Programmation Graphique Haute Performance
Initiation à la programmation CUDA

March 24, 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.
- 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

Télécharger et décompresser l'archive associée au sujet.

```
$ wget http://www.labri.fr/perso/guenneba/pghp_2016/pghp_2016_td11_cuda.tgz
$ tar xzf pghp_2016_td11_cuda.tgz
```

Le projet contient:

- un fichier `CMakeLists.txt` configurant CUDA et l'exécutable à générer via cmake,
- un répertoire `data/` contenant quelques images pour les tests,
- le répertoire des fichiers sources `src/`.

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

```
$ mkdir build-td11
$ cd build-td11
```

Et configurer le projet avec cmake :

```
$ cmake ../pghp_2016_td11_cuda
```

Si la version par défaut de gcc n'est pas supporté par nvcc, utiliser une version antérieur avec, par exemple :

```
$ cmake ../pghp_2016_td11_cuda -DCUDA_NVCC_FLAGS="-ccbin /usr/bin/gcc-4.8"
```

Cela crée les Makefile nécessaires à la compilation du projet avec make (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, et la méthode `raw_data()` retourne un pointer vers les valeurs des pixels. 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 plutot 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 donné, et vous devez remplacez 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 51 et 52 du support du cours (vous pouvez vous référer également au CUDA Reference Manual). Remarquez que les appels aux fonctions CUDA sont encapsuler dans la macro `CUDA_SAFE_CALL` qui test les retours d'erreurs et affiche un message sur la sortie standard le cas échéant. **Rappel sur les** `std::vector<float>` : vous pouvez obtenir l'adresse du premier élément d'un std::vector de la manière suivate:

```
std::vector<float> values;
values[i];                    // accès au i-th élément
int n = values.size();        // nombre d'éléments dans le tableau
float* ptr = values.data();   // addresse du premier élément, ptr[i] <=> values[i]
```

Les variables `DimBlock` et `DimGrid` permettent de définir respectivement le nombre de threads par block et le nombre de blocks. Pour l'instant nous manipulons des données 1D, donc ces variables peuvent être définie de la sorte:

```
dim3 DimBlock(nb_threads_per_block);
dim3 DimGrid(nb_blocks);
```

Le nombre total de threads générés est alors `nb_threads_per_block*nb_blocks`. Celui-ci doit être égale (ou supérieur) au nombre de valeurs à traiter (1 thread traite 1 seule valeur). **Attention :** le nombre de threads par block est limité, et dim3 stocke des `unsigned short` dont la valeur maximal est 65535. La limite sur le nombre de threads par block varie de 256 à 2048 en fonction du matériel.

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 aurez besoins des variables suivantes qui sont automatiquement définies par CUDA :

```
threadIdx.x    : numéro du thread actif au sein du block courant
blockIdx.x     : numéro du block courant
blockDim.x     : ici il sera égale à nb_threads_per_block
```

Pour tester, vous devez compiler 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 ../pghp_2016_td11_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`). **Attention**, si l'image est trop grande, il ne sera pas toujours possible de lancer autant de threads que de pixels en utilisant des blocks et une grille 1D. Adaptez votre code pour qu'un thread traite plusieurs pixels. Finalement, vous pouvez obtenir le nombre maximal de thread par block pour votre GPU avec le code suivant :

```
cudaDeviceProp devProp;
cudaGetDeviceProperties(&devProp, 0);
int maxThreadsPerBlock = devProp.maxThreadsPerBlock;
```

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. Vous avez également une implémentation CPU de référence dans le fichier main.cpp dont le résultat est sauvegardée dans le fichier cpu_convolution.png ce qui permet de comparer les performances et de vérifier les résultats. Pour des raison de performance, le filtre 3×3 sera transféré au kernel via la mémoire constante. Cette étape est déjà implémenter 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. Dans le kernel, vous calculerez les coordonnées 2D du thread courant à partir des coordonnées x et y des variables threadIdx, blockIdx, et blockDim. 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 afin d'activer la convolution. Une fois que cette première étape est validée, vous adapterez votre code afin d'appliquer le filtre un nombre arbitraire de fois.

Travail à rendre : votre version des fichiers `binarize.cu` et `convolution.cu` par email.

Bon courage !