démontrée très problématique, à commencer par l'inclusion de la bibliothèque dans le code:

```
#ifdef __APPLE__
#include <OpenCL/opencl.h>
#else
```

```
#include <CL/cl.hpp>
#endif
```

Pour MAC, c'est la librairie "opencl.h" qui est disponible, ce qui fait que le code écrit est différent de celui rencontré dans quelques tutoriels internet. Cela peut s`expliquer pour la version d'OpenCl en question.

Le plan de match pour paralléliser en utilisant OpenCL était ajouter dans le kernel la boucle responsable de la convolution:

```
__kernel void filter(
__global const uchar *lImage,
__global const int *lHalfK,
__global const int *lWidth,
__global const int *lFilter) {

   int fy, fx;
   //Variables temporaires pour les canaux de l'image
   double lR, lG, lB;
   for(int x = lHalfK; x < (int)lWidth - lHalfK; x++)
   {
       for (int y = lHalfK; y < (int)lHeight - lHalfK; y++)
       {
           // Code de la convulation
       }
   }
}
```

Le code en question est le même donné comme base pour le TP2, aucun changement n'a été fait. Une amélioration possible, proposée aussi pour d'autres équipes lors de la révision du TP2, était de créer une copie du vecteur de l'image pour éviter une race condition.

Malheureusement nous n'étions pas capable de faire s'exécuter le projet sans erreur avant d'essayer cette possibilité.

Toutes les commandes d'initialisation d'OpenCl s'exécutent bien jusqu'à la commande "clEnqueuWriteBuffer" qui ne fonctionne pas:

```
ret = clEnqueueWriteBuffer(command_queue, bufferlImage, CL_TRUE, 0, size, lImage.data(),
0, NULL, NULL);
if (ret != CL_SUCCESS)
{
   cout << "Error: Failed to write to source array &Image!" << getErrorString(ret) <<
"\n";
   exit(1);
}
```

À tout temps nous recevions un code d'erreur -30 qui, selon la documentation, signifiait "CL_INVALID_VALUE".

Malgré les tentatives de changer la façon de passer le paramètre de l'image avec "data()", et l'augmentation de la taille du buffer comme suggéré par d'autres étudiants, le résultat est demeuré négatif.

Les recherches successives sur internet pour une solution possible n'ont pas apporté de solution à ce sujet.

OpenAcc

L'objectif était d'utiliser openAcc afin de confier les calculs à la carte graphique. Le GPU est taillé sur mesure pour compléter ce type de travail, ainsi openAcc permet de définir des directives indiquant quoi faire avec le code et les boucles.

La première étape consistait à voir ce que nous pouvions faire avec les doubles boucles:

```
//#pragma acc parallel loop // Ne fonctionne pas : 'lToken'
referenced in target region does not have a mappable type
//#pragma acc kernels
for (int i = 0; i < lK; i++) {
   for (int j = 0; j < lK; j++) {
        lTok.getNextToken(lToken);
        lFilter[i*lK+j] = atof(lToken.c_str());
   }
}
```

Vous remarquerez que la parallélisation n'a pas été appliquée sur la double boucle; en effet, openAcc ne permettait pas d'appliquer un parallélisme quelconque à cause des variables lTok et lToken qui n'était pas des types pouvant être mappés.

L'hypothèse à ce moment est que la boucle aurait certainement pu être réécrite afin de permettre la parallélisation sur des types mappables qui pourraient ensuite être appliqués sur lTok et lToken.

Nous avons par la suite appliqué une autre directive sur le vecteur d'image:

```
vector<unsigned char> lImage; //Les pixels bruts

#pragma acc data create(lImage)
```

L'idée ici était que par la suite le vecteur allait être recalculé entièrement par la carte graphique, donc autant lui confier tout de suite le vecteur pour accélérer le processus.

Enfin, il y a la double boucle de calcul avec la double boucle de convolution ou nous avons appliqué différent principe de parallélisation afin d'en accélérer le processus, les voici:

```
#pragma acc kernels loop gang(32)
// #pragma acc parallel loop
for(int x = lHalfK; x < (int)lWidth - lHalfK; x++)
{
   #pragma acc loop independent
   for (int y = lHalfK; y < (int)lHeight - lHalfK; y++)
   {
```

Ici l'idée était que la première double boucle enveloppait le reste du traitement, il fallait donc la lancer en lui indiquant d'initialiser un nombre fixe de threads, soit 32, puisque l'hypothèse valait que la carte graphique possède des blocs de 32 processeurs.

Ensuite:

```
#pragma acc loop independent reduction(+:lR, lG, lB)
for (int j = -lHalfK; j <= lHalfK; j++) {
   fy = j + lHalfK;

   for (int i = -lHalfK; i <= lHalfK; i++) {
       fx = i + lHalfK;
       //R[x + i, y + j] = Im[x + i, y + j].R * Filter[i, j]
       lR += double(lImage[(y + j)*lWidth*4 + (x + i)*4]) * lFilter[fx +
fy*lK];
       lG += double(lImage[(y + j)*lWidth*4 + (x + i)*4 + 1]) *
lFilter[fx + fy*lK];
       lB += double(lImage[(y + j)*lWidth*4 + (x + i)*4 + 2]) *
lFilter[fx + fy*lK];
   }
}
```

Ici l'hypothèse voulait que le parallélisme s'effectue en ramenant les valeurs calculées en réduction à la fin dans les variables lR, lG et lB afin d'effectuer l'assignation par la suite dans le vecteur d'image.

Résultats partiels

Nous ne sommes pas en mesure de produire de résultats tangibles puisque nous avons connu passablement de difficulté avec les configurations des technologies sur nos ordinateurs.

OpenAcc

Dans le cas d'openAcc nous utilisions à l'origine une machine virtuelle avec Ubuntu, mais nous avons rapidement constaté que l'accès à la carte graphique n'était pas disponible directement. Il fallait installer des programmes qui permettait à la machine virtuelle d'accéder à la carte graphique.

Il a finalement été possible de rouler le code openAcc… sous Windows 10. Pour ce faire, pour la compilation, il était impératif de supprimer la dll de cygwin du dossier System32 de Windows puisqu'elle empêchait le programme de compiler le code. Mais pour l'exécution, il fallait absolument ramener la dll dans le dossier puisqu'elle était nécessaire pour l'exécution du code parallèle, d'où un beau va-et-vient incessant.

Les résultats aussi se sont avérés un peu décevant puis qu'aucun speedup réel n'a été observé, les temps étant sensiblement identiques entre l'exécution du code séquentiel et parallèle.

OpenCL

Concernant l'utilisation d'OpenCL avec MAC, malgré la présence de la librairie par défaut dans la machine, nous ne sommes pas parvenu à faire exécuter le code en question même si nous étions capables de faire exécuter un exemple simple de somme trouvé sur internet.