université de BORDEAUX

Programmation Multicoeur

SIMULATION DE PARTICULES SUR ARCHITECTURES PARALLÈLES

Auteurs :
- Henri Frank **ANABA**
- Olivier **Constans**

Responsable :
M. R NAMYST

8 mai 2015

Table des matières

Chapitre 1

Présentation du sujet

Dans le cadre du module d'enseignement de programmation parallèle, nous devons réaliser un projet. Ce projet consiste en la conception d'une application de simulation de particules dans un domaine en trois dimensions. Pour ce faire, il a été mis à notre disposition un rendu OpenGL des particules ainsi que son code et une version séquentielle du code. Le rendu permet de visualiser la simulation en temps réel. Le but étant que nous nous concentrions uniquement sur l'accélération des calculs en parallèle.

Ce rapport présente le travail réalisé par le binôme constitué de Henri Frank **ANABA** et d'Olivier **CONSTANS**.Il est en effet le compte-rendu de la réalisation des deux parties du projet : la partie **OpenMP** et la partie **OpenCL**

Chapitre 2

Traitement du sujet

2.1 Parallélisation avec OpenMP

2.1.1 Parallélisation des boucles

La réalisation de la version de base consistait juste en une parallélisation des boucles de la version séquentielle. Pour ce, nous avons choisi d'utiliser la directive OpenMP pour la parallélisation des boucles **for** #pragma omp for.

Pour chaque fonction appelée par la fonction principale de simulation, nous avons optimisé le calcul des boucles for.Dans la fonction, **omp_update_vbo**, parallélisation de la boucle de mise à jour des vitesses des atomes :

```
static void omp_update_vbo (sotl_device_t *dev)
{
  sotl_atom_set_t *set = &dev->atom_set;
  sotl_domain_t *domain = &dev->domain;

   #pragma omp for
  for (unsigned n = 0; n < set->natoms; n++) {
    ...
  }
}
```

Dans la fonction, **omp_move**, parallélisation de la boucle de mise à jour des vitesses de chaque atome :

```
static void omp_move (sotl_device_t *dev)
{
  sotl_atom_set_t *set = &dev->atom_set;

  #pragma omp for
  for (unsigned n = 0; n < set->natoms; n++) {
    set->pos.x[n] += set->speed.dx[n];
    set->pos.y[n] += set->speed.dy[n];
    set->pos.z[n] += set->speed.dz[n];
  }
}
```

Dans la fonction, **omp_ gravity**, parallélisation de l'application de la gravité aux vitesses des atoms :

```
static void omp_gravity (sotl_device_t *dev)
{
  sotl_atom_set_t *set = &dev->atom_set;
  const calc_t g = 0.005;
  #pragma omp for
   for (unsigned current = 0; current < set->natoms; current++) {
        set->speed.dy[current] = set->speed.dy[current]-g;
  }
}
```

Dans la fonction **omp_bounce**, parallélisation de la boucle de calcul.

```
static void omp_bounce (sotl_device_t *dev)
{
  sotl_atom_set_t *set = &dev->atom_set;
  sotl_domain_t *domain = &dev->domain;
    #pragma omp for
     for (unsigned current = 0; current < set->natoms; current++) {
        ...
     }
}
```

Dans la fonction **tri**, parallélisation des boucles de permutation des atomes lors du tri :

```
void tri(sotl_atom_set_t * set){

  int paire = (set->natoms%2==0)?1:0;
  int impaire = (set->natoms%2==0)?0:1;
  for(unsigned i=0; i<set->natoms; i++){
    if(i%2==0){
      #pragma omp parallel
      #pragma omp for schedule(runtime)
      for(unsigned j =0; j<set->natoms-impaire;j+=2){
        if(set->pos.z[j]<set->pos.z[j+1]){
         echange(set,j,j+1);
        }
      }
    }else{

      #pragma omp parallel
      #pragma omp for schedule(runtime)
      for(unsigned j=1;j<(set->natoms-paire);j+=2){
        if(set->pos.z[j]<set->pos.z[j+1]){
          echange(set,j,j+1);
        }
      }
    }
}
```

Dans la fonction **omp_force**, parallélisation des boucles de calcul des forces d'interactions entre les atomes :

```
static void omp_force (sotl_device_t *dev)
{
  sotl_atom_set_t *set = &dev->atom_set;
  tri(set);

  #pragma omp parallel
  #pragma omp for schedule(runtime)
  for (unsigned current = 0; current < set->natoms; current++) {
    ...
  }

 //atome Z inferieur
 #pragma omp parallel
 #pragma omp for schedule(runtime)
 for(int other = current+1;other<set->natoms;other++){
   ...
        }
    }
  }

}
```

2.1.2 Mise en place de la stratégie « frist touch »

Le principe de cette stratégie étant l'allocation d'un bloc de donnée le plus prêt possible de l'endroit où le code va s'exécuter. Pour ce faire, nous avons créé une fonction **first_touch** qui fait une manipulation des données sans modification afin que les valeurs par défaut ne soient pas perdues.

```
void first_touch(sotl_device_t *dev)
{
  sotl_atom_set_t *set = &dev->atom_set;
  sotl_domain_t *domain = &dev->domain;

 #pragma omp parallel
 #pragma omp for schedule(runtime)
  for (unsigned current_atom=0; current_atom< set->natoms; current_atom++)
  {
    vbo_vertex[current_atom*3 + 0] += 0;
    vbo_vertex[current_atom*3 + 1] += 0;
    vbo_vertex[current_atom*3 + 2] += 0;

    set->pos.x[current_atom]+=0;
    set->speed.dx[current_atom]+=0;
    set->pos.y[current_atom]+=0;
    set->speed.dy[current_atom]+=0;
    set->pos.z[current_atom]+=0;
    set->speed.dz[current_atom]+=0;

    // Atom color depends on z coordinate
```

```
      float rate= domain->min_ext[2] / (domain->max_ext[2]);

      vbo_color[current_atom*3 + 0] += 0;
      vbo_color[current_atom*3 + 1] += 0;
      vbo_color[current_atom*3 + 2] += 0;
      atom_state[current_atom] = atom_state[current_atom] + rate - rate;
  }

}
```

Cette fonction est appellée lors de l'initialisation d'une simulation en parallèle.

```
void omp_init (sotl_device_t *dev)
{

#ifdef _SPHERE_MODE_
  ...
#endif
  first_touch(dev);
  borders_enabled = 1;

  dev->compute = SOTL_COMPUTE_OMP; // dummy op to avoid warning
}
```

2.1.3 tri suivant z

Pour éviter les calculs inutiles, une solution était de trier les atomes suivant l'axe z. Ainsi pour un atome donné, le calcul du potentiel de *Lennard Jones* n'est effectué que pour les atomes se situant à une distance inférieure au rayon de coupure. Les autres n'interagissant pas.Notre implémentation du tri suivant Z a été faite dans la fonction **tri**.

```
void tri(sotl_atom_set_t * set){

  int paire = (set->natoms%2==0)?1:0;
  int impaire = (set->natoms%2==0)?0:1;
  for(unsigned i=0; i<set->natoms; i++){
    if(i%2==0){
      #pragma omp parallel
      #pragma omp for schedule(runtime)
      for(unsigned j =0; j<set->natoms-impaire;j+=2){
        if(set->pos.z[j]<set->pos.z[j+1]){
         echange(set,j,j+1);
        }
      }
    }else{
      #pragma omp parallel
      #pragma omp for schedule(runtime)
      for(unsigned j=1;j<(set->natoms-paire);j+=2){
        if(set->pos.z[j]<set->pos.z[j+1]){
          echange(set,j,j+1);
      }
```

```
    }
  }
}
```

Pour faciliter le tri, nous avons implémenté une fonction d'échange qui fait l'échange de position entre deux atomes d'un même tableau.

```
void echange(sotl_atom_set_t * set, int a, int b){
        calc_t tmpx;
        calc_t tmpy;
        calc_t tmpz;
        calc_t tmpsx;
        calc_t tmpsy;
        calc_t tmpsz;

        tmpx = set->pos.x[a];
        tmpy = set->pos.y[a];
        tmpz = set->pos.z[a];
        tmpsx = set->speed.dx[a];
        tmpsy = set->speed.dy[a];
        tmpsz = set->speed.dz[a];

        set->pos.x[a] = set->pos.x[b];
        set->pos.y[a] = set->pos.y[b];
        set->pos.z[a] = set->pos.z[b];
        set->speed.dx[a] = set->speed.dx[b];
        set->speed.dy[a] = set->speed.dy[b];
        set->speed.dz[a] = set->speed.dz[b];

        set->pos.x[b] = tmpx;
        set->pos.y[b] = tmpy;
        set->pos.z[b] = tmpz;
        set->speed.dx[b] = tmpsx;
        set->speed.dy[b] = tmpsy;
        set->speed.dz[b] = tmpsz;

}
```

Une fois les atomes triés, la fonction de calcul des interactions utilise les considérations du tri pour n'évaluer que les atomes se portant au calcul.

```
static void omp_force (sotl_device_t *dev)
{
  sotl_atom_set_t *set = &dev->atom_set;
  tri(set);

  #pragma omp parallel
  #pragma omp for schedule(runtime)
  for (int current = 0; current < (int)set->natoms; current++) {
    calc_t force[3] = { 0.0, 0.0, 0.0 };
```

```
    //atome Z supperieur
    for(int other = current-1;other>-1;other--){

        if(abs(set->pos.z[current]-set->pos.z[other])
                > LENNARD_SQUARED_CUTOFF) {
                break;
        }else{
                calc_t sq_dist = squared_distance (set, current, other);
                if (sq_dist < LENNARD_SQUARED_CUTOFF) {
                        calc_t intensity = lennard_jones (sq_dist);

                        force[0] += intensity
                                        * (set->pos.x[current]
                                           - set->pos.x[other]);
                        force[1] += intensity
                                        * (set->pos.x[set->offset + current]
                                            -set->pos.x[set->offset + other]);
                        force[2] += intensity
                                        *(set->pos.x[set->offset * 2 + current]
                                         -set->pos.x[set->offset * 2 + other]);
                }
        }
    }
    //atome Z inferieur
    for(int other = current+1;other<(int)set->natoms;other++){
        if(abs(set->pos.z[current]-set->pos.z[other])
                 > LENNARD_SQUARED_CUTOFF){
                break;
        }else{
                calc_t sq_dist = squared_distance (set, current, other);
                if (sq_dist < LENNARD_SQUARED_CUTOFF) {
                        calc_t intensity = lennard_jones (sq_dist);

                        force[0] += intensity
                                        * (set->pos.x[current]
                                           - set->pos.x[other]);
                        force[1] += intensity * (set->pos.x[set->offset + current]
                                        -set->pos.x[set->offset + other]);
                        force[2] += intensity
                                        * (set->pos.x[set->offset * 2 + current]
                                        -set->pos.x[set->offset * 2 + other]);
        }
        }
    }
    set->speed.dx[current] += force[0];
    set->speed.dx[set->offset + current] += force[1];
    set->speed.dx[set->offset * 2 + current] += force[2];

  }
}
```

On remarque que l'algorithme peut être divisé en deux phases. Pour un atome donné, on regarde, suivant l'axe z, ses voisins qui interagissent avec lui : d'une part ceux qui on une coordonnée z inférieure à la sienne et d'autre part ceux avec la coordonnée supérieure.

2.1.4 tri par boite

Toujours dans l'optique d'accélérer les calculs, nous avons implémenté un tri par boite. tout comme le tri suivant z, l'idée est de n'évaluer que les atomes se trouvant dans les 26 cubes entourant le cube de a et dans le cube lui même. L'algorithme de ce tri nous a été fourni. Nous l'avons implémenter en combinant le tri et le calcul des forces dans la même fonction (**omp_force_cube**)

```
void omp_force_cube(sotl_device_t *dev){

  sotl_atom_set_t *set = &dev->atom_set;
  sotl_domain_t *domain = &dev->domain;

  sotl_atom_set_t *in = malloc(sizeof(sotl_atom_set_t));
  atom_set_init(in,set->natoms,set->offset);


  for(int i=0; i<(int)set->natoms; i++){
    atom_set_add(in,set->pos.x[i],set->pos.y[i],set->pos.z[i],
                          set->speed.dx[i],set->speed.dy[i],set->speed.dz[i]);
  }

  int NbCubes = (int) domain->total_boxes;

  int* boite= calloc(NbCubes,sizeof(int));
  int* boiten= calloc(NbCubes,sizeof(int));

  for(int i= 0; i < (int)in->natoms; i++){
    int numb =atom_get_num_box(domain,in->pos.x[i],in->pos.y[i],
                          in->pos.z[i],LENNARD_SQUARED_CUTOFF);
    boite[numb]++;
    boiten[numb]++;
  }


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

  for(int i = 0; i < (int)in->natoms; i++){
    int numb =atom_get_num_box(domain,in->pos.x[i],in->pos.y[i],
                      in->pos.z[i],LENNARD_SQUARED_CUTOFF);

    int indice = boite[numb-1]+boiten[numb]-1;
    set->pos.x[indice]=in->pos.x[i];
    set->pos.y[indice]=in->pos.y[i];
    set->pos.z[indice]=in->pos.z[i];
    set->speed.dx[indice]=in->speed.dx[i];
    set->speed.dy[indice]=in->speed.dy[i];
    set->speed.dz[indice]=in->speed.dz[i];
    boiten[numb]--;
  }
    #pragma omp parallel
    #pragma omp for schedule(runtime)
    for (int current = 0; current < (int)set->natoms; current++) {
```

```
    calc_t force[3] = { 0.0, 0.0, 0.0 };

    int bx =(int)((set->pos.x[current] - domain->min_border[0])
                  *LENNARD_SQUARED_CUTOFF);
    int by =(int)((set->pos.y[current] - domain->min_border[1])
                  *LENNARD_SQUARED_CUTOFF);
    int bz =(int)((set->pos.z[current] - domain->min_border[2])
                  *LENNARD_SQUARED_CUTOFF);

    for(int z =(bz==0)?bz:bz-1;
              z<= bz+1 && z<(int)domain->boxes[2]; z++){
      for(int y=(by==0)?by:by-1;
              y<= by+1 && y<(int)domain->boxes[1]; y++){
        for(int x=(bx==0)?bx:bx-1;
              x<=bx+1 && x<(int)domain->boxes[0];x++){
          int numb = z * domain->boxes[0]* domain->boxes[1]
                       + y * domain->boxes[0] + x;
          for(int other =(numb==0)?boite[0]:boite[numb-1];
              other<boite[numb]; other++){
            if(current != other){
              calc_t sq_dist = squared_distance (set, current, other);

              if (sq_dist < LENNARD_SQUARED_CUTOFF) {
                calc_t intensity = lennard_jones (sq_dist);
                force[0] += intensity
                         * (set->pos.x[current] - set->pos.x[other]);
                force[1] += intensity
                         * (set->pos.x[set->offset + current]
                            -set->pos.x[set->offset + other]);
                force[2] += intensity
                         * (set->pos.x[set->offset * 2 + current]
                           -set->pos.x[set->offset * 2 + other]);
              }
            }
          }
        }
      }

      set->speed.dx[current] += force[0];
      set->speed.dx[set->offset + current] += force[1];
      set->speed.dx[set->offset * 2 + current] += force[2];
    }

    free(boiten);
    atom_set_free(in);
    free(in);
    free(boite);
  }
```

2.2 Expérimentations

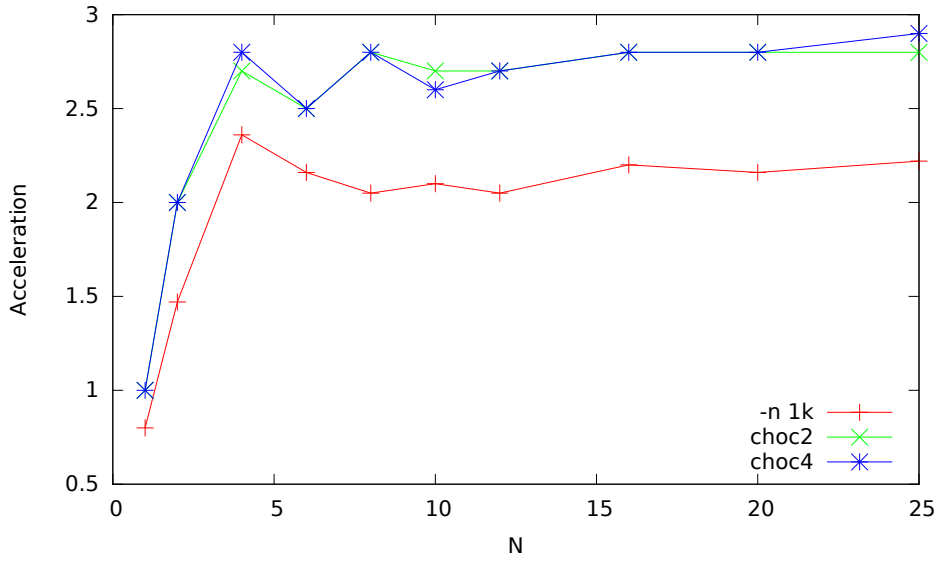
Les expériences consistent en la présentation des courbes d'accélération par la variation du nombre de processeurs utilisés. de trie par boite : 6600 μs

2.2.1 Variations sur le domaine initiale

Dans un premier temps, nous avons fait variée le domaine initial sur les différents algorithmes en séquentiels.

	-n 1 k	choc2.conf	choc4.conf
algorithm de base	7500 μs	179000 μs	180000 μs
algorithm de trie par Z	5600 μs	280000 μs	50000 μs
algorithm de trie par boite	6600 μs	18000 μs	18000 μs

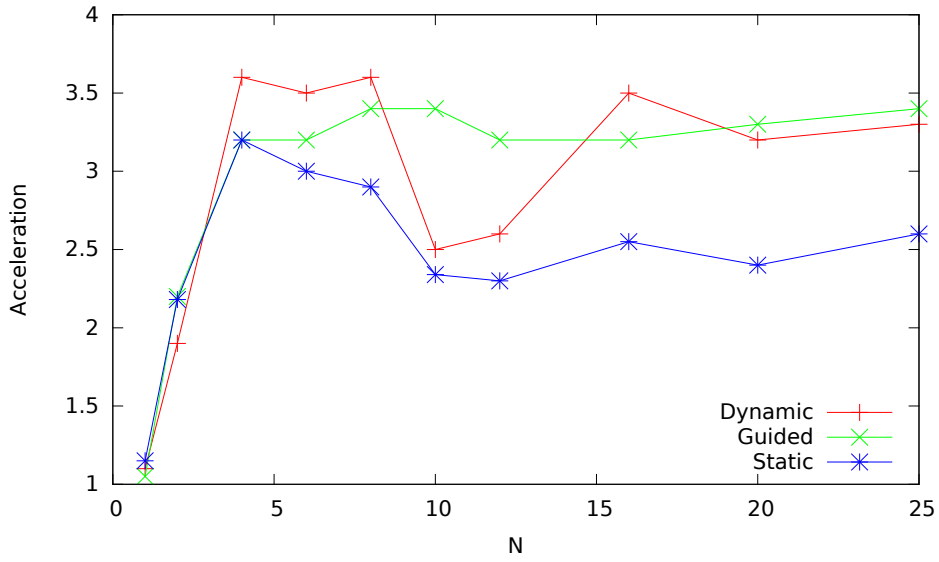
Nous avons utilisé l'algorithme séquentiel le plus performant de chaque domaine pour calculer l'accélération.



Avec les domaines de initiaux *choc*2 et *choc*4, l'accélération augmentes jusqu'à stagner. Alors que pour le domaine initial $-n1k$, l'accélération augmente, puis diminue jusqu'à stagner. La diminution de performance est du au fait qu'on crait trop de thread. Les threads ne sont plus rentable,la création d'un thread prend plus de temps que se que fait gagner le thread.

2.2.2 Variations sur la politique de distribution d'indices

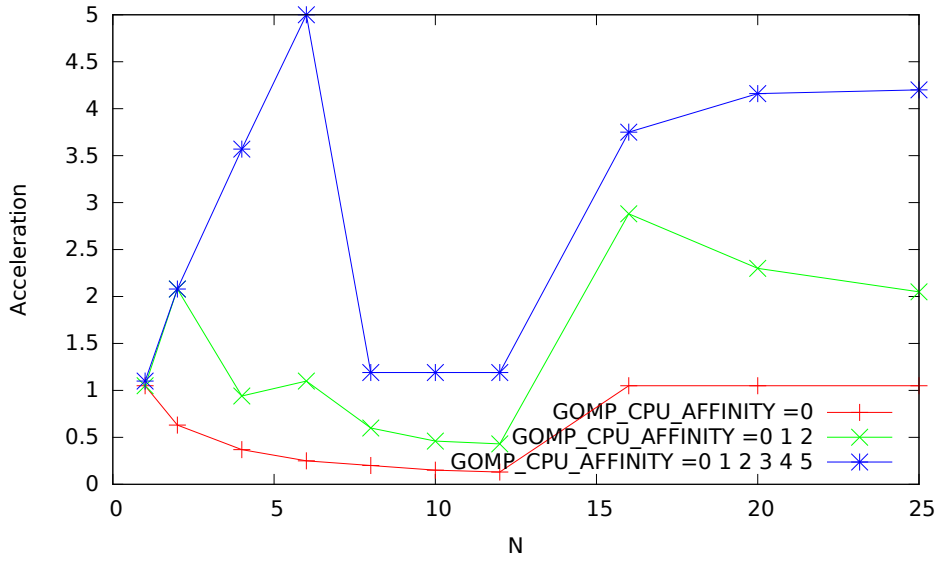
Nous avons ensuite fait varier la politique de distribution d'indice,en modifiant la varaible d'environnement "OMP_SCHEDULE". La distribution dite dynamique et static semble se valoir quand il y a peu de threads. Mais lors que le nombre de thread est important, la distribution dynamique semble meilleurs.



2.2.3 Variations sur le placement des threads et de la mémoire

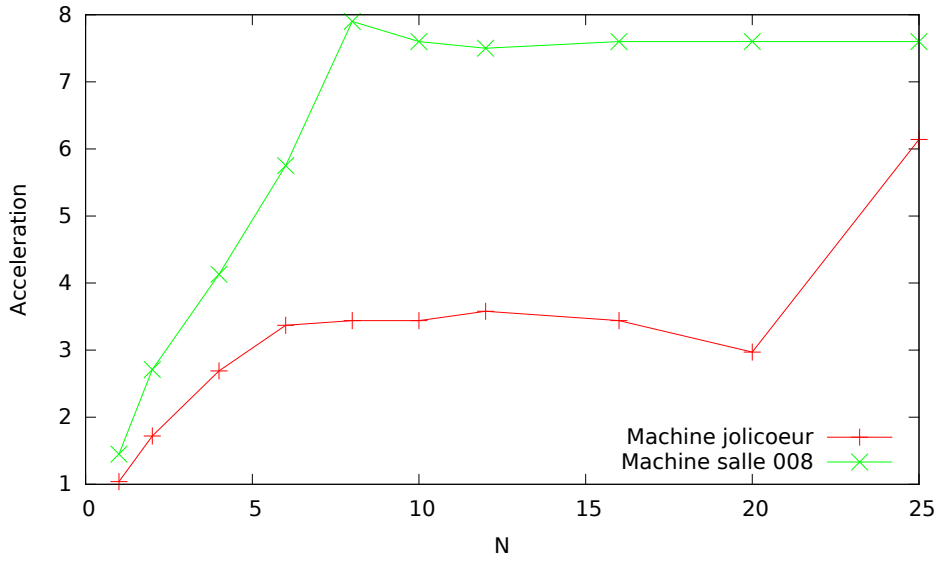
Lorsqu'on utilise un seul coeur, les performances sont décroissantes. Ce qui est normal car on crait des threads sur un seul coeur et on perd du temps en interruption pour changer de thread sur le coeur. On remarque cependant qu'au bout d'un certain nombre de thread, nous récupérons une accélération de 1. Ceci doit être une optimisation automatique de l'ordinateur.

Pour les autres cas, l'accélération augmente jusqu'a que le nombre de thread soit égale au nombre de coeur. Si le nombre de thread est supérieur au nombre de coeurs, l'accélération diminue à cause des pertes de temps pour changer les threads sur les coeurs.



2.2.4 Variation sur le type de machine

On remarque que l'accélération varie selon la machine utilisé. La machine jolicoeur a de moins bonne performance que la machine de la salle 008.



2.3 Parallélisation avec OpenCL

2.3.1 Application de la gravité

```
// This kernel is executed with one thread per atom
__kernel
void gravity (__global calc_t * speed, calc_t gx, calc_t gy, calc_t gz,
              unsigned natoms, unsigned offset)
{
    unsigned index = get_global_id (0);

    if(index < natoms){
     coord_t g;
     g.x =gx * -1;
     g.y = gy * -1;
     g.z = gz * -1;
     inc3coord(speed + index,g,offset);
    }
}
```

2.3.2 Calcul des forces avec tri par boîtes

traitement de la somme prefixée

Comme conseillé sur le sujet, nous avons fait une implémentation de la somme préfixée en dehors du projet. Ainsi, nous avons fait un module prescan, qui calcule len OpenCL , la somme préfixée d'un tableau d'entier. Le noyau OpenCL qui effectue le calcul est présenté ci après :

```
__kernel void prescan(__global const int *A,__global int *B,const uint n)
{
    __local int temp[SIZE]; // allocated on invocation
    int thid = get_local_id(0);
    int offset = 1;
    int last =

    temp[2*thid] = A[2*thid]; // load input into shared memory
    temp[2*thid+1] = A[2*thid+1];

    for (int d = n>>1; d > 0; d >>= 1) // build sum in place up the tree
    {
        barrier(CLK_LOCAL_MEM_FENCE);
        if (thid < d)
        {
            int ai = offset*(2*thid+1)-1;
            int bi = offset*(2*thid+2)-1;
            temp[bi] += temp[ai];
        }
        offset *= 2;

        if (thid == 0) { temp[n - 1] = 0; } // clear the last element
    }
    for (int d = 1; d < n; d *= 2) // traverse down tree & build scan
    {
            offset >>= 1;
            barrier(CLK_LOCAL_MEM_FENCE);
            if (thid < d)
            {
                int ai = offset*(2*thid+1)-1;
                int bi = offset*(2*thid+2)-1;
                float t = temp[ai];
                temp[ai] = temp[bi];
                temp[bi] += t;
             }
    }
    barrier(CLK_GLOBAL_MEM_FENCE);

    B[2*thid-1] = temp[2*thid]; // write results to device memory
    B[2*thid] = temp[2*thid+1];
    barrier(CLK_GLOBAL_MEM_FENCE);
    B[SIZE - 1] = temp[SIZE - 1] + A[SIZE - 1];

}
```

L'algorithme a été pris sur le site `http://http.developer.nvidia.com/GPUGems3/gpugems3_ch39.html` en CUDA et a été adapté et légèrement modifié en OpenCL. N'ayant pas pu intégrer cette fonctionnalité dans le projet, nous nous sommes permis de mettre le module dans notre code source. Le fichier README dit comment le mettre en marche.