Cycle Master, 3A, Option Intelligence Artificielle - GPU for Deep Learning - IA307

Architecture et Programmation des GPUs

Travaux Pratique

Introduction à CUDA, Traitement du Signal et des Images

Elisabeth Brunet et Tamy Boubekeur

Travail personnel. Binômes non autorisés.

Base de code

L'archive pour ce TP est disponible ici :

https://perso.telecom-paristech.fr/boubek/ens/GPU/res/GPU_TP_SIGNAL-IMAGE.zip

Télécharger et décompresser l'archive.

Dans le répertoire GPU_TP_SIGNAL_IMAGE se trouvent divers fichiers nécessaires au TP. Tous les programmes développés au cours du TP seront des programmes en ligne de commande, écrits en CUDA C, sans interface graphique. Pour chaque exercice, un unique fichier source exercice.cu sera implémenté et compilé à l'aide de nvcc :

```
nvcc –lm –o exercice exercice.cu
```

L'exécution de ces programmes devra être paramétrée suivant la syntaxe :

```
login@machin: ./mon-prog [arg1] [arg2] [...] [fichier-entrée] [fichier-sortie]
```

Les sorties numériques pourront s'effectuer sur la sortie standard.
Dans chaque cas, on pourra commencer par implémenter une version CPU du programme, avant de créer une version GPU. Notez qu'une part du code est toujours commune (chargement, écriture) et que l'on se restreint à un unique fichier .cu par exercice. Utilisez une simple macro pour compiler une version ou l'autre.

Exercice.cu

```
#define GPU_COMPUTE // A commenter pour compiler la version CPU
…
#ifdef GPU_COMPUTE
…
// Code parallèle en CUDA C
…
#else
…
// Code séquentiel en C
…
#endif
```

L'utilisation de Makefile est également possible (voir archive).

Debugging

Penser à bien vérifier les messages d'erreurs après les appels CUDA.

```
printf("%s\n", cudaGetErrorString( cudaGetLastError() ) );
```

Remise du travail

Chaque étudiant devra remettre, dans un délai de 7 jours à compter de la dernière séance de TP, une unique archive IA307_NOM_PRENOM.zip contenant :

- un rapport au format PDF décrivant le travail effectué et les éléments techniques intéressants développés (autour de 2 à 4 pages)
- pour chaque exercice réalisé : un fichier <exercice>.cu unique, chaque fichier devant pouvoir être compilé, sur les machines de l'Ecole, avec la commande :
  nvcc –lm -o <exercice> <exercice>.cu

Un lien vers l'archive doit être remis à l'adresse suivant : tamy.boubekeur@telecom-paristech.fr
Le sujet du mail doit être : [IA307][TP]<nom> <prénom>. Ne pas inclure l'archive dans le mail, seulement le lien.

I. SAXPY : Hello World CUDA

On cherche à implémenter l'application linéaire $f(x) = ax + b$ sur un grand vecteur de données.
a/ Créer un fichier saxpy.cu. Ce programme devra:

- allouer un tableau X de taille N sur CPU
- le remplir de valeurs (par exemple $X[i] = i$)
- le copier dans la mémoire GPU globale
- exécuter un noyau saxpy remplaçant chaque valeur $x \in X$ par $ax + b$
- copier le résultat du GPU vers le CPU
- afficher les 10 premiers et 10 derniers résultats (pour vérification)
- prendre en argument les valeurs a et b.

b/ Ajouter N aux paramètres spécifiables du programme, ainsi que K le nombre de blocs composant la grille de calcul associée au noyau. Tester plusieurs découpages «nombre de blocs» / «nombre de threads par bloc», pour différente valeur de N et observer la performance atteinte. On pourra utiliser la commande time dans le *shell* pour mesurer les temps d'exécution.

II. Traitement du signal

On souhaite maintenant travailler sur un signal échantillonné dans un vecteur S de taille N. On souhaite en particulier appliquer divers filtres à ce signal. On s'intéresse aux filtres ayant la forme :

$$S'(x) = \frac{\sum_{y \in [x-s, x+s]} \omega(|x-y|) S(y)}{\sum_{y \in [x-s, x+s]} \omega(|x-y|)}$$

avec ω le noyau de filtrage et s la taille du support du filtre. Un exemple de filtre simple est le filtre boite (équivalent à une simple moyenne):

$$\omega_s^B(t) = 1 \text{ si } t \leq s, \ 0 \text{ sinon.}$$

Un filtrage basé sur un noyau gaussien évite quant à lui certains artefacts du filtre boite :

$$\omega_s^G = \frac{1}{s\sqrt{2\pi}} e^{-\frac{t^2}{2s^2}}$$

On pourra utiliser GNUPlot pour visualiser ces différents noyaux.

II.a Créer un programme signal.cu initialisant un tableau S avec la somme de 2 sinusoïdes de fréquences différentes. Choisir $N >= 1\,000\,000$.

II.b Filtre boite : implémenter un noyau CUDA appliquant sur le signal un filtre boite de taille s paramétrable (passé en argument du programme). Considérer une simple symétrie aux bords.

II.c Exporter le début du signal original et le début du signal filtré (200 premières valeurs) dans un fichier ASCII signal.data (2 colonnes). Visualiser les deux suites avec GNUplot (sous forme de courbes). La sortie EPS ou PDF couleur sera à intégrer au rapport (script GNUplot fourni dans l'archive).

II.d Filtre gaussien : idem mais avec un noyau gaussien.

II.e Exporter le début du signal original, le début du signal filtré «boite» et le début du signal filtré «gaussien» (200 premières valeurs). Visualiser les 3 suites avec GNUplot (sous forme de courbes). Sortie EPS couleur à intégrer au rapport (attention, il faut modifier le script GNUplot).

II.f Implémenter, si ce n'est pas déjà fait, une version CPU, toujours dans signal.cu.

II.g Remplir un tableau à 5 lignes et 30 colonnes avec le temps d'exécution mesuré pour les deux filtres en mode CPU et en mode GPU. La k^{ième} colonne indiquera le temps d'exécution du programme pour une noyau avec support de taille 20k :

k	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**…**
Boite CPU (ms)									
Boite GPU (ms)									
Gaussien CPU (ms)									
Gaussien GPU (ms)									

Intégrer le tableau au rapport. *Note : votre programme peut bien entendu générer le tableau directement.*

III. Traitement d'image

On souhaite maintenant traiter des images, à savoir des signaux en 2 dimensions, à l'aide d'un programme image.cu.

III.a Le format PGM encode une image en ASCII et en niveaux de gris. Voici sa structure:

```
P2
512 512
255
15
```

56
48
...
Ici ce fichier représente une image de résolution 512x512, encodée sur 256 niveaux de gris sur [0, 255]. Les valeurs des pixels sont ensuite listées. On pourra s'aider du logiciel GIMP pour visualiser ces images ou en convertir de nouvelles. Plusieurs versions d'une même image sont fournies dans l'archive du TP.

Ecrire une fonction de chargement et une fonction d'écriture d'images PGM dans image.cu. La tester à l'aide des images fournies.

III.b Le filtrage d'image est très similaire au filtrage de signal 1D. La seule différence est qu'il faut considérer «x» comme une coordonnée à 2 dimensions :

$$I^F(x) = \frac{\sum_{y\in[x-s,x+s]^2} \omega_s(\|x-y\|)I(y)}{\sum_{y\in[x-s,x+s]^2} \omega_s(\|x-y\|)}$$

Note : dans un souci de généralité, on considère la distance euclidienne entre x et y. D'autres mesures de distances sont néanmoins possibles.

Implémenter un noyau CUDA appliquant un filtre boite de taille de support arbitraire (passée en argument du programme) sur l'image. Générer plusieurs images à partir des différentes images de l'archive du TP en faisant varier la taille du support. Tout comme pour le signal 1D, on prendra garde au traitement des bords (symétrie). Intégrer quelques exemples de filtrages obtenus au rapport.

III.c Idem mais avec un filtre Gaussien cette fois. Comparer les images obtenues avec les deux filtres.

III.d Le filtrage bilatéral est un filtre qui, contrairement au filtre gaussien, préserve les détails importants de l'image (forts gradients). Il peut être assimilé à une diffusion anisotrope et se révèle surtout intéressant pour les tailles de noyau élevées (> 25 pixels). D'où l'utilité d'une implémentation CUDA ! Le principe ici est de pondérer la contribution d'un pixel du support non seulement par sa distance au pixel filtré mais également par sa valeur :

$$I^{BL}(x) = \frac{\sum_{y\in[x-s,x+s]^2} \omega_s(\|x-y\|)\omega_r(|I(x)-I(y)|)I(y)}{\sum_{y\in[x-s,x+s]^2} \omega_s(\|x-y\|)\omega_r(|I(x)-I(y)|)}$$

avec r la taille du support dans l'espace des valeurs de l'image (i.e. couleur des pixels).

On pourra se référer au cours dédié au filtrage bilatéral pour plus de détails ici : http://people.csail.mit.edu/sparis/bf_course/
Contrairement au filtrage gaussien, il faut cette fois-ci 2 paramètres en argument du programme : la taille du support spatial s (distance entre pixels) et la taille du support *valeur* r (différence de niveaux de gris entre pixels).

III.e Modifier le programme pour traiter des images couleurs (format PPM : http://netpbm.sourceforge.net/doc/ppm.html). On considérera la distance 3D dans l'espace RGB lors de la pondération dans l'espace des valeurs.

III.f On souhaite enfin ajouter à notre programme la capacité à effectuer des filtrages non-linéaires *morphologiques*, sous la forme d'érosions simples (élément structurant carré de taille s) :

$$I^E(x) = \inf_{y \in [x-s, x+s]^2} I(y)$$

et de dilatations simples :

$$I^D(x) = \sup_{y \in [x-s, x+s]^2} I(y)$$

Ajouter ces filtres à votre programme et modifier son interface (arguments passés) de manière à pouvoir facilement le lancer avec un filtre donné (gaussien, bilatéral ou morphologique) et ses paramètres associés.

Bonus. Faîtes évoluer votre programme image.cu de manière à pouvoir lancer une suite de filtrages sur une image, en effectuant un seul appel au programme, et sans avoir à rapatrier les images intermédiaires en mémoire centrale (CPU). Par exemple, l'appel :

login@machine: ./image –bl 0.1 0.2 –d –bl 0.2 0.3 –e input.ppm output.ppm

produira la séquence :

1. filtrage bilatéral avec s=0.1 et r=0.2
2. dilatation de l'image obtenue
3. filtrage bilatéral avec s=0.2 et r=0.3 de l'image obtenue
4. érosion de l'image obtenue