Programmation Graphique Haute Performance
Initiation à la programmation CUDA

February 3, 2016

L'objectif de ce TD est de s'initier à la programmation CUDA au travers de petits exercices de traitement d'images.

Ressources

- CUDA C Programming Guide.
- CUDA Toolkit Reference Manual.
- **Qt** est un toolkit graphique portable écrit en C++ permettant la création d'applications graphiques complexes. Dans le cadre de ce TDs CUDA, cette bibliothèque sera utilisée uniquement pour le chargement et sauvegarde d'images. (documentation).
- Cmake sera utilisé pour gérer la compilation du projet. (tutorial, documentation).

Les guides de programmation et de référence CUDA ce trouvent également dans le répertoire : `/usr/share/doc/nvidia-cuda-doc/`.

1 Prise en main

Téléchargez et décompressez l'archive associée au sujet.

```
$ wget http://www.labri.fr/perso/guenneba/pghp_IO15/pghp_2015_td1.tgz
$ tar xzf pghp_2015_td1.tgz
```

Le projet contient:

- un répertoire **cmake/** contenant des scripts facilitant la compilation des fichiers .cu par nvcc et gcc avec Cmake,
- un fichier **CMakeLists.txt** configurant CUDA et l'exécutable à générer via cmake,
- un répertoire **data/** contenant quelques images pour les tests,
- le répertoire contenant les fichiers sources **src/**.

Pour compiler le projet avec cmake il faut créer un répertoire de *build* :

```
$ mkdir build-td1
$ cd build-td1
```

Puis configurer le répertoire de build :

```
$ cmake ../pghp_2015_td1
```

Si CUDA/nvcc n'est pas automatiquement détecté, vous pouvez préciser son emplacement avec, par exemple:

```
$ cmake ../pghp_2015_td1 -DCUDA_INSTALL_PREFIX=/opt/local/cuda
```

Cela crée les "Makefile" nécessaires à la compilation du projet (voir plus loin).

Dans ce TD, nous commenceront par considérer les images comme de simples tableaux 1D : l'utilisation d'images permet de facilement vérifier les résultats des calculs que nous réaliseront.

Pour l'instant, l'application réalise les opérations suivantes:

- Chargement de l'image source via la classe **FloatImageGray** (main.cpp, fonction main()). La classe **FloatImageGray** est définie dans le fichier **FloatImage.h**, elle permet de charger et sauvegarder facilement une image en niveau de gris qui est elle même représentée en mémoire comme un simple tableau de float (**std::vector<float>**). La méthode **values()** de **FloatImageGray** permet d'obtenir une référence sur ce tableau. Si l'image en entrée n'est pas une image en niveau de gris, elle est alors automatiquement convertie pour vous. Ici, 0 représente du noir, et 1 du blanc.
- Cette image, ou plutôt tableau de *floats* est ensuite traitée par la fonction **binarize_cuda()** dont un squelette est donné dans le fichier **binarize.cu**. Écrire le code de cette fonction fera l'objet de la première partie de ce TD (voir plus loin).
- A la fin de la fonction main(), l'image est sauvegardée sur disque.

2 Binarisation

Pour ce premier exercice, vous devrez compléter la fonction **binarize_cuda()** et le kernel CUDA **binarize_kernel()** afin de retourner une image en noir (0) et blanc (1) en utilisant le seuil **threshold**. Pour vous guider, la structure générale du code vous est donnée, et vous devez remplacer les **/* TODO */**.

Le rôle de la fonction **binarize_cuda()** est de faire le lien entre l'application et le kernel CUDA. Cette fonction doit copier les données du CPU vers le GPU, appeler le kernel, puis rapatrier le résultat du GPU vers le CPU. Pour cela vous aurez besoins des fonctions cudaMalloc, cudaMemcpy, et cudaFree qui sont expliquées pages 15 et 16 du support du cours (vous pouvez vous référer également au CUDA Reference Manual). Au besoins, vous pouvez tester les retours d'erreurs des fonctions CUDA en les encapsulant dans la macro **CUDA_SAFE_CALL**. Cette macro affiche un message sur la sortie standard le cas échéant. Exemple:

```
CUDA_SAFE_CALL( cudaMalloc(...); )
```

Pour appeler votre kernel, utilisez la macro **iogs_launch_kernel_1D** dont la syntaxe d'utilisation est la suivante :

```
iogs_launch_kernel_1D(name_of_the_kernel_function, number_of_threads)(arg0, arg1, ...);
```

Attention, cette macro génère au minimum **number_of_threads** threads et en pratique, un plus grand nombre de threads peuvent être générés.

Finalement, la dernière étape consiste à implémenter le kernel **binarize_kernel()**. Pour cela, il faut commencer par calculer sur quel élément le thread courant va travailler. Pour cela, vous utiliserez la fonction **iogs_thread_id_1D()** qui retourne le numéro du thread courant.

Pour tester, compilez le projet avec la commande make:

```
$ make
```

Cela crée un exécutable **imgfilter** prenant deux arguments, un fichier image source et un fichier image de destination. Pour tester:

```
$ ./imgfilter chemin_vers_td_cuda/data/lena.png lena_bin.png
```

Si des messages d'erreurs apparaissent, lancez la commande **glxinfo** afin d'activer le GPU (parfois nécessaire après un redémarrage du PC). L'image **lena_bin.png** doit être composée uniquement de pixels noir et blanc. Si il s'agit d'une image en niveau de gris, alors votre code n'est pas correct. Une fois que cela fonctionne, testez avec une image haute résolution (ex: **data/highres_img1.jpg** puis **data/highres_img2.jpg**).

3 Application d'un filtre 3×3

L'objectif de ce deuxième exercice est de convoluer l'image en entrée par un filtre discret 3×3 en utilisant CUDA. Un exemple de masque d'un filtre passe-bas est fourni dans la fonction main() et une ébauche de code vous est fourni dans le fichier convolution.cu. Pour des raison de performance, le filtre 3×3 sera transféré au kernel via la mémoire constante. Cette étape est déjà implémentée dans le fichier convolution.cu (variable globale déclarée avec **__constant__**, et copie des valeurs du filtre avec la fonction **cudaMemcpyToSymbol**).

Dans cet exercice nous devons accéder aux pixels voisins du pixel courant. Nous ne pouvons donc plus considérer notre image comme un simple tableau 1D. C'est pour cela que la fonction **apply_3x3filter_cuda** prend en paramètre un objet de type **FloatImageGray** et non un simple **std::vector**. Afin de simplifier l'accès aux voisins, il est recommandé de générer une grille 2D de threads correspondant au dimensions de l'image avec la macro **iogs_launch_kernel_2D** :

```
iogs_launch_kernel_2D(name_of_the_kernel_function, nx, ny)(arg0, arg1, arg2, ...);
```

qui lance $nx \times ny$ threads. Dans le kernel, vous pouvez obtenir les coordonnées 2D du thread courant avec la fonction **iogs_thread_id_2D()** qui retourne un objet de type **int2** :

```
int2 tid = iogs_thread_id_2D();
tid.x <=> coordonnée x du thread au sein de la grille 2D
tid.y <=> coordonnée y du thread au sein de la grille 2D
```

Dans un premier temps, vous ferez l'hypothèse que le filtre n'est appliqué qu'une seule fois (n=1). Testez en mettant à jour le fichier main.cpp. Une fois que cette première étape est validée, adaptez votre code afin d'appliquer le filtre un nombre arbitraire de fois.