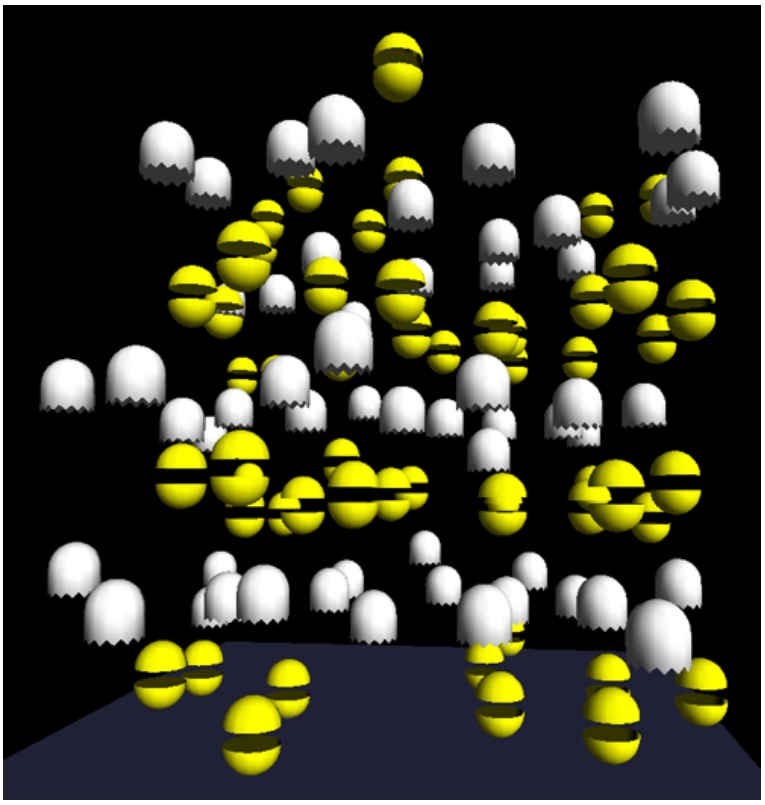
Simulation de particules sur architectures parallèles



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.

1 Premiers pas

1.1 On essaye tout de suite !

Copiez le répertoire `~rnamyst/etudiants/pmg/Particules` 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`. Affichez la liste des périphériques disponibles avec l'option `-l`. Voici un exemple d'affichage obtenu :

```
[rnamyst@happycl] ./bin/atoms -l
[INFO] 2 OpenCL platforms detected
[INFO] Platform 0: Intel(R) OpenCL (Intel(R) Corporation)
[INFO] --- Device 0 : CPU [Intel(R) Xeon(R) CPU E5-2680 v2 @ 2.80GHz] (mem size: 62.99GB, max wg: 81
[INFO] Platform 1: NVIDIA CUDA (NVIDIA Corporation)
[INFO] --- Device 1 : GPU [Tesla K20c] (mem size: 5.00GB, max wg: 1024)
[INFO] --- Device 2 : GPU [Quadro K5000] (mem size: 4.00GB, max wg: 1024)
```

Repérez le numéro associé à la carte graphique NVidia et, en supposant (comme dans cet exemple) que la carte qui nous intéresse ait le numéro 1, lancez une première exécution avec les options suivantes :

```
./bin/atoms -d 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.

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`.

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.
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`).

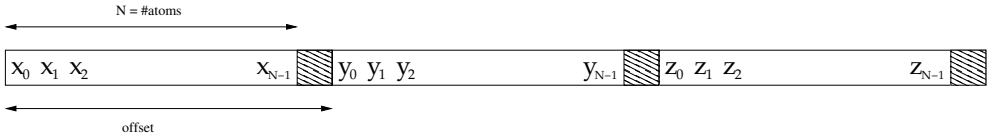


FIGURE 1 – Pour améliorer la performance des accès mémoire sur les accélérateurs, les tableaux `pos_buffer` et `speed_buffer` sont agencés de la manière illustrée ci-dessus : une première tranche contient les coordonnées x (dx pour `speed_buffer`), une seconde contient les y et la troisième contient 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`.

1.4 Mouvement des particules

Regardez le code du noyau `update_position` (dans `kernel/physics.cl`) : c'est celui-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 simple addition de vecteurs... Pourquoi le code est-il aussi simple ?

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` et `max` 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 le noyau `border_collision` 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} (c'est-à-dire `min[1]` dans le code) et, en cas de collision, multiplier la composante vitesse dy (c'est-à-dire `speed[ index + offset]`) par -1.

1.7 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`.

1.8 Détection des collisions entre particules

L'objectif est d'écrire un noyau simple permettant de détecter les collisions entre particules. En cas de collision, les deux atomes impliqués sont immobilisés [2].

1.8.1 Version 1

La façon triviale de procéder est de tester les $N \times (N-1)$ collisions possibles entre tous les atomes. Mais, le problème étant symétrique, il est possible de diviser ce nombre d'opérations par 2, comme illustré en figure 2.

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.
2. Un peu comme dans un Frozen Bubbles 3D, avec beaucoup d'imagination...

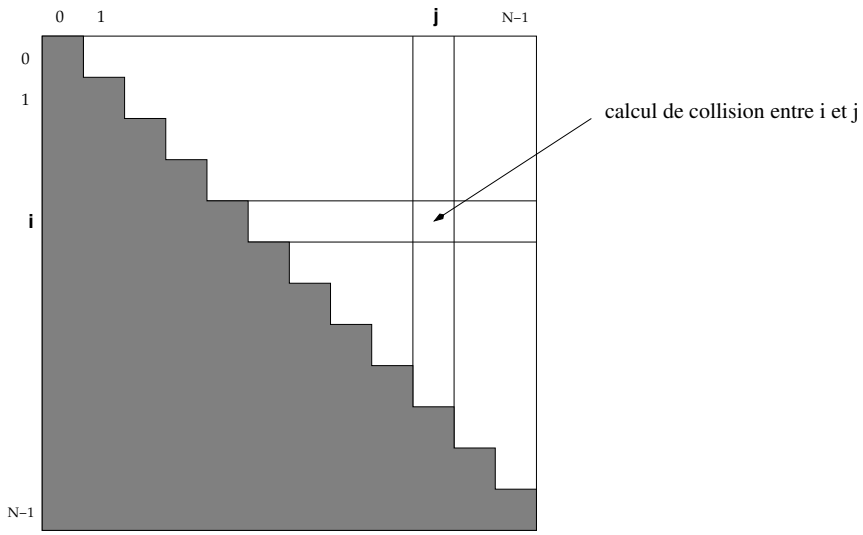


FIGURE 2 – Détecter les collisions entre particules est un traitement symétrique, il n'est donc pas utile de calculer le test tous les couples (i,j) d'atomes, mais simplement pour les couples (i,k) où $j > i$ (zone triangulaire blanche).

Le fichier `physics.cl` contient le noyau OpenCL `atom_collision` qui doit effectuer ce traitement. Regardez comment est elle programmée. Regardez comment le noyau est invoqué depuis le fichier `ocl_kernels.c` (combien de *threads* sont lancés ?), et donnez le code de la fonction `check_collision`. Vous pourrez utiliser la fonction prédéfinie `distance` (qui accepte en arguments des variables de type `float3`, cf *OpenCL 1.2 Reference Card*).
Pour tester cette détection de collisions, il suffit d'appuyer sur « `c` » durant la simulation.

1.8.2 Version 2

Dans la version précédente, le travail n'est pas du tout équitablement réparti entre les threads, ce qui nuit à l'efficacité du noyau. Une façon simple de rétablir une charge de travail uniforme entre les threads est de créer deux fois moins de threads, en faisant en sorte que chaque thread s'occupe de calculer les collisions sur deux lignes de la matrice (figure 3), symétriquement par rapport à la médiane horizontale. Écrivez une seconde version de `atom_collision` en utilisant ce principe. NB : Vous pouvez utiliser une fonction annexe (non préfixée par `__kernel`) pour factoriser du code.
Remarque : Si N est impair, il faut faire un petit ajustement...
Cette version est-elle réellement plus performante que la précédente ? Expliquez pourquoi. Comment faudrait-il s'y prendre pour éviter ce problème ? (on ne demande pas d'écrire le code)

1.9 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`).
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 :

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

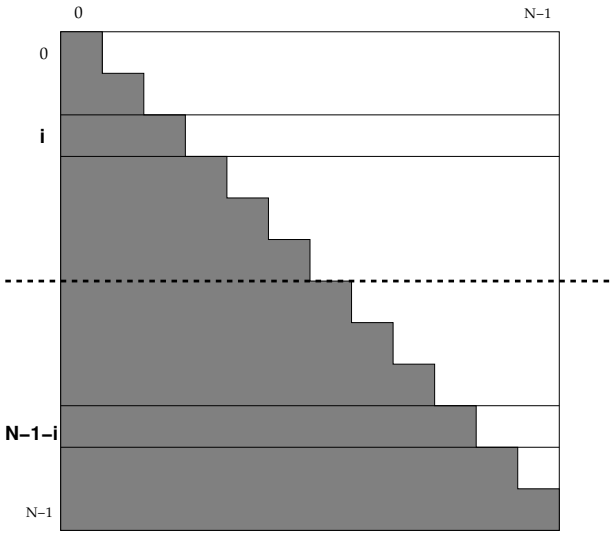


FIGURE 3 – Pour répartir la charge uniformément entre les threads, une solution est de créer deux fois moins de threads que d'atomes, en faisant en sorte que chaque thread i s'occupe de la ligne i ainsi que de la ligne $N - i - 1$.

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.

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

Une fonction `lj_squared_v` vous est fournie pour calculer l'intensité de cette force, à partir d'une distance au carré entre atomes.
En vous inspirant du code qui teste les collisions entre atomes, écrivez le noyau `lennard_jones` qui calcule les forces exercées entre chaque paire d'atomes (en $O(n^2)$ donc) pour mettre à jour les vitesses des atomes.
Essayez votre implémentation 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`...