Subject-Project-fr.md - Grip

TP Partie 0 : faire fonctionner votre GPU

Quel est votre matériel ?

Pour découvrir quel matériel est installé sur votre machine, nous allons utiliser le programme *deviceQuery*. Il est fourni par Nvidia dans chaque distribution de CUDA. Voici les différentes étapes pour le compiler et l'utiliser.

```
$ cd 0-deviceQuery
$ mkdir build && cd build
$ cmake ..
$ make
$ ./deviceQuery
```

Servez-vous du résultat de *deviceQuery* pour optimiser les performances de votre code dans les parties suivantes et répondre aux questions suivante :

1. Combien y-a-t-il d'unités de calcul (ALU) sur votre GPU ?
2. On considère qu'une ALU peut fournir le résultat de 2 opérations flotantes par coup d'horloge. Combien d'opérations par secondes en FLOPS (Floating-Point Operation per Second) peut effectuer votre GPU ?
3. Comparez la fréquence de fonctionnement du GPU avec celle du CPU (vous pouvez utiliser *lscpu* par exemple). Pourquoi est-il quand même intéressant d'utiliser un GPU ?

TP Partie 1 : Obfuscation

Intro

L'obfuscation est l'art de cacher les choses. Ici vous aurez deux images, qui prises séparement, ne représentent que du bruit. Pour en tirer leur secret, il va valoir les sommer, pixel par pixel.

Allez dans le dossier *1-Obfuscation* et visualisez les images dans *data*. Sous Linux, vous pouvez ouvrir une image depuis le terminal avec `eog $name_of_the_file.tif`.

Lisez l'intégralité de cette partie avant de démarrer !

Organisation du code

- CMakeLists.txt -- fichier de configuration de cmake
- data -- images d'entrée
    - big_frag_1.tif
    - big_frag_2.tif
    - small_frag_1.tif
    - small_frag_2.tif
- include
    - obfuscate.hpp -- fichier header contenant des macros utiles
- src
    - main.cpp -- structure de base du programme, vous n'avez pas besoin de faire des modifications ici

- obfuscate.cu -- **l'endroit où vous devez faire vos modifications**
- reference.cpp -- contient l'implémentation de référence sur CPU

Compiler the code

Avant tout de chose, allez dans le dossier *external/libtiff*, créez et allez dans le dossier *build* puis générez le Makefile avec `cmake ..` et compilez avec `make`.

Tout comme *deviceQuery*, ce projet utilise cmake. Une fois dans le dossier *1-Obfuscation*, créez et allez dans le dossier *build* puis générez le Makefile avec `cmake ..` et compilez avec `make`. À chaque nouveau changement du code, vous n'avez qu'a exécuter `make`. Lancer le programe avec `./obfuscation exercise_1`.

Premiers pas en CUDA

Dans obfuscate.cu, vous allez devoir implémenter la structure de base de tout programme CUDA:

1. Allouer la mémoire sur le GPU (device).
2. Transferer les données sur cet espace mémoire.
3. Lancer le kernel.
4. Copier les données du device sur l'host (CPU).
5. Libérer la mémoire du GPU.

Il y a déjà du code de présent pour vous aider. Dans cet exercice, vous devrez faire fonctionner votre kernel sur **un seul** block. Vous devez lui donner une taille appropriée pour vos données.

Ici, l'image est représentée comme un tableau continu d'octet non-signés (niveaux de gris). Vous devez calculer la somme des deux images et stocker le résultat dans la sortie.

Vous pouvez regarder l'implémentation de référence dans *reference.cpp*. Si vous avez besoin de détails supplémentaires, la documentation de CUDA est fournie avec le projet. Vous pouvez la décompresser comme ceci : `tar xzf documentation.tar.gz`.

Attention

Vous pouvez utiliser la fonction `checkCudaErrors`. Elle vous permettra de vérifier qu'une fonction s'est bien déroulée. Voici un example de comment l'utiliser : Avant :

```
cudaMalloc(&d\_red, sizeof(unsigned char) * numRows * numCols);
```

Après :

```
checkCudaErrors(cudaMalloc(&d\_red, sizeof(unsigned char) * numRows * numCols));
```

Écrire du code propre et sûr est toujours un peu plus long, mais c'est indispensable pour corriger ses erreurs rapidement. Si vous ne faites pas cette vérification et qu'il y a une erreur, tous les appels suivants ne vont rien faire, et se sera plus difficile de se rendre compte de pourquoi. Ecrire du code sûr vous permettra d'avoir l'information dès qu'il y a une erreur.

Enfin, n'oubliez de libérer la mémoire de ce que vous avez alloué.

Questions

1. Combien de temps prend votre implémentation CUDA ? Quelle est fraction de cette durée consacrée à la gestion mémoire ?
2. Comparez avec l'implémentation CPU.

Images RGB

Il y quelques changements par rapport à la partie I :

- l'image est en RGB, sur 32 bits. Utilisez les macros de obfuscate.hpp pour vous aider.
- l'image est trop large pour rentrer dans un seul block. Vous aller devoir diviser votre problème dans différents blocs.

Questions

Idem qu'en partie I:

1. Combien de temps prend votre implémentation CUDA ? Quelle est fraction de cette durée consacrée à la gestion mémoire ?
2. Comparez avec l'implémentation CPU.

Nouvelle question: 3. Faîtes différents essais de combinaisons taille des blocks / taille de la grille. Pour quelle découpe la temps d'exécution est-il le meilleur ? Quelle est votre justification ?

TP Partie 2 : flou et convolutions

Pendant ce TP vous allez devoir flouter une image. Pour ce faire, imaginez que nous avons un tableau réprésentant les valeurs des pixels. Pour chaque pixel de l'image, imaginez que l'on superpose un tableau de poids centré sur le pixel courant. Pour calculer la valeur du pixel flouté, on multiplie chaque paire de nombre qui se font face. C'est-à-dire, on multiplie chaque pixel avec le poids "au dessus". Pour finir, on somme tous les résultats des multiplications et on donne cette valeur au pixel en cours. On répète ce procédé pour tous les pixels de l'image.

Pour vous aidez à commencer, voici quelques notes :

Etape 0 (TODO1-9)

Transformation de la structure

Une image en couleur possède plusieurs canaux de couleurs (RGBA). Nous allons les séparer pour avoir des tableaux continus de valeur plutôt qu'entrelacé. Cela simplifiera le code.

Au lieu d'avoir RGBARGBARGBARGBARGBA... nous préférons avoir 3 tableaux distincts (on ignore le canal alpha):

1. RRRRRRRR...
2. GGGGGGGG...
3. BBBBBBBB...

La structure initiale se nomme Array of Structures (AoS) tandis que la seconde est Structure of Arrays (SoA).

En guise d'échauffement, vous allez devoir écrire un kernel qui effectue cette séparation. Ensuite, le coeur du problème est le flou (ou convolution). Un kernel qui recombine les canaux est déjà fourni.

Convolutions

Vous devez compléter le kernel `gaussian_blur` pour effectuer le floutage de inputChannel en utilisant le tableau de poids puis stocker le résultat dans outputChannel.

Voici un exemple de comment calculer le flou en utilisant une moyenne pondérée, pour un seul pixel de l'image :

Tableau des coefficients :

```
0.0 0.2 0.0
0.2 0.2 0.2
0.0 0.2 0.0
```

Image (notez qu'on aligne le tableau des coefficients au centre de la "boîte") :

```
1 2   5   2 0 3
   --- --- ---
3 |2   5   1| 6 0      0.0*2 + 0.2*5 + 0.0*1 +
4 |3   6   2| 1 4 -> 0.2*3 + 0.2*6 + 0.2*2 + -> 3.2
0 |4   0   3| 4 2       0.0*4 + 0.2*0 + 0.0*3
   --- --- ---
9 6   5   0 3 9
      (1)                    (2)              (3)
```

Par où commencer

Comme auparavant, vous aurez à lier chaque thread à un pixel de l'image. Ensuite, chaque thread peut faire les étapes 2 et 3 du diagramme ci-dessus indépendamment des autres.

Notez que le tableau des coefficients est un carré, sa hauteur est égale à sa largeur. On nomme le tableau des coefficients `filter` et sa largeur `filterWidth`.

Questions :

1. Une fois que vous avez vérifié que votre programme fonctionne sur la petite image `cinque_terre_small.tiff`, donnez le temps de calcul sur la grande: `cinque_terre_large.tiff`.
2. Encore une fois, essayez de trouver une taille de block / grille qui maximise la performance.

Etape 1 (TODO10-14)

Pour cette nouvelle étape, il faut mettre en place une optimisation sur notre code. Vous pouvez repartir de ce que vous avez déjà en étape 0.

Le filtre de convolution va maintenant être directement stocké dans la mémoire constant plutôt qu'en mémoire globale.

La variable a déjà été déclarée `filter_constant` mais il faut transférer les données sur cet espace grâce à cudaMemcpyToSymbol. Vous pouvez utiliser la documentation de CUDA pour vous aider.

Questions :

3. Quel est le speedup obtenu avec cette amélioration sur la grande image ? ((Speedup = Ancien_tps_de_calcul - Nouveau_tps_de_calcul) / Nouveau_tps_de_calcul)

Etape 2 (TODO15-19)

Encore une fois, vous pouvez repartir du travail fait dans les étapes précédantes.

Maintenant, on veut chercher a réutiliser les valeurs de l'image lues dans le bloc. Pour cela, le bloc va tout d'abord écrire dans la mémoire partagée ce dont il aura besoin.

Attention Il faut penser à mettre une barrière de synchronization une fois le transfert terminer. Voir *__syncthread()* dans la documentation.

Questions :

4. Pour une convolution avec un filtre de largeur et hauteur 5, combien de fois chaque pixel de l'image doit-il être lu ?
5. Quel est le speedup par rapport aux étapes 0 et 1 ?