Simulation de particules sur architectures parallèles

Il s'agit de concevoir une application de simulation de particules dans un domaine en trois dimensions, en calculant des interactions à courte distance entre particules. Le déroulement de la simulation pourra être visualisé en « temps réel » grâce à un rendu OpenGL des particules. Une version séquentielle naïve du code vous est fournie, ainsi que la partie visualisation, de façon à ce que vous puissiez uniquement vous focaliser sur l'accélération des calculs en parallèle.

Ce projet est à faire en binôme et est composée de deux parties :
- Une partie OpenMP ;
- Une partie OpenCL sur accélérateurs. Un complément d'information vous sera transmis en temps utile sur le travail à réaliser.

1 Premiers pas

1.1 On essaye tout de suite !

Copiez le répertoire `~rnamyst/etudiants/pmg/Projet` sur votre compte. Dans le répertoire `fichiers/`, il vous faut d'abord générer le Makefile automatique à l'aide de `cmake` :

```
mkdir build
(cd build ; cmake ..)
```

Ensuite vous pouvez compiler :

```
(cd build ; make)
```

Si tout va bien, le binaire `bin/atoms` est construit. Affichez l'aide en ligne en tapant `./bin/atoms -h`. Lancez une première exécution avec les options suivantes :

```
./bin/atoms -v -s 1 -o 1
```

Normalement, un ensemble d'une centaine d'atomes apparaît dans une fenêtre OpenGL. Vous pouvez alors :
- changer l'angle de vue à la souris (cliquer-déplacer) ;
- taper `>` (resp. `<`) pour zoomer (resp. dézoomer) ;
- taper `+` (resp. `-`) pour accélérer (resp. ralentir) la simulation ;
- taper `m` pour activer/désactiver le mouvement des atomes ;
- taper `q` ou la touche *Escape* pour quitter l'application.

1.2 Structure de la simulation

Le code source de la simulation est organisé au sein d'une bibliothèque rassemblant les fonctions de simulation et de visualisation des particules. Le fichier `main.c` du programme sert à analyser les options de la ligne de ocmmande, à éventuellement charger la configuration initiale depuis un fichier, et à lancer la boucle principale de la simulation. La bibliothèque est organisée de la façon suivante :

`sotl.c`	Point d'entrée principal de la bibliothèque, ce fichier rassemble les fonctions appelables depuis le programme principal. Il est également en charge de découvrir les accélérateurs disponibles (pour OpenCL) et d'initialiser la bibliothèque.
`atom.c`	Gestion des atomes.
`domain.c`	Gestion du domaine 3D contenant les atomes.
`seq.c`	Implémentation séquentielle de la simulation. C'est probablement le premier fichier à regarder, à modifier pour comprendre son fonctionnement, etc.
`openmp.c`	Fichier dans lequel vous implémenterez la version OpenMP de la simulation. Les fonctions de ce module seront appelées lorsque l'option `--omp` sera utilisée en ligne de commande (voir aide en ligne du programme).
`window.c`	Initialisation d'OpenGL et boucle principale de rafraichissement d'écran.
`vbo.c`	Gestion des points et des triangles destinés à l'affichage OpenGL. Normalement, vous n'aurez pas besoin de consulter/modifier ce module. On peut même très bien vivre sans l'avoir regardé...
`ocl.c`	Initialisation et gestion des différentes structures liées à OpenCL. Il est utile d'y jeter un oeil pour comprendre quels sont les *buffers* alloués sur la carte graphique pour les besoins de cette simulation.
`ocl_kernels.c`	Ensemble des « *wrappers* » permettant d'exécuter les noyaux OpenCL.
`physics.cl`	Code OpenCL des noyaux qui s'exécuteront sur la carte graphique.

Pour vous familiariser avec le cœur de l'application, regardez dans `sotl.c` la fonction `sotl_main_loop` et suivez les fonctions appelées dans `seq.c`.

1.3 Positions et coordonnées des atomes

Pour calculer le résultat des interactions entre atomes, on mémorise pour chaque atome sa position (x, y, z) et sa vitesse (dx, dy, dz). Pour simplifier, on considère que la vitesse d'un atome est calibrée de manière à ce qu'à chaque itération de la simulation, (dx, dy, dz) représente le vecteur qu'il faut ajouter à la position d'un atome pour obtenir sa position à l'itération suivante.

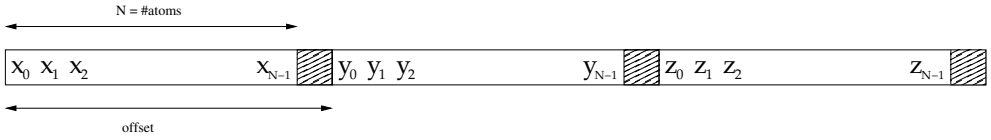

FIGURE 1 – Pour améliorer la performance des accès mémoire sur les accélérateurs, les tableaux `pos_buffer` et `speed_buffer` sont agencés de la manière illustrée ci-dessus : une première tranche contient les coordonnées x (dx pour `speed_buffer`), une seconde contient les y et la troisième contient les z. La taille totale de chaque tranche est agrandie de manière à correspondre à un multiple de 16 éléments. Le nombre d'atomes d'un ensemble `set` d'atomes est `set.natoms`. La taille totale d'une tranche est contenue dans `set.offset`.

L'application mémorise la position des atomes et leur vitesse dans deux tableaux distincts, respectivement nommés `pos` et `speed` dans la structure `atom_set` (définie dans `atom.h`).

1.4 Mouvement des particules

Regardez le code de la fonction `seq_move` (dans `seq.c`) : c'est celle-ci qui met à jour, à chaque itération, les positions de tous les atomes en fonction de leur vitesse. Notez qu'il s'agit d'une addition de vecteurs...

1.5 Visualisation

Pour les besoins de la visualisation, un *buffer* OpenGL nommé `vbo_buffer` (« *Vertex Buffer Object* ») contient les coordonnées de chaque point utilisé pour afficher un atome. Ce tableau contient une suite de triplets (x, y, z) (chaque coordonnée étant de type `float`). Les `vertices_per_atom` $\times 3$ floats forment donc les coordonnées du premier atome, etc [1].

1.6 Rebond sur les parois du domaine

Dans chaque fichier de configuration, en plus des coordonnées des atomes, sont définies les coordonnées de deux points `min_ext` et `max_ext` délimitant le parallélépipède rectangle contenant tous les atomes.
Afin de garantir que les atomes restent dans ce parallélépipède, nous allons les faire rebondir sur les parois à chaque fois qu'ils entrent en collision avec l'une d'elles.
Écrivez la fonction `seq_bounce` qui teste, pour chaque atome, la collision avec l'un des bords. Si le centre d'un atome franchit un bord, il faut inverser la composante vitesse qui est orthogonale à ce bord. Par exemple, pour tester le rebond sur le sol, il faut pour chaque atome comparer y et y_{min_ext} et, en cas de collision, multiplier la composante vitesse dy par -1.

1.7 Potentiel de Lennard Jones

De nombreux phénomènes physiques entrent en jeu lorsqu'il s'agit de modéliser les interactions entre atomes au sein d'un gaz, d'un solide ou d'un liquide (cf `http://fr.wikipedia.org/wiki/Potentiel_interatomique`).

1. Cette façon de stocker les coordonnées n'est pas forcément idéale pour les noyaux OpenCL que nous écrirons ultérieurement, mais elle est imposée par OpenGL.

Nous nous intéresserons ici au potentiel de Lennard-Jones, qui capture à la fois les phénomènes d'attractions entre atomes lorsqu'ils sont distants, et les phénomènes de répulsion lorsqu'ils sont trop proches (effets quantiques). L'intensité F_{ij} de la force exercée par un atome j sur un atome i est donnée par la formule suivante :

$$F_{ij} = \begin{cases} 24\frac{\epsilon}{r}\left(\left(\frac{\sigma}{r}\right)^6 - \left(\frac{\sigma}{r}\right)^{12}\right) & \text{si } r \leq r_c \\ 0 & \text{sinon.} \end{cases} \quad (1)$$

ou r est la distance entre i et j. σ et ϵ sont des constantes choisies en fonction des caractéristiques physiques du matériau simulé. Notez que σ représente la distance à laquelle l'interaction entre les atomes est nulle.
Lorsque la distance entre deux atomes excède un seuil nommé *rayon de coupure* (r_c), les forces sont négligées.

$$\vec{F_{i*}} = \sum_{j \neq i} F_{ij}.\hat{u}_{ij} \text{ avec } \hat{u}_{ij} = \frac{\vec{ij}}{r} \quad (2)$$

L'implémentation des interactions entre atomes est effectuée dans la fonction `seq_force`. Notez que les distances entre atomes sont effectuées pour tous les couples d'atomes, d'où un temps d'exécution en $O(n^2)$. Notez aussi que, bien qu'il aurait suffit de calculer qu'une seule fois l'intensité de la force pour chaque paire d'atomes, elle est ici calculée deux fois, une fois par atome. Pourquoi selon vous ?
Essayez cette implémentation séquentielle avec un fichier de configuration contenant un nombre modéré d'atomes, tel que `choc1.conf` :

```
./bin/atoms -v -s 1 -o 1 conf/choc1.conf
```

Appuyez sur `f`, puis sur `m`...

2 Parallélisation avec OpenMP

2.1 Version de base

Inspirez-vous des fonctions de `seq.c` pour implémenter celles de la version OpenMP. Parallélisez les fonctions [2] au niveau des boucles. Dans un second temps on essayera de réduire les appels au pragma `parallel`.
Testez visuellement votre nouvelle version (avec le fichier `choc1.conf` par exemple), puis mesurez les performances en mode *batch*. Par exemple :

```
./bin/atoms -v --omp 1 -i 10 -n 1k
```

Ensuite on travaillera sur la localité des accès aux données : il s'agira de mettre en œuvre une stratégie de type « *first touch* » pour placer physiquemement les données à proximité des cœurs qui les écrivent.

2.2 Optimisation algorithmique : trier les atomes pour éviter des calculs

On sait qu'il est inutile de calculer le potentiel de Lennard-Jones entre deux atomes lorsqu'ils sont séparés par une distance supérieure au rayon de coupure. Aussi, il est intéressant de mettre en œuvre des structures de données reposant sur la géométrie de la configuration pour limiter autant que possible les calculs inutiles. Pour ce projet vous devez implémenter a minima l'une des deux techniques suivantes :

2. Pas seulement la fonction `seq_force`

2.2.1 Choix 1 : Tri selon Z

Supposons que les atomes soient rangés dans un tableau à une dimension et triés suivant l'axe Z. On observe alors qu'un atome placé à l'indice i n'aura d'interaction avec aucun atome rangé après un certain indice (disons j) : en particulier cela se vérifie dès que la distance selon Z (c'est à dire $|z_i - z_j|$) est supérieure au rayon de coupure. Symétriquement, il est possible de trouver un indice $h \leq i$ tel que tous les atomes rangés à un indice inférieur à h soit eux aussi trop éloignés de l'atome rangé à l'indice i pour influencer le calcul. En utilisant cette propriété il est donc possible de réduire parfois très significativement le nombre de calculs.
Il s'agit donc d'implémenter cette technique en utilisant un tri des atomes selon l'axe Z. Le tri pourra dans un premier temps être séquentiel, mais devra être au final parallélisé. Naturellement, vous pourrez vous inspirer de tris parallèles existants. N'oubliez de référencer algorithmes et sources que vous emprunteriez à d'autres.

2.2.2 Choix 2 : Tri par boîtes

Une autre organisation est de découper l'espace en cube et de ne considérer pour le calcul des forces que les atomes présents dans les cubes avoisinants le cube de l'atome considéré. En utilisant des cubes dont la taille d'un côté est égale au rayon de coupure, on sait que les atomes influençant un atome a donné sont forcément dans le cube de a ou dans les 26 cubes entourant le cube de a, ce qui donne 27 cubes à parcourir pour calculer les forces.
Il s'agit de mettre en place les structures de données permettant de trier les atomes par cube, afin de réduire le nombre d'atomes examinés lors du calcul des forces. Voici un algoritme possible utilisant trois tableaux :

- Un tableau `boite` d'entiers dont la dimension correspond au nombre de cubes ;
- Un tableau d'atomes `in` contenant tous les atomes (non triés) ;
- Un tableau d'atomes `out` dans lequel on va ranger les atomes triés suivant les boites.

Tout d'abord on définit une bijection entre les indices du tableau `boite` et les cubes. Ensuite trois parcours de tableaux suffisent pour ranger les atomes par cube :

1. lors d'un premier parcours du tableau `in` on calcule le nombre d'atomes contenu dans chaque cube, le nombre d'atomes du i-ème cube sera placé dans `boite[i]` ;
2. on calcule ensuite la somme préfixée du tableau `boite` :

   ```
   for (int i = 1; i < NbCubes; i++)
     boite[i] += boite[i-1];
   ```

3. lors d'un second parcours du tableau `in`, on range les atomes dans le tableau `out` en utilisant le fait que les atomes du i-ème cube sont à ranger entre les indices `boite[i-1]` et `boite[i]-1`.

On pourra alors utiliser à nouveau le tableau `boite` pour accéder aux atomes d'un cube.
Pour paralléliser la construction du tableau des boîtes en OpenMP, il sera nécessaire d'utiliser un compilateur qui supporte les directives « `atomic capture` » permettant d'incrémenter un entier tout en récupérant son ancienne valeur et ce de manière atomique

```
#pragma omp atomic capture
x = y++;
```

2.2.3 Expérimentations à présenter

Il s'agira de présenter et surtout d'expliquer des courbes d'accélération (durée d'exécution d'une itération séquentielle divisée par la durée en parallèle) en faisant varier le nombre de processeurs utilisés (via la variable d'environnement `{OMP_NUM_THREADS`). On produira et on expliquera des courbes en jouant sur les paramètres suivants :

- Variations sur le type de machine : on fera des expérimentations sur une machine de la salle 008 et sur un des serveurs AMD (boursouf, boursouflet et jolicoeur) ;
- Variations sur le placement des threads et de la mémoire (on tâchera de mesurer l'influence de la variable `GOMP_CPU_AFFINITY`) ;
- Variations sur la politique de distribution d'indices.
- Variations sur le domaine initial (on regardera trois cas `-n 1k`, `choc2` et `choc4`.)

3 Parallélisation avec OpenCL

Il s'agit maintenant d'écrire des noyaux OpenCL permettant de calculer les interactions entre atomes sur des accélérateurs.
Rappel : vous trouverez des ressources utiles dans le répertoire `/net/cremi/rnamyst/etudiants/opencl`.
Et notamment, vous trouverez une synthèse des primitives OpenCL utiles dans un « *Quick Reference Guide* » situé ici :

```
/net/cremi/rnamyst/etudiants/opencl/Doc/opencl-1.2-quick-reference-card.pdf
```

En cas de doute sur le prototype ou le comportement d'une fonction, on pourra se reporter au guide de programmation OpenCL accessible au même endroit :

```
/net/cremi/rnamyst/etudiants/opencl/Doc/opencl-1.2.pdf
```

3.1 Prise en main

Sauvez vos fichiers `seq.c` et `openmp.c` dans un endroit sûr, et récupérez une copie toute neuve des fichiers depuis `/net/cremi/pwacreni/PMG/PROJET/fichiers/`. Replacez ensuite vos fichiers `seq.c` et `openmp.c` dans `fichiers/libsotl/src/`, et relancez `cmake` ainsi que `make`.
Vous pouvez tester la simulation [3] en OpenCL en utilisant l'option `-d` (en lieu et place de `-s` ou `-O`) et en choisissant le *device* correspondant au GPU, par exemple :

```
./bin/atoms -v -d 0 -o 0 conf/default.conf
```

Les noyaux OpenCL sont tous définis dans le fichier `libsotl/kernel/physics.cl`, et les fonctions C qui positionnent leurs paramètres et les invoquent se trouvent dans le fichier `libsotl/src/ocl_kernels.c`.
Regardez le noyau `update_position` (regardez aussi combien de threads sont lancés pour son exécution) et observez l'utilisation de variables de type `float3` pour manipuler la position ou la vitesse d'un atome.

3.2 Détection des collisions entre particules (observation)

La simulation permet, de manière assez rudimentaire, de détecter les collisions entre atomes. En cas de collision, les deux atomes impliqués sont immobilisés [4].
Pour tester cette détection de collisions, il suffit d'appuyer sur « `c` » durant la simulation.
Le fichier `physics.cl` contient le noyau OpenCL `atom_collision` qui effectue ce traitement. Regardez comment est elle programmée. Le problème étant symétrique, la fonction n'effectue que $\frac{n^2}{2}$ tests de proximité, comme illustré en figure 2. Remarquez l'utilisation de la fonction prédéfinie `distance` (cf *OpenCL 1.2 Reference Card*).
Regardez également comment le noyau est invoqué depuis le fichier `ocl_kernels.c`, avec combien de *threads*, etc.

3.3 Application de la gravité

Dans le fichier `libsotl/src/ocl_kernels.c`, observez les paramètres et le nombre de threads utilisés lors du lancement du noyau OpenCL `gravity`.
Implémentez le noyau correspondant dans le fichier `libsotl/kernel/physics.cl`.

3.4 Calcul des forces (observation)

Examinez l'implémentation du noyau `lennard_jones`, qui est exécuté à raison d'un thread par atome. Contrairement à la détection des collisions entre atomes, on n'a pas cherché ici à exploiter la symétrie du problème.

3. À ce stade, presque tous les noyaux sont opérationnels sauf l'application de la force de gravité (touche « g »).
4. Un peu comme dans un Frozen Bubbles 3D, avec beaucoup d'imagination...

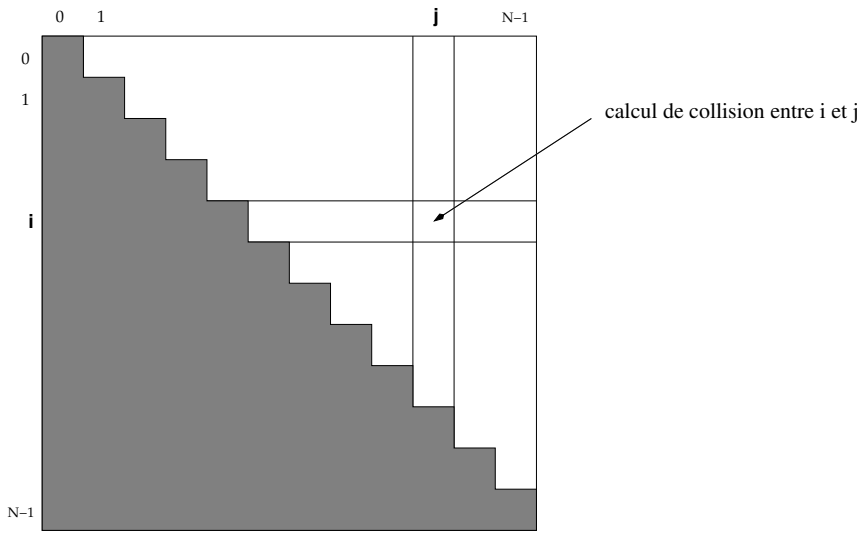

FIGURE 2 – Détecter les collisions entre particules est un traitement symétrique, il n'est donc pas utile de calculer le test tous les couples (i,j) d'atomes, mais simplement pour les couples (i,k) où $j > i$ (zone triangulaire blanche).

Remarquez que l'implémentation, bien que de complexité $\mathcal{O}(n^2)$, utilise tout de même de la mémoire locale partagée au sein des *workgroups* afin de réduire les lectures des positions des atomes depuis la mémoire globale.

3.5 Calcul des forces avec tri par boîtes

L'essentiel des noyaux nécessaire au calcul des forces en utilisant des atomes triés par boîtes est déjà implémenté : examinez la fonction `ocl_one_step_move` (dans `libsotl/src/ocl.c`) et remarquez l'enchainement des noyaux dans le cas où la variable `is_box_mode` est vraie.
Par défaut, cette variable est initialisée à `false` dans le fichier `global_definitions.c`, ce qui provoque l'utilisation du calcul des forces vu à la question précédente.
Lorsque vous basculez cette variable à `true`, le calcul des forces échoue car il manque le noyau qui calcule la somme préfixe à partir du tableau contenant le cardinal de chaque boîte : `scan`.
Votre travail consiste donc à implémenter un ou plusieurs noyaux permettant de réaliser cette somme préfixe. À noter que les données initiales sont contenues dans le tableau désigné par `dev->box_buffer`, et qu'il faut que le résultat du calcul soit placé dans ce même tableau. Le tableau est de taille `dev->domain.total_boxes + 1`.
Conseil : dans un premier temps, implantez l'algorithme de somme préfixe en dehors du projet, par extension de votre algorithme de réduction.