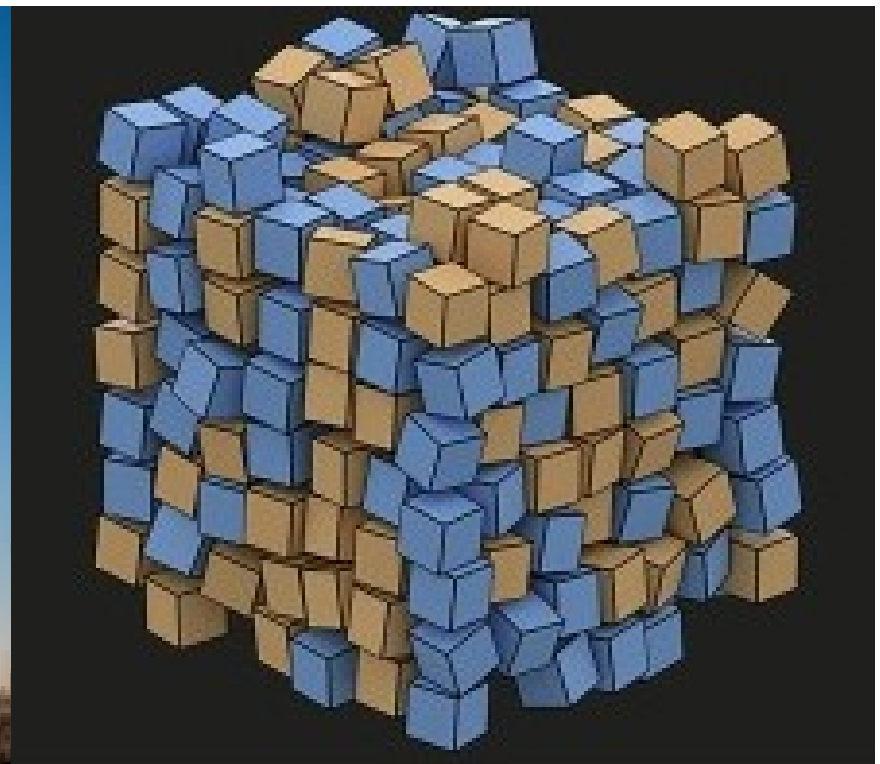


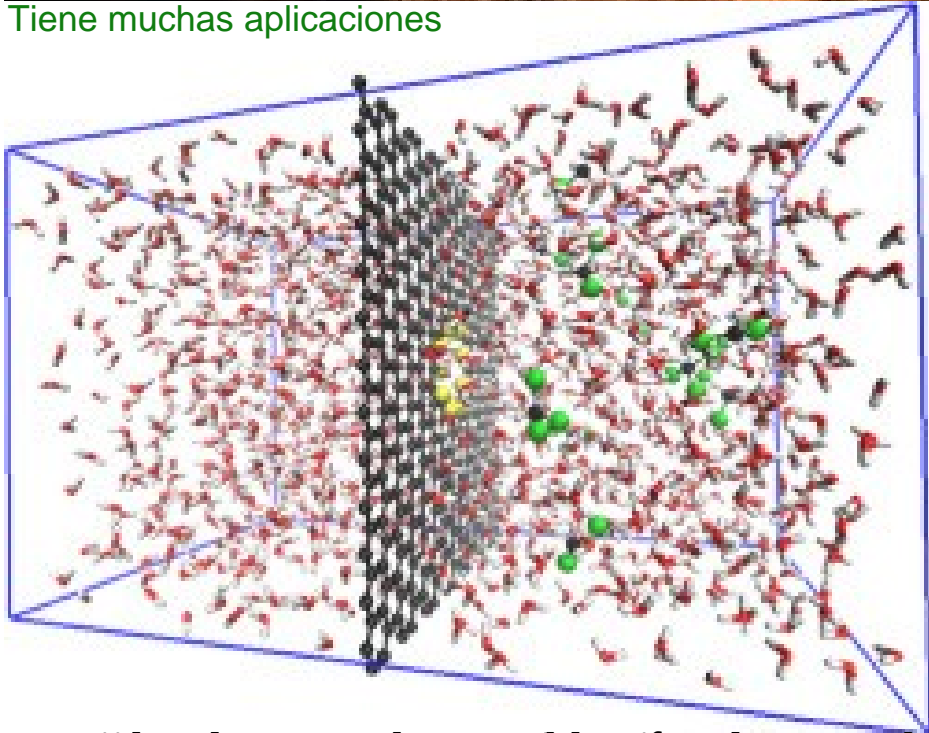
ICNPG 2023

Clase 13: Dinámica molecular





Tiene muchas aplicaciones



<https://developer.nvidia.com/blog/fast-large-scale-agent-based-simulations-on-nvidia-gpus-with-flame-gpu/>

Atractivo a larga distancia y repulsivo a corta distancia

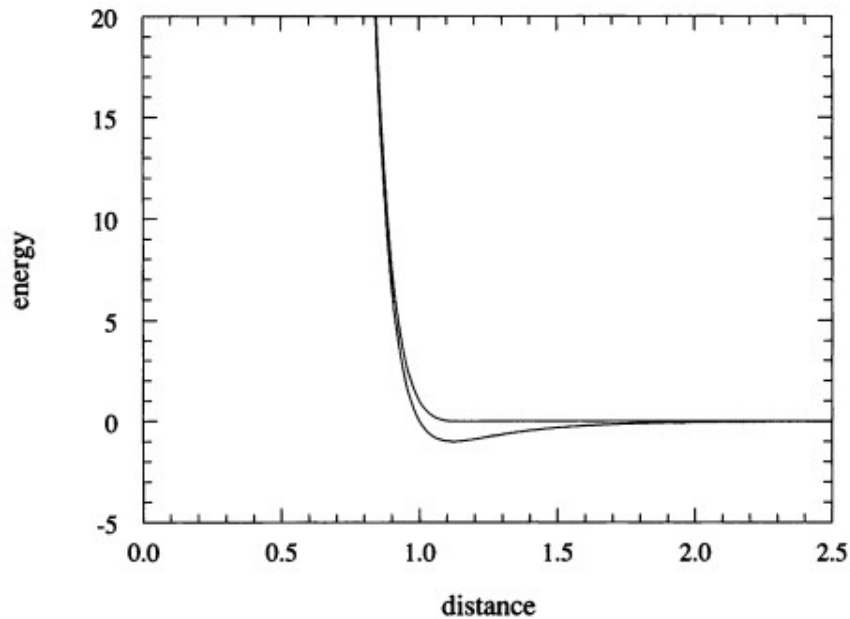
Gas de Lennard-Jones

Ecuaciones de Newton

$$m\ddot{\mathbf{r}}_i = \mathbf{f}_i = \sum_{\substack{j=1 \\ (j \neq i)}}^{N_m} \mathbf{f}_{ij}$$

La parte complicada está en hacer esta cuenta.
Hay que paralelizar ese paso y si uno quiere,
todo el resto

Interacciones de a pares



$$u(r_{ij}) = \begin{cases} 4\epsilon \left[\left(\frac{\sigma}{r_{ij}} \right)^{12} - \left(\frac{\sigma}{r_{ij}} \right)^6 \right] & r_{ij} < r_c \\ 0 & r_{ij} \geq r_c \end{cases}$$

$$\mathbf{f}_{ij} = \left(\frac{48\epsilon}{\sigma^2} \right) \left[\left(\frac{\sigma}{r_{ij}} \right)^{14} - \frac{1}{2} \left(\frac{\sigma}{r_{ij}} \right)^8 \right] \mathbf{r}_{ij}$$


Prácticamente todo el computo esta dominado por el calculo de fuerzas de interacción

Gas de Lennard-Jones

Adimensionalización e integración numérica

$$m\ddot{\mathbf{r}}_i = \mathbf{f}_i = \sum_{\substack{j=1 \\ (j \neq i)}}^{N_m} \mathbf{f}_{ij}$$

length: $r \rightarrow r\sigma$
energy: $e \rightarrow e\epsilon$
time: $t \rightarrow t\sqrt{m\sigma^2/\epsilon}$


$$\ddot{\mathbf{r}}_i = 48 \sum_{j(\neq i)} \left(r_{ij}^{-14} - \frac{1}{2} r_{ij}^{-8} \right) \mathbf{r}_{ij}$$

Leapfrog

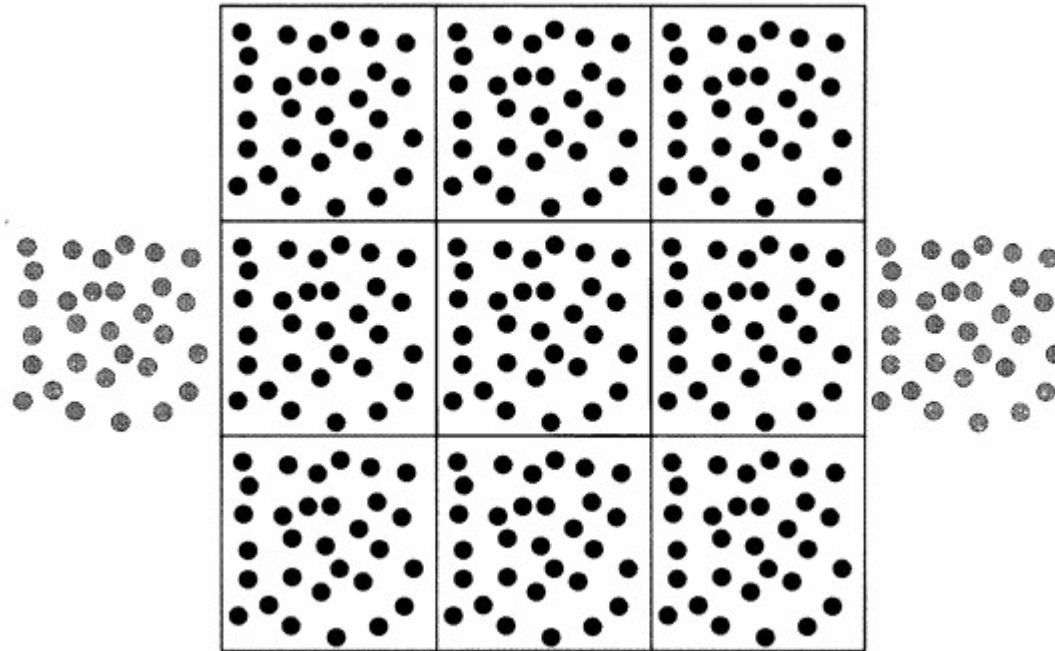
$$v_{i,x}(t + h/2) = v_{i,x}(t - h/2) + h a_{i,x}(t)$$

$$r_{i,x}(t + h) = r_{i,x}(t) + h v_{i,x}(t + h/2)$$

Este método que pertenece a la categoría de "métodos simplécticos", conserva la energía en valor medio. En valor medio porque siempre hay errores numéricos al estar discretizando

Gas de Lennard-Jones

Condiciones
periódicas

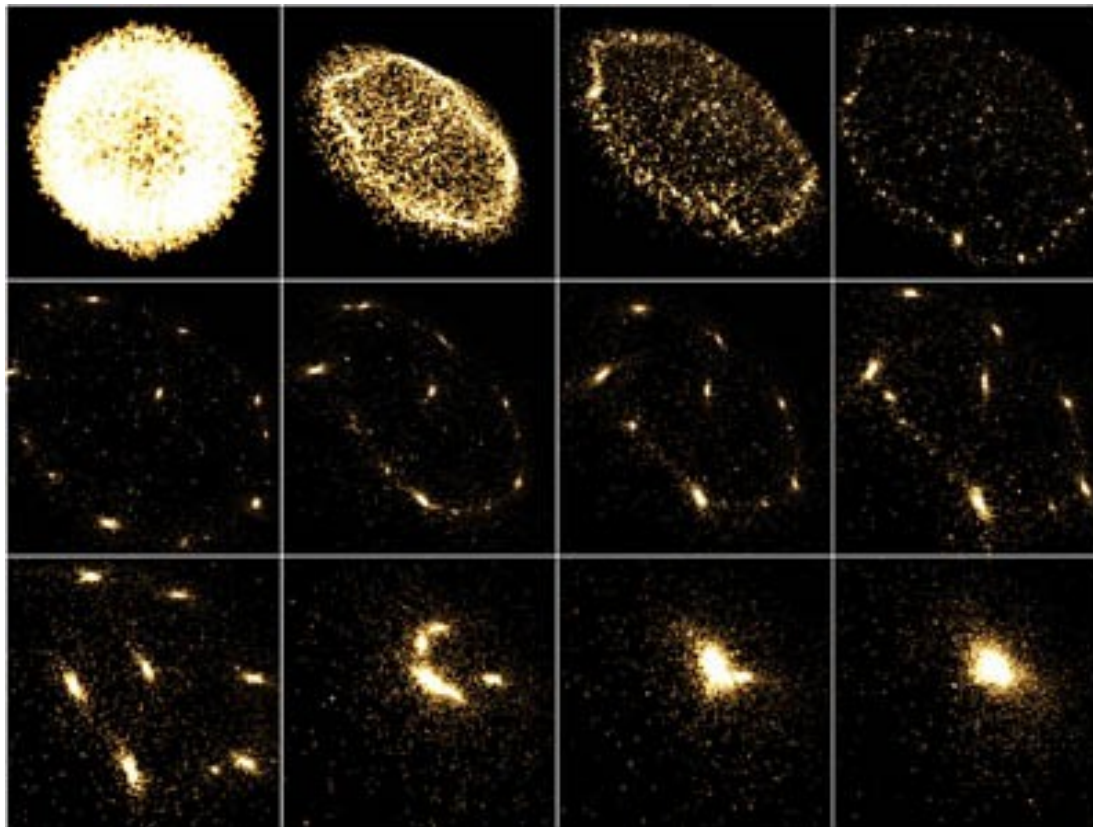


Si uno no considera condiciones
periódicas, estaría teniendo efectos
de borde



Interacciones de largo alcance

- <https://github.com/harrism/mini-nbody/>
- https://github.com/NVIDIA/cuda-samples/tree/master/Samples/5_Domain_Specific/nbody
- <https://developer.nvidia.com/gpugems/gpugems3/part-v-physics-simulation/chapter-31-fast-n-body-simulation-cuda>



nbody

$$U(r) = 1/r$$

No está bien
truncarlo porque si
bien a larga
distancia el
potencial es menor,
también tengo más
partículas

Serial, OMP y OpenACC

- nbody.c Programa serial

```
void bodyForce(Body *p, float dt, int n) {  
    #pragma omp parallel for schedule(dynamic)  
    #pragma acc kernels  
    for (int i = 0; i < n; i++) {  
        float Fx = 0.0f; float Fy = 0.0f; float Fz = 0.0f;  
  
        for (int j = 0; j < n; j++) {  
            float dx = p[j].x - p[i].x;  
            float dy = p[j].y - p[i].y;  
            float dz = p[j].z - p[i].z;  
            float distSqr = dx*dx + dy*dy + dz*dz + SOFTENING;  
            float invDist = 1.0f / sqrtf(distSqr);  
            float invDist3 = invDist * invDist * invDist;  
  
            Fx += dx * invDist3; Fy += dy * invDist3; Fz += dz * invDist3;  
        }  
  
        p[i].vx += dt*Fx; p[i].vy += dt*Fy; p[i].vz += dt*Fz;  
    }  
}
```

Paralelización
en CPU con
openMP

Paralelización
en CPU con
openMP

p es una estructura que tiene (x,y,z)

Para evitar la divergencia cuando $r = 0$

La fuerza es $\vec{f}_{ij} = - \frac{\vec{r}_{ij}}{r_{ij}^3}$

Acá no hay condiciones de borde o, mejor dicho, están "en el infinito".

g++ nbody.c -o nbodycpu

g++ -fopenmp nbody.c -o nbodyomp

pgc++ -acc -ta=tesla nbody.c -o nbodyacc

Comparar performances

Antes de usar este comando hay que cargar el módulo module
load nvhpc-21.9

Para ver info de la paralelización, hay que correr el mismo
comando agregando al final -Minfo

OpenACC mejorado

- nbodyacc.c (mejora usando localidad de datos en device)

```
void bodyForce(Body *p, float dt, int n) {  
    #pragma omp parallel for schedule(dynamic)  
    #pragma acc parallel loop present(p[0:n]) present(p[0:n])  
    for (int i = 0; i < n; i++) {  
        float Fx = 0.0f; float Fy = 0.0f; float Fz = 0.0f;  
  
        for (int j = 0; j < n; j++) {  
            float dx = p[j].x - p[i].x;  
            float dy = p[j].y - p[i].y;  
            float dz = p[j].z - p[i].z;  
            float distSqr = dx*dx + dy*dy + dz*dz + SOFTENING;  
            float invDist = 1.0f / sqrtf(distSqr);  
            float invDist3 = invDist * invDist * invDist;  
  
            Fx += dx * invDist3; Fy += dy * invDist3; Fz += dz * invDist3;  
        }  
  
        p[i].vx += dt*Fx; p[i].vy += dt*Fy; p[i].vz += dt*Fz;  
    }  
}
```

Ya no tiene que hacer copias. Todas las cuentas se harán en GPU

main

```
#pragma acc data copy(p[0:nBodies])  
for (int iter = 1; iter <= nIters; iter++) {
```

pgc++ -acc -ta=tesla nbodyacc.c -o nbodyacc2

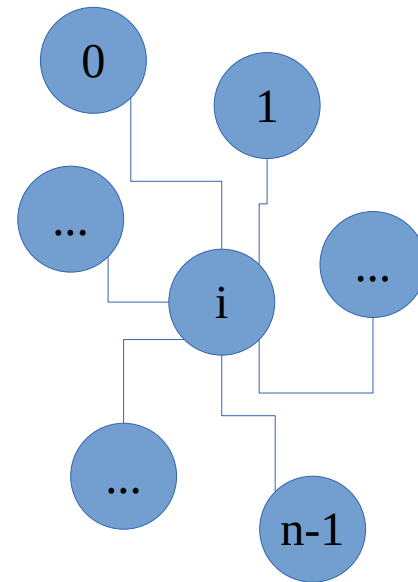
Comparar performances

Cuda naive, todo en global memory

A cada hilo se le da una partícula

```
__global__  
void bodyForce(Body *p, float dt, int n) {  
    int i = blockDim.x * blockIdx.x + threadIdx.x;  
    if (i < n) {  
        float Fx = 0.0f; float Fy = 0.0f; float Fz = 0.0f;  
  
        for (int j = 0; j < n; j++) {  
            float dx = p[j].x - p[i].x;  
            float dy = p[j].y - p[i].y;  
            float dz = p[j].z - p[i].z;  
            float distSqr = dx*dx + dy*dy + dz*dz + SOFTENING;  
            float invDist = rsqrtf(distSqr);  
            float invDist3 = invDist * invDist * invDist;  
  
            Fx += dx * invDist3; Fy += dy * invDist3; Fz += dz * invDist3;  
        }  
  
        p[i].vx += dt*Fx; p[i].vy += dt*Fy; p[i].vz += dt*Fz;  
    }  
}
```

Esto es en memoria privada



nvcc nbody-orig.cu -I../ -o nbody-orig

Para que no se queje del <timer.h>

Hay muchas lecturas de cada partícula, todas en memoria global ...

Esta es la optimización "+ fuerte"

Cuda usando shared memory

```
__global__
void bodyForce(float4 *p, float4 *v, float dt, int n) {
    int i = blockDim.x * blockIdx.x + threadIdx.x;
    if (i < n) {
        float Fx = 0.0f; float Fy = 0.0f; float Fz = 0.0f;

        for (int tile = 0; tile < gridDim.x; tile++) {
            __shared__ float3 spos[BLOCK_SIZE];
            float4 tpos = p[tile * blockDim.x + threadIdx.x];
            spos[threadIdx.x] = make_float3(tpos.x, tpos.y, tpos.z);
            __syncthreads();

            for (int j = 0; j < BLOCK_SIZE; j++) {
                float dx = spos[j].x - p[i].x;
                float dy = spos[j].y - p[i].y;
                float dz = spos[j].z - p[i].z;
                float distSqr = dx*dx + dy*dy + dz*dz + SOFTENING;
                float invDist = rsqrtf(distSqr);
                float invDist3 = invDist * invDist * invDist;

                Fx += dx * invDist3; Fy += dy * invDist3; Fz += dz * invDist3;
            }
            __syncthreads();
        }

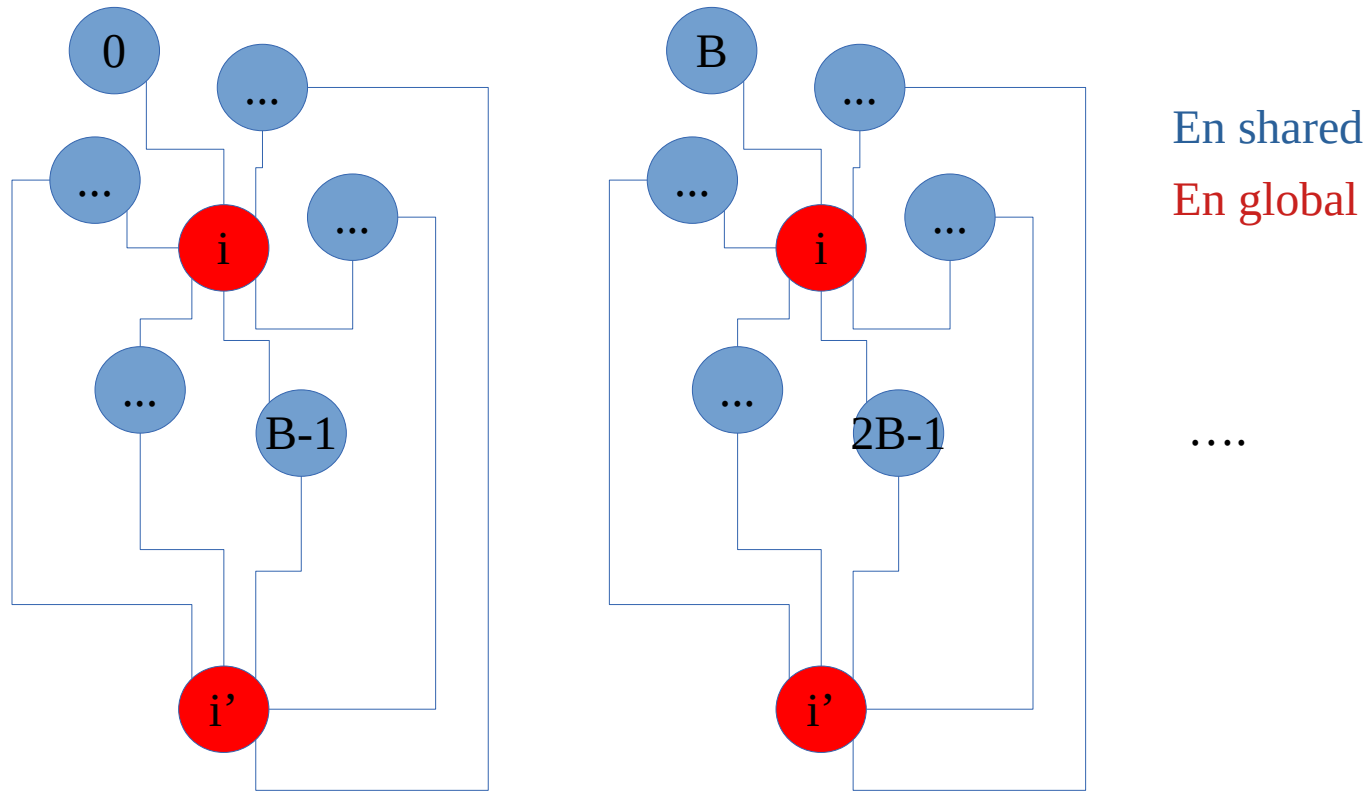
        v[i].x += dt*Fx; v[i].y += dt*Fy; v[i].z += dt*Fz;
    }
}
```

- Cada hilo se ocupa de sumar las fuerzas sobre una partícula *i*.
- Los hilos de un bloque cooperan para leer un "tile" del array global de partículas.
- El tile se guarda en shared-memory *spos*.
- Las fuerzas sobre una partícula del tile son calculadas con *spos* y *tpos*.
- Se continua hasta que no hay mas tiles a procesar.
- Se avanzan las velocidades con las aceleraciones calculadas.

nvcc nbody-block.cu -I../ -o nbody-block

Las fuerzas se van calculando "de a tandas" desde la memoria compartida. Las lecturas ahora están compartidas en una memoria rápida

Cuda usando shared memory



Partículas del mismo bloque i, i' leen el mismo tile de ancho B , uno por uno

- Cada hilo se ocupa de sumar las fuerzas sobre una partícula i .
- Los hilos de un bloque cooperan para leer un “tile” del array global de partículas.
- El tile se guarda en shared-memory *spos*.
- Las fuerzas sobre una partícula del tile son calculadas con *spos* y *tpos*.
- Se continua hasta que no hay mas tiles a procesar.
- Se avanzan las velocidades con las aceleraciones calculadas.

`nvcc nbody-block.cu -I../ -o nbody-block`

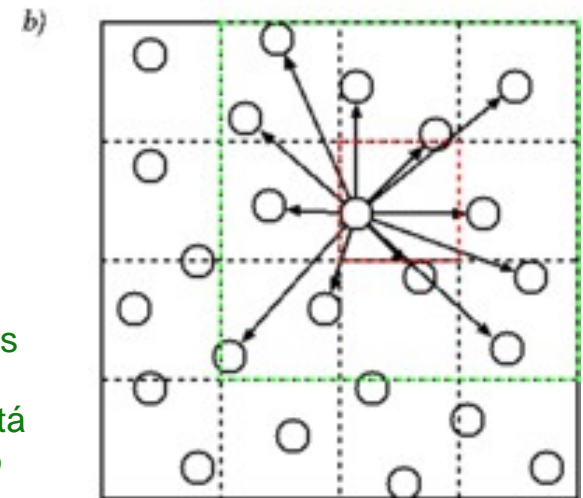
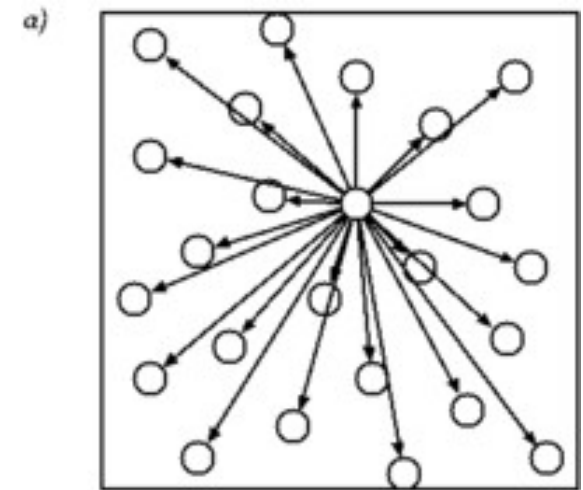
Para correr, luego hay que hacer
`qsub -N cpu jobGPU ./archivo`
Creo que "-N cpu" solo le pone nombre al qsub

Interacciones de corto alcance

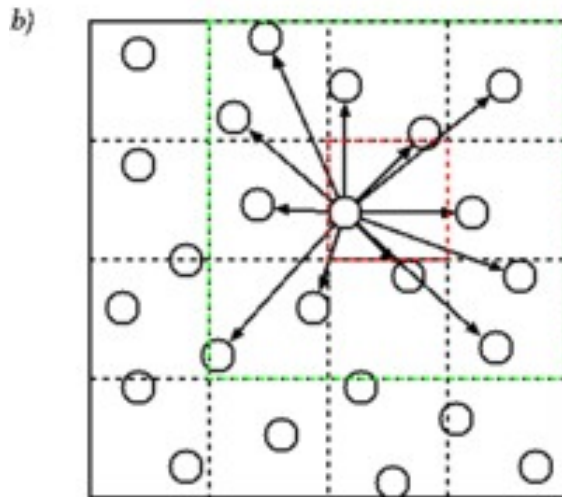
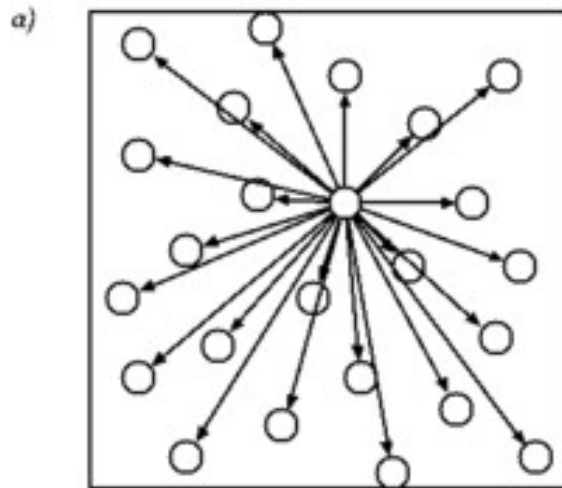
- La interacción se puede despreciar más allá de un cutoff.
- Se pierde tiempo calculando del orden de $N*N$ interacciones que son despreciables.
- Se pierde tiempo preguntando a que distancia están $N-1$ partículas de una dada.
- Una solución es dividir el espacio en celdas de tamaño cutoff y mantener una lista de partículas en cada una. Una dada partícula, interactuará con las partículas de la misma celda o de celdas vecinas. Si la densidad por unidad de volumen es n , interactuará con $N_c \sim n * \text{cutoff}^D$ en vez de con $N \sim n * L^D \gg N_c$ si $L \gg \text{cutoff}$.

Hay un criterio para ver si es de corto o largo alcance.
Depende de la dimensión

Si la densidad es constante, entonces en media el nro de partículas con la que interactúa una dada partícula está más o menos fijo y podría ser mucho menor a N



Interacciones de corto alcance



```

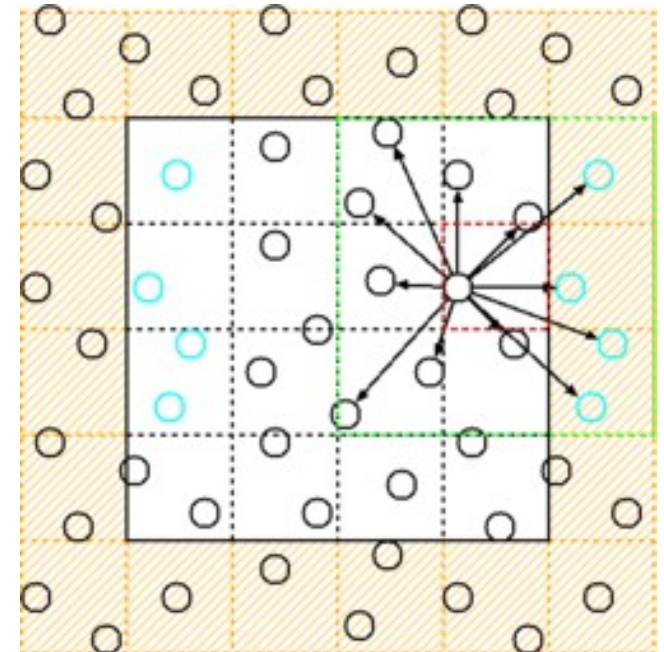
for all neighbouring cell pairs  $(C_\alpha, C_\beta)$  do
  for all  $p_\alpha \in C_\alpha$  do
    for all  $p_\beta \in C_\beta$  do
       $r^2 = \|\mathbf{x}[p_\alpha] - \mathbf{x}[p_\beta]\|_2^2$ 
      if  $r^2 \leq r_c^2$  then
        Compute the interaction between  $p_\alpha$  and  $p_\beta$ .
      end if
    end for
  end for
end for

```

OJO!
Las partículas cambian de celda, así que cada tanto hay que actualizar las listas.

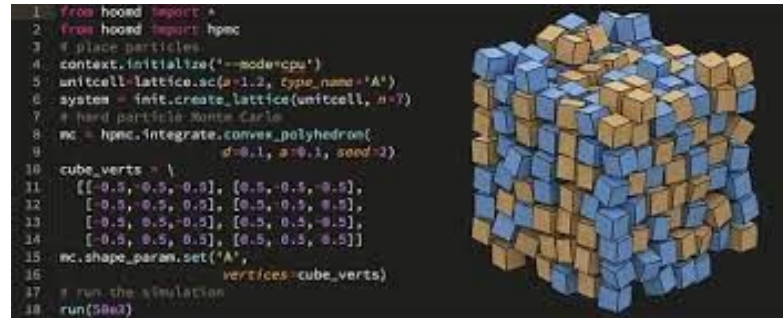
Una optimización extra además de que todas las partículas de una misma celda estén en shared-memory, se puede reordenar la memoria de modo que las partículas que estén cerca en el espacio, estén tmb cerca en memoria. Esto se puede hacer en cupy (hay un ejemplo de notebook en el classroom de la clase 13)

Condiciones de contorno periódicas



Software de dinámica molecular

- NAMD
- Amber
- LAMMPS
- OpenMM
- HOOMD-blue
- ACEMD
- DL_POLY
- GROMOS



*Todos soportan GPU
Algunos Multi-GPU
Algunos Multi-nodo*

