Simulation de particules sur architectures parallèles - version OpenCL

Alexis Couture, Marc Sarraute

3 mai 2015

1 Bilan

Pour ce second délivrable, nous avons implémenté une étape du tri par boite, 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 boites 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 antèrieures de box_buffer)
- On écrit les valeurs stockées dans temp dans calc_offset_buffer

Notre noyau utilise un thread par boite 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 assez 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 boites et m le nombre de workgroups) et utilise n threads (un thread par boite), ce qui n'est pas efficient. En conséquence, nous n'obtenons pas de performances supèrieures à celles du tri par Z version OpenCL, ce qui n'est pas satisfaisant étant donné que le tri par boite est théoriquement plus rapide. Une idée d'amélioration serait de

suivre l'implémentation décrite dans ce papier (LE PDF DE M.HARRIS), qui utilise moins de threads et possède une complexité logarithmique.

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 materiel ou non. On compare également 4 versions données ou implémentées :
- La version séquentielle, fournie au début du projet
- 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 boite, dont on évalue les performances

En comparant les temps d'exécution sur différentes configurations, présentés sur les figures 1 et 2, 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 boites 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 boites identique entre ces 4 configurations. Comme nous l'avons dit précédemment, notre algorithme a une complexité linéaire liée au nombre de boites, ce qui explique ce résultat.

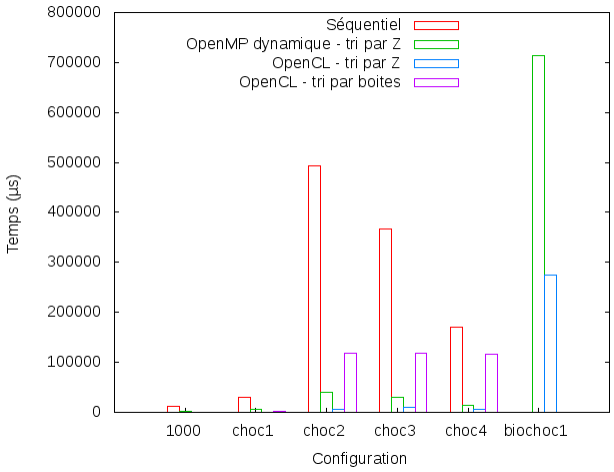

FIGURE 1 – Comparaison entre les différentes versions sur plusieurs configurations - kira

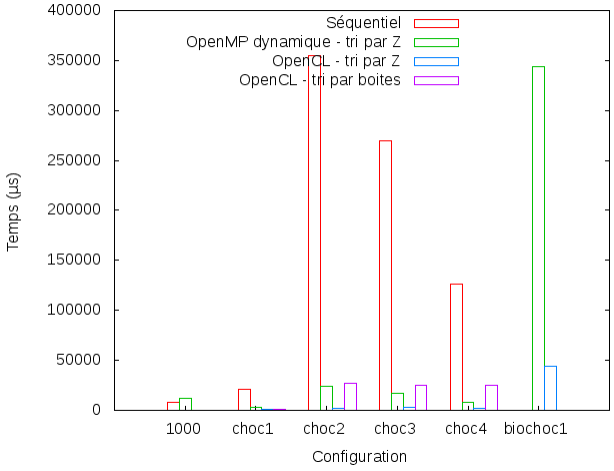

FIGURE 2 – Comparaison entre les différentes versions sur plusieurs configurations - tesla