

Simulation de particules sur architectures parallèles - version OpenCL

Alexis Couture, Marc Sarraute

4 mai 2015

1 Bilan

Pour ce second livrable, nous avons implémenté une étape du tri par boîte, la fonction scan, sous forme de noyau OpenCL (le reste du tri étant déjà implémenté).

On a réussi à implémenter un scan (ou prefix sum) qui s'exécute de manière parallèle. On a en entrée un tableau `box_buffer` contenant le nombre d'atomes par boîtes et on écrit le résultat dans `calc_offset_buffer`. Notre fonction se déroule de la manière suivante :

- On divise `box_buffer` en le stockant par parties dans la mémoire locale de chaque workgroup, dans un tableau `temp`
- Chaque workgroup réalise un prefix sum sur sa partie de `box_buffer`, toujours sur `temp`
- Chaque workgroup stocke la dernière valeur de son prefix sum dans `box_buffer` (le workgroup `i` stocke son résultat dans la case `i`)
- Chaque workgroup fait un prefix sum sur `box_buffer`, et le stocke en mémoire locale dans le tableau `local_sumz`
- Chaque workgroup ajoute à son prefix sum, la somme des prefix sum obtenus par les workgroups précédents (traitant les parties antérieures de `box_buffer`)
- On écrit les valeurs stockées dans `temp` dans `calc_offset_buffer`

Notre noyau utilise un thread par boîte et partage le travail entre plusieurs workgroups. On utilise le tiling pour réduire, mais sans éliminer, le problème de dépendance inhérent à l'opération scan, où chaque thread a besoin du résultat du thread travaillant sur la case précédente. Au final, nous obtenons de moins bonnes performances qu'avec le tri par Z, signe que notre noyau est très perfectible.

2 Limitations

Tout d'abord, il n'existe pas en OpenCL de moyen propre pour synchroniser les workgroups entre eux car ceux-ci doivent travailler de manière indépendante. Pourtant, chaque workgroup a besoin des résultats des workgroups précédent pour "terminer" son calcul. Pour traiter ce problème, chaque workgroup stocke son résultat en mémoire globale dans `box_buffer`, à la case d'indice de son `group_id`, et récupère le résultat des autres workgroups sans vérifier si ceux-ci ont fini leur calcul.

Ensuite, notre implémentation a une complexité linéaire $n + m$ (où n est le nombre de boîtes et m le nombre de workgroups) et utilise n threads (un thread par boîte), ce qui n'est pas efficient. En conséquence, nous n'obtenons pas de performances supérieures à celles du tri par Z version OpenCL, ce qui n'est pas satisfaisant étant donné que le tri par boîte est théoriquement plus rapide. Une idée d'amélioration serait

de suivre l'implémentation décrite par M. Harris¹, qui utilise moins de threads et possède une complexité logarithmique.

Enfin, notre kernel ne peut s'exécuter sur les plus grosses configurations (biochoc1 par exemple), en raison d'un grand nombre de boîtes. Là encore, on peut lier ce problème au grand nombre de threads utilisés mais aussi éventuellement à la quantité de mémoire locale utilisée, qui grandit avec le nombre de workgroups.

3 Tests

Nous avons effectués des mesures de performances sur 2 machines différentes :

- kira, doté d'un CPU de 12 coeurs hyperthreadés et d'un GPU NVIDIA Quadro K2000 (2 Go de RAM)
- tesla, doté d'un CPU de 20 coeurs hyperthreadés et d'un GPU NVIDIA Tesla K20Xm (6 Go de RAM)

Pour résumer, la machine tesla dispose d'un bien meilleur GPU que kira, ce qui va permettre de déterminer si notre kernel tire parti du matériel ou non. On compare également 4 versions données ou implémentées :

- La version séquentielle, fournie au début du projet (uniquement sur kira)
- La version OpenMP avec un tri par Z, implémentée lors de la première partie du projet, avec un nombre de threads dynamique (meilleurs résultats)
- La version OpenCL avec un tri par Z, fournie pour la seconde partie du projet
- ... et la version OpenCL avec un tri par boîte, dont on évalue les performances

En comparant les temps d'exécution sur différentes configurations, présentés sur les figures ?? et ??, on s'aperçoit que notre kernel gagne significativement en rapidité sur le meilleur support, tesla. Néanmoins, sur les 2 machines, la version OpenCL du tri par boîtes est toujours plus lente que la version OpenMP du tri par Z, signe que notre algorithme n'est vraiment pas optimal pour GPU.

En étudiant les résultats par configuration, on peut remarquer le résultat constant obtenu sur choc1, choc2, choc3 et choc4, qui s'explique par un nombre de boîtes identique entre ces 4 configurations. Comme nous l'avons dit précédemment, notre algorithme a une complexité linéaire liée au nombre de boîtes, ce qui explique ce résultat. Aussi, notre kernel ne peut pas s'exécuter sur la configuration biochoc1, qui contient bien plus de boîtes (et d'atomes), ce qui est assez regrettable.

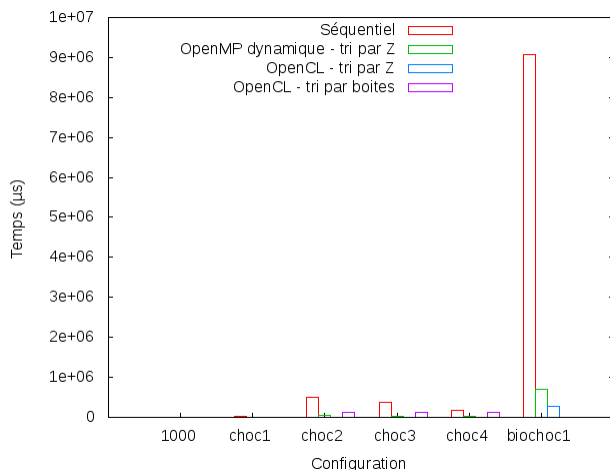


FIGURE 1 – Comparaison entre les différentes versions sur plusieurs configurations - kira

1. http://developer.download.nvidia.com/compute/cuda/1.1-Beta/x86_website/projects/scan/doc/scan.pdf

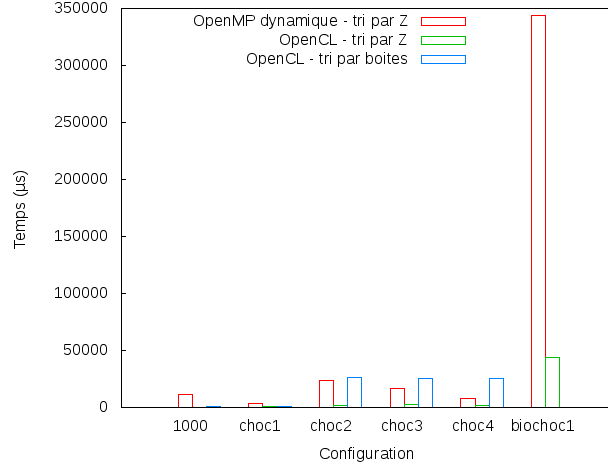


FIGURE 2 – Comparaison entre les différentes versions sur plusieurs configurations - tesla

Nous avons ensuite étudié les temps d'exécution de notre tri par boîte et du tri par Z version OpenCL fourni, en fonction du nombre d'atomes strictement (résultats figure ??). On y observe cette fois que notre tri par boîte est bien plus rapide, avec une courbe qui augmente lentement par rapport au tri par Z. On peut expliquer ce changement de rapport de performances par le faible nombre de boîtes, qui induit une exécution plus rapide de notre kernel scan, et par le tri par Z, qui est théoriquement bien moins efficient que le tri par boîtes.

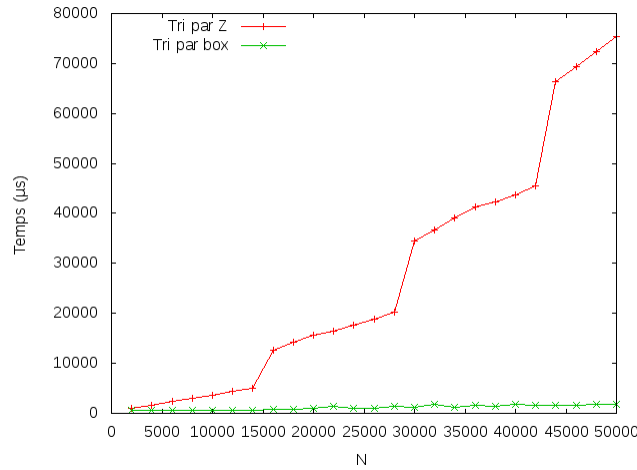


FIGURE 3 – Temps d'exécution du tri par boîtes et du tri par Z versions OpenCL, en fonction du nombre d'atomes, sur tesla

Au final, ces mesures montrent que notre kernel est très perfectible, au vue des performances obtenues par le tri par Z avec OpenCL.