Introduction à CUDA, Traitement du Signal et des Images

Jamila Rejeb

21 février 2019

Le but de ce TP est de s'initier aux notions d'architecture et de programmation des GPUs. Le code est fourni sur github dans le répo suivant :
`https://github.com/jrejeb94/Introduction_to_CUDA`

1 SAXPY : Hello World CUDA

Le première exercice est une introduction à CUDA qui nous a permis de comprendre comment coder un kernel CUDA et comment allouer de la mémoire sur le device. J'ai pu aussi voir l'importance du choix des paramètres tel que le nombre de thread par block et le nombre de block ...Comme mentionné dans l'énoncé du tp, le programme a été nommé *saxpy.cu*. On y trouve deux kernels :

- saxpy_cpu : excécuté par le cpu
- saxpy_gpu : excécuté par le gpu

Pour appeler saxpy, il faudra suivre cet ordre :

```
./saxpy [N] [a] [b] [CPU or GPU?] [nThreadsPerBlocks]
```

Le dernier argument (nThreadsPerBlocks) reste optionnel (si non indique une valeur par défault égale à 512). Voici quelques exemples d'appel :

```
c51−02% ./saxpy 100000 2 5 CPU
```

```
c51−02% ./saxpy 200000000 2 5 GPU 512
```

CPU ou GPU ? taille du tableau	1000	10000000	200000000
CPU	0 (ms)	50 (ms)	0.9 (s)
GPU, nThreadsPerBlocks = 4	60 (ms)	90 (ms)	0.37 (s)
GPU, nThreadsPerBlocks = 256	60 (ms)	80 (ms)	0.35 (s)
GPU, nThreadsPerBlocks = 512	50 (ms)	80 (ms)	0.33 (s)
GPU, nThreadsPerBlocks = 1024	40 (ms)	80 (ms)	0.38 (s)

A partir de ces valeurs, on conclut que pour un tableau de petite taille, il n'est pas utile d'utiliser une architecture GPU (les thread vont plutôt se battre entre elles pour chaque case et beaucoup de blocks ne vont pas être utilisés). Plus on augmente la taille de la structure sur laquelle on va faire les calculs, mieux est le résultat. On remarque à titre d'exemple que pour une taille de 2.10^8, le kernel du gpu est presque trois fois plus rapide que celui du cpu. Ainsi on peut conclure que pour faire tourner un code sur le GPU il faudra avoir beaucoup de donnée à modifier.

2 Traitement du signal

Dans cette partie, on va essayer d s'approfondir avec un execice de filtrage de signal 1D. On commence par le générer. Notre signal a cette forme là :

$$S = A_1 sin(2\pi f_1 t) + A_2 sin(2\pi f_2 t), \text{ avec } A_1{=}50,\ A_2{=}5,\ f_1{=}0.005,\ f_2 = 0.05$$

L'exécution de ce programme devra être paramétrée suivant la syntaxe suivante :

```
./signal [N] [s] [w] [CPU or GPU?] [output_file]
```

avec :

- N : taille du signal
- s : taille du filtre appliqué
- w : type de filtrage à utiliser. A choisir entre "g" pour gaussien et "b" pour boite
- CPU or GPU ? : pour choisir entre utilisé le noyau CUDA (donc mentionner "GPU" comme valeur) ou pas (donc "CPU")
- output_file : le nom du fichier dans lequel les 200 premières valeurs du signal filté vont être enregistrées.

Exemple d'éxécution :

```
c51−02% ./signal 10000000 60 b CPU signal.data
Elapsed: 13.140000 seconds
Created signal.data ....%
c51−02% ./signal 10000000 20 g GPU signal_gpu.data
Elapsed: 3.110496 seconds
Created signal_gpu.data....%
```

Etant donné qu'il n'est pas possible d'utiliser des fonctions telle que la fonction exponentielle ou la racine carrée dans les noyau CUDA, j'ai décidé de calculer les valeurs des filtres boite et gaussien an amont et ensuite de les utiliser pour paralléliser la convolution pour chaque point du signal. On a donc deux fonctions $\_\_host\_\_$ qui sont $kernel\_gauss$ et $kernel\_boite$ qui vont être appelé par le cpu pour calculer les valeurs des filtres, une fonction $\_\_global\_\_$ qui va être appelée depuis le main mais exécutée sur le GPU.

Voici les réultats obtenus pour N = 10000000 et s= 20k (avec k $\in \{1, 2, 3, 4, 5, 50\}$)

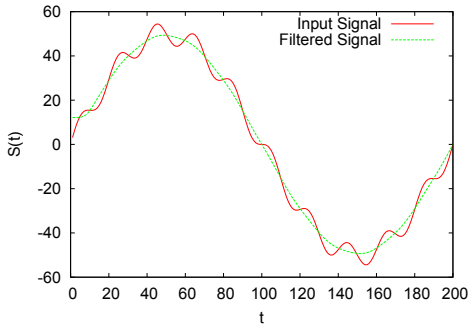

FIGURE 1 – La courbe des 200 premières valeurs du signal initial et de celui filtré (ici filtrage gaussien utilisé).

	20	40	60	80	100	1000
Boite CPU (s)	4.4	8.86	13.1	17.35	21.74	214
Boite GPU (s)	3	3.1	3.11	3.1	3.11	3.11
Gaussien CPU (s)	4.48	8.84	13.1	17.5	21.8	214.08
Gaussien GPU (s)	3.1	3	3.11	3.1	3.11	3.11

On remarque ainsi l'utilité du noyau CUDA quand le noyau de filtrage est grand : l'architecture GPU nous a permis de gagner du temps en excécutant le même code dix fois plus rapidement que le noyau CPU (pour un filtre de taille=1000). On note aussi que pour des tableau de grande taille (à l'odre de 10^{10}) les 200 dernières valeurs ne se sont pas bien calculées. On pourra dire que c'est dû au fait que tous les blocks du GPU sont occupés avec un nombre maximal de thread.

3 Traitement d'image

```
./image [s] [w] [input_file] [output_file] [r]
```

- s : taille du filtre appliqué
- w : type de filtrage à utiliser. A choisir entre "g" pour gaussien, "b" pour boite, "bl" pour bilatéral, "d" pour dilatation et "e" pour erosions.
- input_file : le nom de l'image à prendre en entrée pour filtrer
- output_file : le nom de l'image produite
- r : taille du support valeur (à indiqué si le filtre choisi est un filtre bilatéral)

```
./image 20 g image_256x256.pgm image_256x256_out.pgm
./image 40 bl image_256x256.pgm image_256x256_bl.pgm 50
```

Dans la première exécution, on a fait un filtrage gaussien de support égal à 20 sur l'image $image\_256x256.pgm$ fournie en entrée pour avoir en sortie une image qu'on a nommée $image\_256x256\_out.pgm$. Dans la deuxième exécution on a choisi d'appliquer un filtre bilatéral avec un support valeur r = 50.

Les valeurs du filtre bilatéral ont était aussi précalculées. Etant donné qu'on a 256 niveaux de gris possibles, ainsi |I(x) - I(y)| peut prendre toutes les valeurs entre 0 et 255. La valeur du paramètre r est alors utilisée comme écartype pour le calcul de la gaussienne. Ci-dessous quelques sorties du programme.

(a) (b) (c)

FIGURE 2 – Exemples des images : (a) l'image initiale (1024x1024) (b) avec un filtrage gaussien appliqué (c) filtrage boite. La talle des deux filtres utilisée = 25.

Conclusion

Utiliser l'architecture GPU nous permet de gagner de temps dans le cas où la taille des données est assez importante. Il faudra aussi penser à comparer le nombre de thread par bloc que prend le programme pour faire un choix optimal. La plus grande difficulté pour moi réside dans la façon avec lequel on doit organiser le code afin de pouvoir tirer profit de l'architecture GPU étant donné qu'on n'a pas toutes les fonctions du cpu.