

TP CUDA – ESIEE

Nicolas GAC – Julien Demouth -- 21 Septembre 2012

Contexte

Le raytracing (lancer de rayons) est une technique de rendu graphique qui permet de produire des images de haute qualité visuelle. Le principe de l'algorithme est de simuler le lancer de rayons depuis chaque pixel de l'image à créer. A chaque pixel, le lancer de rayons permet de déterminer si un objet est visible et de connaître, le cas échéant, sa couleur.

L'efficacité de l'algorithme de raytracing repose, entre autres, sur celle de l'algorithme d'intersection des rayons avec les objets. Pour simplifier, plus le nombre d'intersections testées par l'algorithme est grand, plus le temps de calcul et la difficulté de la tâche augmentent.

Afin de réduire le nombre d'intersections testées de nombreuses techniques ont été mises au point. Parmi celles-ci, nous nous intéressons aux structures de données qui permettent de décrire une simplification géométrique de la scène à rendre. L'idée derrière ces structures est assez simple : si on enferme des objets dans une boîte, l'absence d'intersection entre la boîte et un rayon garantit l'absence d'intersection entre ce rayon et les objets contenus dans la boîte. On évite ainsi de tester l'intersection du rayon avec chaque objet de la boîte.

Il existe de nombreuses structures géométriques de « boîtes englobantes » : Octrees, Kd-trees, BVH, etc. Dans ce TP, nous allons nous intéresser au tri de nombres entiers. C'est la primitive de base qui permet de construire un BVH (Bounding Volume Hierarchy) sur le GPU.

Pour aller plus loin

Des implémentations des algorithmes de construction d'Octrees, Kd-trees ou BVH pour le GPU sont disponibles dans le projet Open Source suivant : <http://code.google.com/p/slash-sandbox/>. Le code est développé par Jacopo Pantaleoni.

1^{ère} partie : La réduction

L'algorithme de réduction permet de calculer la somme des éléments d'un tableau. La version séquentielle de cet algorithme est :

```
int sum = 0;
for( int i = 0 ; i < n ; ++i )
    sum += a[i];
```

Questions

- Comment peut-on implémenter cet algorithme sur le GPU ? Remplir le « code à trous » à partir des slides Powerpoint additionnelles.

- (Optionnelle) Comment peut-on améliorer la performance de notre implémentation ?

2^{ème} partie : Le scan

L'algorithme de scan permet de calculer, pour chaque élément d'un tableau, la somme des éléments qui le précèdent. La version séquentielle de cet algorithme est :

```
int sum = 0;
for( int i = 0 ; i < n ; ++i )
{
    b[i] = sum;
    sum += a[i];
}
```

Questions

- Comment peut-on implémenter cet algorithme sur le GPU ? Remplir le « code à trous » à partir des slides Powerpoint additionnelles.
- (Optionnelle) Comment peut-on améliorer la performance de notre implémentation ?

3^{ème} partie : Le tri

Les algorithmes de tri « traditionnels » tels que Selection Sort, Bubble Sort, Quick Sort ou Merge Sort, sont basés sur des comparaisons. La complexité en temps dans le pire des cas de ces algorithmes varie entre $O(n \log n)$ et $O(n^2)$. De plus nous connaissons une borne inférieure de $\Omega(n \log n)$. Pour *certains problèmes*, il existe toutefois des algorithmes de tri qui possèdent une complexité théorique inférieure. C'est notamment le cas du tri par base ou Radix Sort qui a une complexité $O(n)$ pour des entiers.

Le principe du Radix Sort est de considérer les bits des entiers par blocs (prenons 8, par exemple) et d'effectuer un tri sur ces bits. Une fois les nombres triés suivant ces bits, l'algorithme passe à un autre bloc de bits.

```
for( int bit = 0 ; bit < 32 ; bit += 8 )
    sort the array using bits [bit...bit+8)
```

Afin d'effectuer le tri des nombres sans utiliser de comparaison, l'algorithme tire parti du fait que le nombre possible d'entiers codés sur 8 bits n'est que de 256. Grâce à cela, il suffit d'allouer un tableau de 256 éléments, initialisés à 0, et de compter le nombre d'occurrence de chaque combinaison de bits. Une fois ces nombres connus, il est simple de réordonner le tableau de départ.

Questions

- Remplir le « code à trous » pour implémenter Radix Sort sur le CPU en utilisant C++. Vous pouvez modifier le nombre de bits considérer à chaque étape (il existe des implémentations optimisées de Radix Sort qui utilisent différents nombres de bits par passe).
- Comment peut-on implémenter cet algorithme sur le GPU ? Remplir le « code à trous » à partir des slides Powerpoint additionnelles.

- (Optionnelle) Comment peut-on améliorer la performance de notre implémentation ?

Aller plus loin

Le code d'accompagnement ne contient pas le code d'une version optimisée de Sort. La version de la bibliothèque Thrust est en revanche optimisée. Il existe toutefois des versions encore plus rapide que `thrust::sort`. L'une de ces implémentations est disponible à l'adresse :

<http://code.google.com/p/back40computing/wiki/RadixSorting> (voir la branche FastSm20). Cette version est développée par Duane Merrill et Sean Baxter.

La page de Sean Baxter peut servir de bonne référence pour approfondir ses connaissances sur ces algorithmes : <http://www.moderngpu.com/>