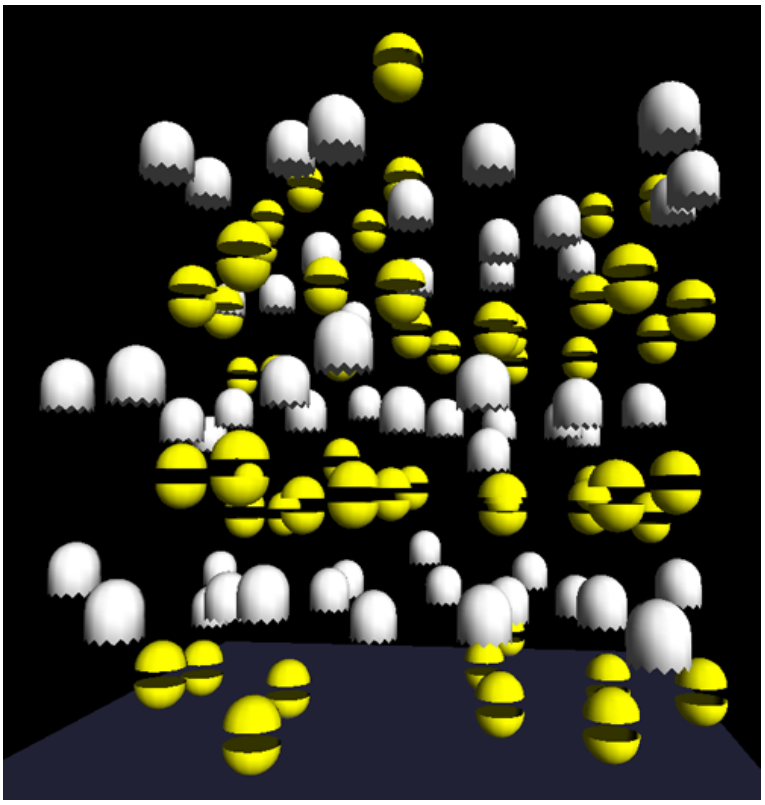
Simulation de particules sur GPU



Il s'agit de concevoir une application de simulation de particules dans un domaine en trois dimensions, en utilisant OpenCL pour calculer les interactions entre particules, et en utilisant OpenGL pour visualiser le déroulement de la simulation en temps réel.
La partie « visualisation OpenGL » vous est fournie, de façon à ce que vous puissiez uniquement vous focaliser sur l'accélération des calculs en OpenCL.
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`

1 Premiers pas

1.1 On essaye tout de suite !

Copiez le répertoire `~rnamyst/etudiants/pmg/Projet` sur votre compte. Dans le répertoire `fichiers/`, tapez `make` puis `./atoms`. 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 'q' ou la touche Escape pour quitter l'application.
Tapez `e` et observez l'animation de certains atomes. C'est la seule animation disponible pour l'instant.
Note : Vous pouvez taper alternativement `p` (mode pacman) et `n` (mode normal), ce qui modifie les coordonnées et les couleurs des points affichés par OpenGL, mais cela n'introduit aucun changement concernant les données manipulées par OpenCL...

1.2 Structure de la simulation

Le code source de la simulation est organisé de la façon suivante :

main.c : Initialisation d'OpenGL et boucle principale de rafraichissement d'écran.

atom.h, atom.c : Gestion des atomes côté CPU. C'est le module qui s'occupe de charger le fichier de configuration (`default.conf` par défaut, mais vous pouvez en essayer d'autres), d'initialiser les positions et les vitesses, etc. C'est surtout le module qui enchaine les noyaux OpenCL destinés à calculer les interactions entre atomes.

vbo.h, 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 très bien vivre sans l'avoir regardé...

ocl.h, ocl.c : Initialisation d'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.

physics.cl Code OpenCL des noyaux qui s'exécuteront sur la carte graphique.

Les fichiers sur lesquels vous allez vous concentrer sont donc `physics.cl` et `atom.c`.
Pour vous familiariser avec le cœur de l'application, regardez dans `main.c` de quelle manière la fonction `animateGPU` est appelée. Le code de cette fonction se trouve dans le fichier `atom.c`. Inspectez cette fonction et regardez notamment le code de la fonction `eating`, qui exécute le seul noyau OpenCL disponible pour le moment.

1.3 Positions et coordonnées des atomes

Pour calculer le résultat des interactions entre atomes, on mémorisera 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.

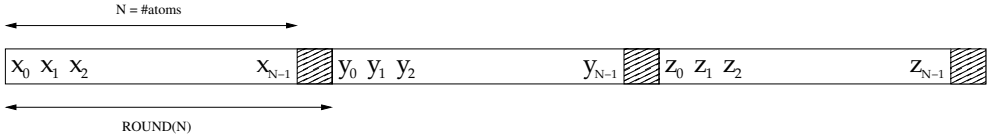


FIGURE 1 – Pour améliorer la performance des accès mémoire sur le GPU, les tableaux `pos_buffer` et `speed_buffer` sont agencés de la manière illustrée ci-dessus : une première tranche contient les coordonnées x (dx pour `speed_buffer`), une seconde contient les y et la troisième contient les z. La taille totale de chaque tranche est arrondie au multiple de 16 immédiatement supérieur au nombre d'atomes N. Ce nombre est obtenu en évaluant la macro `ROUND(N)`.

L'application stoque la position des atomes et leur vitesse dans deux tableaux distincts (voir la fonction `initializeAtoms` au début du fichier `atom.c`). Une fois initialisés, ces tableaux sont recopiés dans des *buffers* OpenCL alloués sur la carte graphique : `pos_buffer` et `speed_buffer`. Pour des raisons d'efficacité, les données de ces tableaux sont organisées comme illustré en figure 1.
Les données sont transférées « une fois pour toutes [1] » en début de simulation depuis la mémoire principale vers la mémoire du GPU : Regardez à la fin de la fonction `ocl_init` (dans `ocl.c`).
Pour les besoins de la visualisation, un *buffer* alloué sur le GPU nommé `vbo_buffer` (« *Vertex Buffer Object* ») contient les coordonnées de chaque point utilisé pour former une sphère par atome. Ce tableau contient une (grande) 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 [2].

2 Mouvement des particules

2.1 Déplacement des points OpenGL

Examinez le noyau OpenCL « `eating` », qui provoque le déplacement selon l'axe y des points d'un atome sur deux. Les points de l'hémisphère nord d'une sphère se déplacent de façon opposée aux points de l'hémisphère sud.
Inspirez-vous de ce noyau pour écrire le noyau `move_vertices`, qui doit appliquer à chaque point formant une sphère le vecteur déplacement (dx, dy, dz) de l'atome correspondant. Il y a plusieurs stratégies possibles concernant le nombre de threads utilisés. Dans le noyau `eating`, le nombre de thread créés est égal au nombre d'éléments du tableau `vbo_buffer`...
Pour tester votre noyau, lancez le programme `./atoms` et appuyez sur `m` (pour « move »). Vous devriez voir les atomes se déplacer (selon une vitesse initiale déterminée aléatoirement).

2.2 Mise à jour de la position des atomes

Les noyaux que nous écrirons par la suite n'utiliseront pas les données de `vbo_buffer`, mais simplement celles de `pos_buffer` qui contiennent les coordonnées des centres de chaque atome. Il faut donc qu'à chaque itération ce tableau soit également mis à jour en fonction des vitesses des atomes.
Écrivez le noyau `update_position`. À peu de choses près, cela devrait ressembler à une addition de vecteurs...
Pour tester que votre noyau est correct, vous pouvez rappatrier les données depuis `pos_buffer` vers la mémoire principale et vérifier leur nouvelle valeur.

3 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 le noyau `border_collision` qui teste, pour chaque atome, la collision avec l'un des bords. Si la distance entre le centre d'un atome et le bord est inférieure au rayon d'un atome, 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 $y - y_{min_ext}$ par rapport au rayon et, en cas de collision, multiplier la composante vitesse dy par -1.
Réfléchissez bien à créer un nombre de threads maximisant le parallélisme sur le GPU.

1. OK, ce n'est pas tout à fait vrai, mais c'est anecdotique.
2. Cette façon de stocker les coordonnées n'est pas forcément idéale pour les noyaux OpenCL, mais elles est imposée par OpenGL : le buffer `vbo_buffer` est en effet partagé entre OpenGL et OpenCL.

4 Détection des collisions entre particules

L'objectif est d'écrire un noyau simple permettant de détecter les collisions entre particules. En cas de collision, on se limitera simplement [3] à rendre les deux atomes impliqués immobiles.

4.1 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.

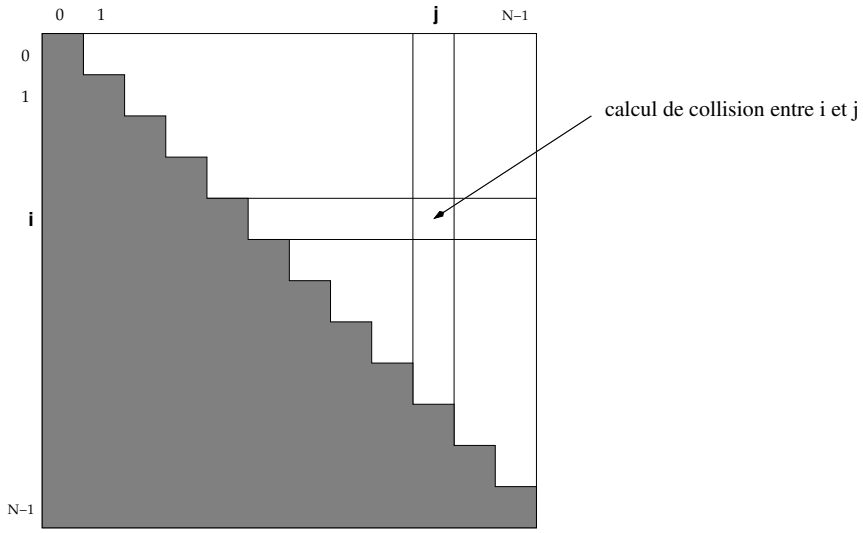


FIGURE 2 – Détecter les collisions entre particules est un traitement symétrique, il n'est donc pas utile de calculer le test tous les couples (i,j) d'atomes, mais simplement pour les couples (i,k) où $j > i$ (zone triangulaire blanche).

Écrivez une première version du noyau `atom_collision` en lançant N threads (N étant le nombre total d'atomes). Vous pourrez utiliser des variables de type `float3` pour manipuler les positions des atomes et, surtout, calculer efficacement les distances entre atomes (fonction `distance`, cf *OpenCL 1.2 Reference Card*).
Testez le noyau en appuyant sur `c` (pour « collision ») durant l'exécution.

4.2 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)

4.3 Version 3

En remarquant que les mêmes données sont lues de nombreuses fois depuis la mémoire globale de la carte, proposez une version utilisant des *groupes* de threads partageant de la mémoire pour diminuer le

3. Pour l'instant...

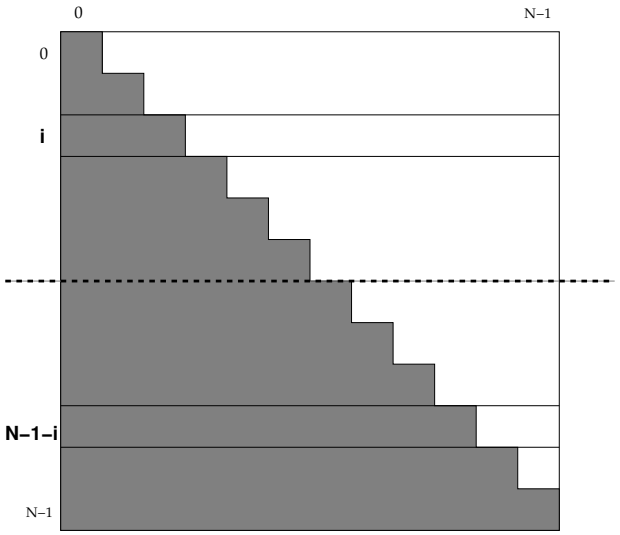


FIGURE 3 – Pour répartir la charge uniformément entre les threads, une solution est de créer deux fois moins de threads que d'atomes, en faisant en sorte que chaque thread i s'occupe de la ligne i ainsi que de la ligne $N - i - 1$.

nombre de ces accès.
Comparez les performances de vos trois versions avec un grand nombre d'atomes (fichier `huge.conf` par exemple).

5 Force gravitationnelle uniforme

Notre premier calcul de forces appliquées aux atomes est très simple : il s'agit d'appliquer une force $\vec{g}$ (gravité) colinéaire au vecteur unitaire $\vec{y}$ à chaque atome. Pour cela, l'intensité du vecteur $\vec{g}$ sera simplement retranchée à la composante vitesse dy de chaque atome, à chaque itération.
Écrivez le noyau `gravity`. Testez-le en appuyant sur `g` (pour « gravité ») durant l'exécution.

6 Chocs élastiques entre atomes

Nous allons maintenant simuler de véritables collisions entre atomes suivant un modèle de *choc élastique* entre particules. Dans un monde parfait sans frottements ni perte d'énergie, les deux particules impliquées lors d'un choc s'échangent leurs quantités de mouvement perpendiculairement au « plan de collision » tout en conservant leurs vitesses tangentielles (cf `http://fr.wikipedia.org/wiki/Choc_élastique`).
Dans un univers en trois dimensions, il n'est pas facile de calculer cet échange de quantités de mouvement (et donc de vitesses car nous considérons des atomes de masse identique) directement. La technique usuelle est d'effectuer un changement de repère pour que cette opération soit triviale dans le nouveau référenciel, puis de revenir ensuite au repère initial. Dans le cas présent, il s'agit donc de trouver, pour un couple donné de particules entrant en collision, un repère orthonormé $(\vec{i}, \vec{j}, \vec{k})$ tel que le vecteur $\vec{i}$ soit colinéaire à la droite passant par le centre des deux atomes.
Si on nomme C_A et C_B les centres des atomes A et B entrant en collision, le vecteur $\vec{i}$ peut être défini comme suit :

$$\vec{i} = \frac{\vec{C_B} - \vec{C_A}}{\|\vec{C_B} - \vec{C_A}\|}$$

Notez qu'en OpenCL, une primitive *normalize* simplifie grandement le calcul des coordonnées de $\vec{i}$:

```
float3 Ca, Cb;
float3 i = normalize(Cb - Ca);
```

Il nous reste maintenant à déterminer $\vec{j}$ et $\vec{k}$ parmi une infinité de possibilités. Dans le rapport publié ici `http://hal.archives-ouvertes.fr/hal-00685302/` pages 34–37, l'auteur indique une façon judicieuse de choisir $\vec{j}$ de façon à simplifier les calculs de changement de repère. Il suffit de définir :

$$\vec{j} = \frac{\vec{x} \wedge \vec{i}}{\|\vec{x} \wedge \vec{i}\|}, \vec{k} = \vec{i} \wedge \vec{j}$$

Pour passer du repère $(\vec{x}, \vec{y}, \vec{z})$ au repère $(\vec{i}, \vec{j}, \vec{k})$, il suffit d'effectuer une rotation autour du vecteur $\vec{j}$. En posant $\vec{i} = (x_i, y_i, z_i)$, la matrice M de transformation est définie comme suit :

$$M = \begin{bmatrix} x_i & y_i & z_i \\ -y_i & x_i + \frac{z_i^2}{1+x_i} & \frac{-y_i z_i}{1+x_i} \\ -z_i & \frac{-y_i z_i}{1+x_i} & x_i + \frac{y_i^2}{1+x_i} \end{bmatrix}$$

Notez que si $\vec{i}$ $\vec{x}$ sont colinéaires, alors il n'y aucunement besoin d'effectuer une rotation. On peut minimiser le traitement de ce cas particulier en posant :

$$M = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Pour calculer les nouvelles vitesses $\vec{V'_A}$ et $\vec{V'_B}$ des deux atomes, étant données leurs vitesses respectives $\vec{V_A}$ et $\vec{V_B}$ au moment du choc, il suffit de poser :

$$\vec{V_{A_R}} = M\vec{V_A} = \begin{bmatrix} i_A \\ j_A \\ k_A \end{bmatrix}, \vec{V_{B_R}} = M\vec{V_B} = \begin{bmatrix} i_B \\ j_B \\ k_B \end{bmatrix}$$

$$\vec{V'_{A_R}} = \begin{bmatrix} i_B \\ j_A \\ k_A \end{bmatrix} \vec{V'_{B_R}} = \begin{bmatrix} i_A \\ j_B \\ k_B \end{bmatrix}$$

$$\vec{V'_A} = {}^tM\vec{V'_{A_R}}, \vec{V'_B} = {}^tM\vec{V'_{B_R}}$$

À vous de jouer ! Notez qu'OpenCL ne dispose pas (encore) des opérations permettant de multiplier des matrices par des vecteurs. Toutefois, une primitive `dot` permet de calculer le produit scalaire entre deux vecteurs (e.g. deux `float3`). Aussi, une bonne idée serait de stocker chaque ligne de la matrice M dans un vecteur...
Vous pouvez tester votre gestion des collisions à l'aide de différents fichiers de configuration :

./atoms bounce.conf pour une simple collision selon l'axe x ;

./atoms collide.conf pour une collision entre 2 atomes ;

./atoms plan.conf pour une collision en chaîne ;

./atoms pour des collisions arbitraires dans un gaz parfait.

N'hésitez pas à vous fabriquer vos propres jeux d'essais (le format des fichiers de configuration est simple).

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`).
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).

Si on appelle d la distance entre deux atomes, on peut calculer l'énergie potentielle associée à cette paire d'atomes de cette façon :

$$E = 4\epsilon \left((\frac{\sigma}{d})^{12} - (\frac{\sigma}{d})^{6} \right)$$

σ est une constante qui représente la distance à laquelle l'énergie entre les atomes est nulle. Au delà, les atomes s'attirent. En Deça, ils se repoussent. Vous pourrez expérimenter avec une valeur égale à un peu plus du double du rayon des atomes.
Note : Si vous avez correctement implémenté les collisions entre particules, vous devriez pouvoir vous passer du terme $(\frac{\sigma}{d})^{12}$ qui matérialise la répulsion. Essayer !

7.1 Version 1

Écrivez le noyau `http://fr.wikipedia.org/wiki/Potentiel_interatomique` qui calcule les $N \times (N-1)$ interactions entre atomes. On utilisera bien entendu de la mémoire locale pour minimiser les accès mémoire globaux (comme à la question 4.3).

7.2 Version 2

Comme dans le cas des collisions, il est normalement possible de tirer parti de la symétrie du problème pour ne calculer que $\frac{N(N-1)}{2}$ interactions. La difficulté réside dans l'accumulation des forces résultantes pour chaque atome, qui sont des opérations concurrentes...
Jetez un oeil aux opérations atomiques dans l'*OpenCL API Ref Card* et déduisez-en une façon de programmer correctement cette seconde version.

8 Remise du projet

Dans ce projet, on attend une évaluation des performances de chaque implémentation proposée. À cette fin, vous pourrez basculer dans un mode de visualisation « élémentaire » de façon à pouvoir augmenter significativement le nombre d'atomes. Il suffit pour cela de commenter la ligne suivante dans `vbo.h` :

```
#define _SPHERE_MODE_
```

Il faudra également augmenter la valeur de la constante `MAX_VERTICES` (dans ce même fichier).
Vous décrirez les courbes de performances obtenues pour chaque implémentation (n'oubliez pas de tester avec un grand nombre d'atomes) dans un rapport. Vous pourrez également expliquer les grandes lignes de ces implémentations.